

# 前言

本书是介绍 C++ 编程语言的教科书，同时也是进行 C++ 语言编程的参考书。尽管本书包含了一些编程的相关技巧，但主要还是围绕 C++ 语言的特征进行组织的，而不是针对某些编程技术展开的技术教程。本书主要面向在 C++ 编程语言方面还没有丰富经验的大学本科学生。因此，作为 C++ 编程语言方面的教科书和参考书，本书适合不同层次的读者。本书开始的章节专门为初学者而准备，其中方框中的内容是面向有经验的程序员介绍的 C++ 语言的基本语法。后续章节的编写也考虑到了初学者，但主要是针对进阶读者介绍 C++ 语言的一些高级主题。本书同样适合那些想自学 C++ 语言的读者。（有些读者或许希望能有一本包含更多教学内容和基本编程技巧的教科书，这些读者可以参考我编写的另一本书：*Problem Solving with C++*，第 8 版，Pearson 公司出版。）

本书很全面地介绍了 C++ 语言的相关知识，很多内容超出了初学者应该掌握的范围。例如，本书详细介绍了继承、多态、异常处理和标准模板库（STL）的相关知识。

## 新版变化

第 5 版和第 4 版采用了相同的编程哲学。对于教师而言，无须改变课程各主题的顺序以及各主题对应的章节和具体内容。相比第 4 版，此次更新的内容包括：

- 第 1 章增加了字符串类的简单介绍，第 2 章增加了文本文件数据读取的简单介绍。增加这些内容后，教师可以很方便地从第 2 章开始就向学生引入大规模的或者现实世界中存在的问题。
- 第 12 章增加了对 `stringstream` 的介绍，从而方便在 `string` 类型和其他类型之间做类型转换。
- 第 13 章简捷地介绍了尾递归的概念并且给出了一个相互递归的例子。
- 增加了 10 个自测练习题和 25 个编程项目练习。同时，应读者的要求，部分新增的题目更长且具有更少的限制，给学生留下了更多发挥的空间，可以让他们设计编程方案的能力得到更好的锻炼。
- 更正了第 4 版中出现的几个错误。

## ANSI/ISO C++ 标准

本书介绍的所有内容完全依照最新的 ANSI/ISO C++ 语言标准进行。

## 标准模板库

标准模板库（STL）是一个预先编写好的包含很多数据结构和算法的编程库。STL 的相关知识和内容恐怕和 C++ 核心知识不相上下，因此本书使用了足够多的篇幅介绍



## C++ 基础



### 1.1 C++简介 2

C++语言的起源 2

C++与面向对象编程 3

C++的特点 3

C++术语 3

C++程序示例 3

### 1.2 变量、表达式及赋值语句 5

标识符 5

变量 6

赋值语句 8

string类简介 9

陷阱：未初始化变量 9

提示：采用有意义的变量名 10

更多赋值语句 11

赋值兼容性 11

字面值 12

转义序列 14

命名常量 14

算术运算符和表达式 16

整数和浮点数除法 17

陷阱：全整数除法 17

类型转换 18

自增和自减运算符 20

陷阱：求值顺序 21

### 1.3 控制台输入/输出 22

使用cout输出 22

换行符 23

提示：以\n或endl结束程序 24

格式化浮点数 24

用cerr输出 25

用cin输入 25

提示：输入/输出中的换行 28

### 1.4 编程风格 28

注释 28

### 1.5 库与命名空间 29

库与include命令 29

命名空间 29

陷阱：库名的问题 30

# 第 1 章 C++ 基础

分析机不是什么问题都能解决，它只能根据我们告诉它的方式完成事情。它可以遵循我们给出的分析，但没有能力对新的分析关系或事实进行预测。归根到底，它只是帮助我们去做一些我们已经知道的事情。

奥古斯塔·艾达·洛夫莱斯

## 概述

本章简要介绍 C++ 语言。通过相关知识的介绍，读者可以编写基本的 C++ 程序，其中包括表达式、赋值语句以及控制台输入/输出的使用。表达式和赋值语句的使用与其他高级语言类似。而对于控制台输入/输出语句，每个语言则有自己对应的语法。如果读者是第一次接触 C++，这部分可能会显得不太一样。

## 1.1 C++ 简介

语言是科学唯一的工具。

塞缪尔·约翰逊

本节将对 C++ 语言做出概要的介绍。

### C++ 语言的起源

可以将 C++ 语言看作是加入了面向对象编程思想以及其他特性的 C 语言。C 语言首先由贝尔实验室的 Dennis Ritchie 于 20 世纪 70 年代设计，最初应用于 UNIX 操作系统的编写以及维护中。在此之前，UNIX 操作系统的开发采用汇编语言以及 B 语言——UNIX 操作系统之父 Ken Thompson 所设计的一种编程语言。C 语言是一种通用编程语言，应用非常广泛，可以用来编写各类程序，但是它的成功和流行与 UNIX 操作系统密不可分。因为要维护 UNIX 操作系统，就必须掌握 C 语言。C 语言和 UNIX 操作系统配合得如此完美，以至于没过多久，不仅仅是系统程序，几乎所有运行在 UNIX 操作系统上的商业软件都采用 C 语言来开发。由于 C 语言如此流行，以至于后来产生了可运行于其他操作系统的版本。从此，C 语言的使用也不再局限于 UNIX 操作系统平台，而是广泛用于各个操作系统平台。尽管 C 语言非常流行，但也并非毫无缺陷。

C 语言的独特之处在于，作为一个高级语言，它却包含了许多低级语言才有的特性。它介于严格的高级语言和低级语言之间，兼具两者的优缺点。和低级的汇编语言一样，C 语言可以直接操作计算机的内存。同时又和高级语言一样，相比汇编语言，它又有便于阅读和书写的优点。这种特点使得 C 语言成为编写系统程序的不二选择。然而，相对于其他程序（甚至一部分系统程序）而言，C 语言不如其他语言易于理解，同时它也缺少许多高级语言中的自动检测特性。

为了克服 C 语言的这些缺陷，贝尔实验室的 Bjarne Stroustrup 在 20 世纪 80 年代设计了 C++ 语言。Stroustrup 试图设计一个更好用的 C 语言，因此将其命名为 C++。C++ 包含了绝大多数 C 的特性，是 C 语言的一个超集，这也决定了几乎所有的 C 程序同时也是 C++ 程序。和 C 语言不同，C++ 支持面向对象编程思想，并且提供了相关的内建机制以支持相关的特性。

## C++与面向对象编程

面向对象编程是当前流行的一种功能强大的编程方法，主要的思想包括封装、继承和多态。封装指的是对信息的隐藏和抽象。继承用于编写可复用的代码。多态则是指，同样的名字在不同的继承上下文具有不同的含义。这些概念，对于从来没有接触过面向对象编程的读者可能会“一头雾水”，不过没有关系，本书的后续章节会对以上概念做详尽的介绍。为了支持面向对象编程方法，C++ 提供了类的概念。类是一种包含了数据和方法的数据结构。C++ 并不是有些学者所说的纯面向对象语言，C++ 对面向对象的支持，结合了效率和实用性的考虑。尽管实际的使用中，有些操作并没有完全遵守面向对象编程思想，但是这种结合却使得 C++ 成为当下应用最广泛的面向对象编程语言，虽然它并不是严格支持面向对象编程思想的。

## C++的特点

C++ 通过类的概念来支持面向对象编程思想，允许对函数和运算符进行重载。相对于其他新的面向对象编程语言，C++ 和 C 的联系使得它像是一个传统的面向过程编程语言，但是它比现在流行的其他语言具有更强大的抽象机制。C++ 模板通过类型参数，可以非常方便地对算法进行抽象。最新的 C++ 标准以及大多数的 C++ 编译器支持多重命名空间，从而支持类名和函数名的重复使用。C++ 的异常处理机制和其他语言类似。内存管理和 C 语言类似，程序员自己负责内存的申请和释放。由于 C 语言是 C++ 的一个子集，因此大多数编译器允许 C++ 程序中采用 C 风格的内存管理方式。但是，C++ 有自己风格的内存管理方式，并且在编写 C++ 程序的时候，建议采用 C++ 风格的内存管理方式。本书将只采用 C++ 风格的内存管理。

## C++术语

### 函数 程序

所有过程式的实体都被称为**函数**。其他语言中的过程、方法、函数和子程序，在 C++ 中都被统一称为函数。下一小节，我们将看到，一个最基本的 C++ 程序其实就是一个被称为 main 的函数。程序执行时，运行时系统会自动调用 main 函数。其他 C++ 术语与其他语言都很类似，在用到的时候，本书会对其详加解释。

## C++程序示例

示例 1.1 给出了一个简单的 C++ 程序以及两种可能的运行结果。从示例中可以看到，一个 C++ 程序其实就是一个 main 函数的函数定义。main 函数的函数体包含在 {} 中。程序执行时，main 函数被调用，并执行函数体中的语句。



### 示例 1.1 一个 C++ 程序

---

```

1  #include <iostream>
2  using namespace std;

3  int main( )
4  {
5      int numberOfLanguages;

6      cout << "Hello reader.\n"
7           << "Welcome to C++.\n";

8      cout << "How many programming languages have you used? ";
9      cin >> numberOfLanguages;

10     if (numberOfLanguages < 1)
11         cout << "Read the preface. You may prefer\n"
12              << "a more elementary book by the same author.\n";
13     else
14         cout << "Enjoy the book.\n";

15     return 0;
16 }
```

#### 示例运行结果1

```

Hello reader.
Welcome to C++.
How many programming languages have you used? 0 ← 用户通过键盘输入 0，
Read the preface. You may prefer                    这里用粗体表示。
a more elementary book by the same author.
```

#### 示例运行结果2

```

Hello reader.
Welcome to C++.
How many programming languages have you used? 1 ← 用户通过键盘输入 1，
Enjoy the book                                     这里用粗体表示。
```

---

下面两行代码引入程序中用到的控制台输入/输出库。详细的介绍在 1.3 节、第 9 章、第 11 章和第 12 章。

```

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

`int main()` 下面的代码表明 `main` 是一个无参数、返回值为 `int` 的函数：

```
int main( )
```

有些编译器允许省略上面的 `int` 或者用 `void` 替代，表明该函数没有返回值。但是上面的方式是在 C++ 程序中开始 `main` 函数的最规范形式。

`return 0;` 下面代码的执行标志着程序运行的终止：

```
return 0;
```

这个语句结束 main 函数的运行，并设置函数的返回值为 0。根据 ANSI/ISO C++ 标准，该语句不是必需的，但是目前许多编译器要求该语句。C++ 函数的具体内容将在第 3 章中做详尽的介绍。

C++ 中变量的声明和其他语言类似，示例 1.1 中下边的代码行声明一个 int 类型的变量 numberOfLanguages：

```
int numberOfLanguages;
```

其中，int 代表整数类型，是 C++ 中的一个关键字。

如果读者从未使用过 C++，那么下面的两行代码将显得很陌生。其中涉及控制台输入/输出中 cin 和 cout 的使用。本章的后面会对这部分内容做详细的介绍，这里我们不妨从中获得一个大致的印象。考虑示例 1.1 中的两行代码：

```
cout << "How many programming languages have you used? ";  
cin >> numberOfLanguages;
```

字符串

C 风格的  
字符串

第一行代码输出引号中的文本到屏幕。引号部分的内容被称为字符串，或者更确切地说，C 风格的字符串。第二行要求用户在键盘上输入一个数字，并将其存入变量 numberOfLanguages 中。

再看如下的语句：

```
cout << "Read the preface. You may prefer\n"  
      << "a more elementary book by the same author.\n";
```

以上两行代码输出两个字符串到屏幕。\\n 是换行符，告诉计算机在输出时从新的一行开始。通过以上简短的介绍，读者应该对 cin 和 cout 的使用有了一个大概的了解，详细的介绍会在 1.3 节。

也许到目前为止，你还不清楚如何去写具体的每行代码，但是对于 if-else 语句，或许你已经猜出它的大概意思。关于这部分内容，下一章会做详细介绍。

另外，本书的学习虽然不要求任何 C++ 基础，但一些基本的程序编写基础还是必需的。关于这部分内容，读者可阅读本书的前言部分，以了解相关情况。

## 1.2 变量、表达式及赋值语句

一个人一旦理解了编程中变量的使用方式，那么他也就掌握了编程的精髓。

艾兹格·迪科斯彻，《结构化编程注解》

C++ 中，变量、表达式和赋值语句与其他通用编程语言类似。

### 标识符

标识符

变量以及其他在程序中定义的项目的名字被称为标识符。C++ 中的标识符必须以字母或者下划线开始，其余部分包括字母、数字或者下划线。例如以下示例是合法的标识符：

```
x x1 x_1 _abc ABC123z7 sum RATE count data2 bigBonus
```

以上示例都是合法的标识符，会被编译器接受。但是前五个标识符是比较糟糕的，因为没有表明该标识符的具体用途。以下示例都是非法的标识符：

```
12 3X %change data-1 myfirst.c PROG.CPP
```

前三个没有以字母或者下画线开始。其余的三个包含除了字母、数字和下画线之外的字符。

虽然以下画线开头的标识符是合法的，但是不建议这么做，因为一般来讲，以下画线开头的标识符约定为系统或者标准函数库中的标识符。

区分大小写

C++ 区分大小写，因此以下为三个不同的标识符：

```
rate RATE Rate
```

但是为了避免混淆，不建议将其用在同一个程序中。另外，作为约定，变量的首字母要求小写。预定义标识符，如 `main`、`cin`、`cout` 等，必须全为小写。针对面向对象编程，约定采用大小写混合的方式，变量首字母小写，词界采用大写字母标识。如下所示：

```
topSpeed, bankRate1, bankRate2, timeOfArrival
```

C++ 中，这些约定不像其他语言那么普遍，但是作为一种良好的编程习惯，这些约定应用得越来越广泛。

C++ 中标识符没有长度的限制，但是具体的编译器会忽略一定长度之后的所有字符。

## 标识符

C++ 标识符必须以字母或者下画线开始，其余字符为字母、数字或者下画线。标识符区分大小写，且没有长度限制。

关键字  
保留字

有一类特殊的标识符叫**关键字**或者**保留字**。这些标识符预先在 C++ 中定义，有特殊的用途，不能用作变量名或者其他用途。本书中出现的所有代码，其中关键字都采用不同的颜色给予了标注。附录 A 给出了 C++ 所有关键字的列表。

一些预先定义的标识符，如 `cin` 和 `cout` 不是关键字。它们不是 C++ 语言的核心，可以对它们进行重定义。虽然这些预先定义的标识符不是关键字，但是却在 C++ 的标准库中进行了定义。毫无疑问，将预定义的标识符用作一般用途，会导致很多问题，因此我们应该尽量避免这种情况。最安全和方便的做法就是将所有预定义的标识符都当作关键字。

## 变量

声明

C++ 中每个变量在使用前都必须先**声明**。变量的声明其实是告诉编译器，该变量将存储什么类型的数据。例如，如下的代码：

```
int numberOfBeans;
```

```
double oneWeight, totalWeight;
```

浮点数据

第一行代码声明了变量 `numberOfBeans`，它用来存储一个 `int` 类型的数据。类型 `int` 是 `integer` 的缩写，表示所有的整数。第二行代码声明了两个变量 `oneWeight` 和 `totalWeight`，用来存储 `double` 类型的数据，`double` 标识所有的浮点数据。如上所示，声明中出现多个变量的时候，变量之间采用逗号分隔，且声明采用分号结尾。

所有的变量在使用之前都应首先声明。变量的声明可以在任何地方进行，当然，考虑到程序的可读性，变量应该放在一个合适的地方。一般来讲，变量的声明要么放在变量使用之前，要么放在代码块的起始部分。除了保留字外，所有合法的标识符都可以用作变量名。<sup>1</sup>

C++ 提供了基本的数据类型，如字符、整数、浮点类型。示例 1.2 给出了 C++ 的基本数据类型。其中，`int` 代表整数类型，`char` 表示单个的字符。此外，`char` 类型可以被视为整数类型，但是并不提倡这种用法。浮点类型中用得最多的是 `double`，除非有其他的原因，才用到其他类型的浮点类型。`bool` 类型有 `true` 和 `false` 两个值，它不是整数类型，但为了兼容旧的代码，它可以与任意一种整数类型互相转换。除此之外，程序员还可以定义数组、类以及指针类型，这些类型将在本书后面的章节中进行介绍。

示例 1.2 基本数据类型

类型名	占用内存	取值范围	精度
<code>short</code> ( <code>short int</code> )	2 字节	-32 768 ~ 32 767	-
<code>int</code>	4 字节	-2 147 483 648 ~ 2 147 483 647	-
<code>long</code> ( <code>long int</code> )	4 字节	-2 147 483 648 ~ 2 147 483 647	-
<code>float</code>	4 字节	约 $10^{-38}$ ~ $10^{38}$	7 位
<code>double</code>	8 字节	约 $10^{-308}$ ~ $10^{308}$	15 位
<code>long double</code>	10 字节	约 $10^{-4932}$ ~ $10^{4932}$	19 位
<code>char</code>	1 字节	所有的 ASCII 字符（虽然可以用于存储整数，但并不建议这么做。）	-
<code>bool</code>	1 字节	<code>true</code> , <code>false</code>	-

这里列出的值只是让大家对各个类型的差异有一个大概的认识，不同的系统，会有较大差异。精度是指有意义的数字的位数，包括小数点前的数字。其中 `float`、`double` 和 `long double` 类型的取值范围只给出了正数部分的，负数部分的取值范围类似，只需在每个数字前加上负号。

<sup>1</sup> C++ 中，变量的声明和定义是不同的。声明一个变量时，仅仅是引入该变量的名字和类型；而定义一个变量时，会分配该变量的存储空间。就本章所提到的变量以及本书提及的绝大多数变量，声明一个变量包含了变量的声明和变量的定义，也就是说为变量分配了存储空间。许多作者模糊了变量的声明和定义之间的区别，然而对于某些类型的变量，变量的声明和定义还是有很大不同的，后面的章节会提及这点。

**变量的声明**

所有变量在使用之前都必须首先加以声明。变量声明的语法如下所示。

**语法**

```
Type_Name Variable_Name_1, Variable_Name_2, ...;
```

**示例**

```
int count, numberOfDragons, numberOfTrolls;
double distance;
```

**unsigned**

每种整数类型都有一个 **unsigned** 的版本，这种版本只包含非负值，例如 unsigned short、unsigned int 和 unsigned long。它们在非负值域的取值范围要比 short、int 和 long 大很多（因为它们与相对应的 short、int 和 long 类型相比，占用的存储空间相同，但是不用表示负数域的数值）。一般情况下，我们可能不会用到这些类型，但是会在一些 C++ 库的预定义函数声明中碰到它们。本书的第 3 章会对这一部分做进一步介绍。

**赋值语句****赋值语句**

修改变量值的最直接方法就是使用赋值语句。C++ 采用等号作为赋值语句的运算符。赋值语句包括等号左边的变量、等号以及等号右边的表达式三部分，并且以分号结尾。等号右边的表达式可以是变量、数组，或者由变量、数字运算符以及函数调用组成的更为复杂的表达式。计算机执行赋值语句时，首先会计算等号右边表达式的值，然后将该值赋给等号左边的变量。以下是一些赋值语句的例子：

```
totalWeight = oneWeight * numberOfBeans;
temperature = 98.6;
count = count + 2;
```

**赋值语句**

赋值语句执行时，首先计算等号右边表达式的值，然后将该表达式的值赋给等号左边的变量。

**语法**

```
Variable = Expression;
```

**示例**

```
distance = rate * time;
count = count + 2;
```

第一个赋值语句将 oneWeight 乘以 numberOfBeans 的结果赋给 totalWeight (C++ 中乘法运算符为 \*)。第二个赋值语句将 temperature 的值设为 98.6。第三个赋值语句将 count 的值加 2。

C++ 中，可以将赋值语句用作表达式。此时，该表达式的值为赋值语句中赋给变

量的值。例如，考虑如下的赋值语句：

```
n = (m = 2);
```

表达式 `(m = 2)` 将 `m` 的值改为 2 并返回 2。因此上面的表达式将 `n` 和 `m` 同时设为 2。本书第 2 章会介绍运算符的优先级，这里的括号其实是可以省略的，因此上面的赋值语句可以写为：

```
n = m = 2;
```

建议最好不要将赋值语句用作表达式，但是应该对该用法有清楚的认识，因为有些编程错误与此相关。例如，可以解释当你将：

```
n = m = 2;
```

错误地写为：

```
n = m + 2;
```

的时候，编译器为什么没有报错（这是一个经常发生的错误，因为 `=` 和 `+` 是键盘上的同一个键）。

### 左值和右值

C++ 相关的书籍经常会提及左值和右值。左值是出现在赋值运算符左边的各种类型的变量。右值是出现在赋值运算符右边的各种可求值表达式。

## string 类简介

string

C++ 没有表示字符串的基本类型，但是提供了 `string` 类，用于对字符串进行各种操作和处理。关于基本类型（如 `int` 类型）和类之间的差别，请参照本书第 6 章。另外，本书的第 9 章也会对 `string` 类进行详细介绍。

要使用 `string` 类，必须在程序中引入 `string` 库，可以通过如下代码进行：

```
#include <string>
```

`string` 类型的使用方法与基本数据类型如 `int` 和 `double` 相同。例如，可以通过如下的代码声明一个 `string` 变量，并将字符串 “durian” 存放在其中：

```
string fruit;  
fruit = "durian";
```

### 陷阱：未初始化变量

在程序赋给变量值之前，变量的值是没有任何意义的。例如，变量 `mininumNumber` 的值，如果既没有通过赋值语句进行赋值，也没有以其他方式进行赋值（例如通过 `cin` 语句），那么下面的代码就有问题：

```
desiredNumber = mininumNumber + 10;
```

这是因为 `mininumNumber` 的值没有意义，因此等号右边的整个表达式也没有意

未初始化变量。类似 `mininumNumber` 这样没有赋值的变量被称为**未初始化变量**。事实上，变量没有初始化比变量没有值更糟糕。未初始化变量，如 `mininumNumber`，含有一些毫无意义的值，这些值由对应的内存状态决定（或者是内存的一种随机状态，或者是最近一个使用这部分内存的程序留下来的）。

避免未初始化变量的一个方式是在变量声明的同时对变量进行初始化。可以在变量声明语句的后边加上一个等号和要初始化的值，如下所示：

```
int mininumNumber = 3;
```

上面的代码声明了 `int` 类型的变量 `mininumNumber` 并将其初始化为 3。除了简单的值外，还可以采用更复杂的表达式（如包括加法、乘法等运算）来对变量进行初始化。看另外一个例子，如下的代码声明了三个变量，并对其中的两个进行了初始化：

```
double rate = 0.07, time, balance = 0.00;
```

C++ 提供了另外一种方式，对变量进行声明和初始化。这种方式如下所示，它与之前的代码作用完全相同。

```
double rate(0.07), time, balance(0.00);■
```

## 变量初始化

可以在声明变量的同时对该变量进行初始化。

### 语法

```
Type_Name Variable_Name_1 = Expression_for_Value_1,
Variable_Name_2 = Expression_for_Value_2, ...;
```

### 示例

```
int count = 0, limit = 10, fudgeFactor = 2;
double distance = 999.99;
```

### 语法

初始化的另外一种方式：

```
Type_Name Variable_Name_1(Expression_for_Value_1),
Variable_Name_2(Expression_for_Value_2), ...;
```

### 示例

```
int count(0), limit(10), fudgeFactor(2);
double distance(999.99);
```

**提示：**采用有意义的变量名

程序中变量以及其他标识符的名字应该能体现出它们所代表的含义，这样可以提高程序的可读性。对比如下代码：

```
x = y * z;
```

和

```
distance = speed * time;
```

两条语句完全等价，很明显第二条语句更容易理解。■

### 更多赋值语句

C++ 提供了一种快捷方式将赋值操作与算数操作结合在一起，这样可以对一个给定的变量进行操作，使其增加、减少、乘以以及除以一个特定的值。使用方法如下：

```
Variable Operator = Expression
```

等价于：

```
Variable = Variable Operator (Expression)
```

其中的 *Expression* 可以是一个变量、常数或者一个复杂的算数表达式。下面是一些例子。

示例	等价形式
<code>count += 2;</code>	<code>count = count + 2;</code>
<code>total -= discount;</code>	<code>total = total - discount;</code>
<code>bonus *= 2;</code>	<code>bonus = bonus * 2;</code>
<code>time /= rushFactor;</code>	<code>time = time / rushFactor;</code>
<code>change %= 100;</code>	<code>change = change % 100;</code>
<code>amount *= cnt1 + cnt2;</code>	<code>amount = amount * (cnt1 + cnt2);</code>

### 自测练习题

1. 声明两个 `int` 类型的变量 `feet` 和 `inches`，两个变量都初始化为 0，给出相关的程序语句。
2. 声明两个变量 `count` 和 `distance`，变量类型分别为 `int` 和 `double`，并分别初始化为 0 和 1.5，给出相关程序语句。
3. 编写一个程序，其中包含 5 ~ 6 个未初始化的变量，输出这些变量的值。编译并运行这个程序，解释程序的输出。

### 赋值兼容性

很明显，一种类型的值不能存储在另外一种类型的变量中。例如，大多数编译器不会接受如下的代码：

```
int intVariable;  
intVariable = 2.99;
```

问题就出在类型不匹配上，常数 2.99 是一个浮点数，属于 `double` 类型，而变量 `intVariable` 则是 `int` 类型。对于这种类型不匹配的情况，不同的编译器处理方



式不同。有些会给出错误信息；有些只是给出警告信息；有些编译器甚至会接受这种赋值，不给出任何提示信息。即使有些编译器接受以上的赋值，变量 `intVariable` 中存储的值也只是 2，而不是 3。既然你不能指望编译器去接受上面的赋值，那么最好的做法就是不要进行这种类型不匹配的赋值。

即使你使用的编译器允许这种类型不匹配的赋值操作，也不推荐这么做。因为这么做，会使程序难以移植且让别人难以理解。

但是在一些特殊的情况下，可以进行这种类型不匹配的赋值操作。可以将一个整数类型的值，如 `int` 类型，赋给一个浮点类型的变量，比如 `double` 类型。例如，下面是合法的代码：

```
double doubleVariable;
doubleVariable = 2;
```

以上代码将变量 `doubleVariable` 的值设为 2.0。

虽然不推荐这么做，但是的确可以将一个 `int` 类型的值，如 65，存放在一个 `char` 类型的变量中；另外也可以将一个字母，如 'z'，存放在一个 `int` 类型的变量中。由于众多原因，C 语言将字符看作小整数，而 C++ 语言又继承了这一点。其中的原因就在于相对 `int` 类型，`char` 类型需要更少的内存空间，因此采用 `char` 类型进行数学运算更节省内存。但是很明显，为了不致引起混淆，对整数类型的变量还是尽量采用 `int` 类型，对字符类型的变量还是采用 `char` 类型。

一般的规则是，不要将一种类型的值放在另外一种类型的变量中，当然也有例外，而且似乎这种例外情况出现得更多。即使编译器对这条规则不做严格的检查，仍然建议严格遵循这条规则。将一种类型的值存放在另外一种类型的变量中会导致一些问题，因为编译器会对这个值做相应的改变以适应变量的类型，而这个改变后的值往往不是我们所期望的。

`bool`（布尔）类型的值可以存放在整数类型的变量中，同时整数类型的值可以存放在 `bool` 类型的变量中。其中的细节如下：将一个整数赋给 `bool` 类型的变量时，任何非零的整数都会被转化为 `true`，而零会被转化为 `false`。将一个 `bool` 类型的值赋给一个整型变量时，`true` 被转化为 1，而 `false` 则转化为 0。显然，这是一种非常不好的编程风格，因此不建议使用。

## 字面值

### 字面值

字面值是一种具体的值。相对于变量来讲，字面值是一种常量。常量的值不会改变，而变量的值可以改变。整型常量的书写方式与我们平时使用的数字没有什么不同。`int` 类型的常量（或者其他整型常量）不能包含小数点。`double` 类型的常量有两种表示方式。简单的方式就如日常书写小数的方式，这种方式必须包含小数点。C++ 中数字常量不能含有逗号。

另外一种表示 `double` 常量的方法被称为科学计数法或者浮点记法，是针对特别大或者特别小的数专门设计的。例如  $3.67 \times 10^{17}$ ，也就是

```
367000000000000000.0
```

将 `int` 类型的值赋给 `double` 类型的变量

类型的混合

整数类型和布尔类型

科学计数法  
浮点记法

在C++中最好表示成 $3.67\text{e}17$ 。 $5.89 \times 10^{-6}$ 也就是 $0.00000589$ 最好表示成 $5.89\text{e}-6$ 。其中e代表指数，表示乘以10的多少次方，e既可以大写也可以小写。

可以将字母e后边的数字理解成小数点移动的位数和方向。例如，要将 $3.49\text{e}4$ 转化为常规不带e的表示方式，只需将小数点向右移动四位，从而得到 $34900.0$ 。如果e后边的数字是负数，那么小数点应该向左移动相应的位数。例如 $3.49\text{e}-2$ 等价于 $0.0349$ 。

字母e之前的数字可以包含小数点，也可以没有。但是e之后的数字一定为整数，不能含有小数点。

### 何谓 double 类型？

为何带有小数部分的数字类型称为double类型？有没有一种称为single的类型，其大小是double类型的一半？答案是不存在，但是存在一种类型，其大小是double的一半。许多编程语言对带有小数部分的数字自始就采用两种表示方式。第一种方式占用较少的存储空间，但精度较低。第二种方式使用两倍的存储，因此精度较高的同时具有较大的表示范围。使用两倍内存表示的数字称为双精度数字，而采用占用内存较少方式表示的数字称为单精度数字。遵循这样的传统，C++中，对应双精度的类型称为double类型，对应单精度的类型称为float。C++还有第三种带有小数部分的数字类型，称为long double。

char类型的常量是采用带单引号的字符进行表示的，如下所示：

```
char symbol = 'Z';
```

需要注意，字符左边和右边的引号完全是一个字符。

字符串常量采用双引号括起来的字符串表示，以下代码是例1.1中的一行，代码向屏幕输出一个字符串常量：

```
cout << "How many programming languages have you used? ";
```

#### 引号

切记，字符串常量是用双引号括起来的，而字符常量则是用单引号括起来的，引号的不同代表了不同的意思。'A'和"A"有不同的含义。'A'是一个字符常量，可以存放在一个char类型的变量中。而"A"是一个字符串常量，只不过这个字符串只有一个字符。另外需要注意的是，不管是字符常量还是字符串常量，左右两边的引号都是同一个字符。

#### C风格的字符串

双引号括起来的字符串，如"Hello"，经常被称为C风格的字符串。C风格的字符串和本书之前介绍的string类，虽然都是用来存储字符序列的，而且有时候也可以互换使用，但是这两者还是有许多差异的。具体的差异我们会在第9章进行介绍。考虑到安全性和灵活性，建议还是尽可能多地使用string类来操作字符串，而不是C风格的字符串。

bool类型有两个常量，true和false。这两个常量可以赋给任何一个bool类型的变量或者出现在任何一个需要bool类型表达式的地方。需要注意的是，这两个变量都为小写。

## 转义序列

### 转义序列

反斜杠\告诉编译器，紧跟其后的字符含义发生了变化，这样的序列被称为转义序列。其中反斜杠和字符之间没有空格。C++ 中已经预定义了许多转义序列。

如果想在字符串常量中包含一个反斜杠或者双引号，必须用“\\”代替“\”，用“\n”代替“n”，以便对反斜杠和双引号进行转义。

字符串中一个零散的反斜杠，比如\z，在不同的编译器中含义不同。有些仅仅是简单地返回z，有的则认为这是一个错误。ANSI/ISO 标准指明，未指定的转义序列具有未定义的行为，这意味着编译器可以做任何合适的事情，结果造成使用了未定义转义序列的代码不可移植。因此，不建议使用 C++ 标准之外的转义序列。C++ 的转义序列如示例 1.3 所示。

示例 1.3 一些转义序列

转义序列	含义
\n	换行符
\r	回车
\t	水平制表符
\a	警告
\\	反斜杠
\'	单引号
\"	双引号
下面几个很少使用：	
\v	垂直制表符
\b	空格
\f	换页
\?	问号

## 命名常量

程序中的数字存在两个问题。第一个问题是搞不清楚其所代表的含义。例如某个程序中出现的数字 10，如果该程序属于银行业领域，那么该数字有可能是该银行分部的数目，或者是总部出纳窗口的数目。为了理解这个程序，我们必须知道每个常量的含义。存在的第二个问题是，当程序要改变一些数字的值时，会经常出现一些错误。假设数字 10 在一个银行业程序中出现了 12 次，其中的 4 次代表分部的数目，另外 8 次代表总部出纳窗口的数目。当该银行开设一家分部需要更新程序时，就很可能会出现某些 10 应该变为 11 但却没改，而某些 10 不应该改动却被修改的情况。为了避免这种问题，应该为每个数字命名，然后用名称代替程序中出现的数字。例如，上面提

到的银行业程序应该有两个常量：BRANCH\_COUNT 和 WINDOW\_COUNT。

这两个常量开始时都为 10，当该银行开设一家新的分部时，只需重新定义 BRANCH\_COUNT 就可以对程序进行更新。

如何在 C++ 中对一个数字命名？一个方法是定义一个变量并将其初始化为该数字。例如：

```
int BRANCH_COUNT = 10;
int WINDOW_COUNT = 10;
```

但是，这种方法存在一个问题，有可能一不小心改变这些变量的值。C++ 提供了另外一种方法，可以将一个已初始化的变量标记为不可变。当程序尝试去改变这些变量时，就会产生错误。要使一个变量不可变，只需在该变量声明之前添加 const 关键字。例如：

const

```
const int BRANCH_COUNT = 10;
const int WINDOW_COUNT = 10;
```

如果变量的类型一致，也可以将以上两行代码合并为一行：

```
const int BRANCH_COUNT = 10, WINDOW_COUNT = 10;
```

但是为了清楚起见，大多数程序员都选择将每一个定义单列成行。关键字 const 又被称为**修饰符**，因为它改变了被修饰变量的性质。

修饰符  
修饰常量

由 const 修饰符修饰的变量通常被称为**修饰常量**。尽管 C++ 并不要求将声明常量全部大写，但这却是程序员的一个默认风格。

一个数字一旦以这种方式被命名，所有使用该数字的地方都可以无差别地用该名字进行替换。要改变一个声明常量，只需在 const 变量的声明中改变初始化的值即可。例如，为了将 BRANCH\_COUNT 的值由 10 改成 11，只需在 BRANCH\_COUNT 声明常量的声明中将初始化值改为 11。

示例 1.4 给出了一个使用 const 修饰符的简单程序。

#### 示例 1.4 命名常量

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      const double RATE = 0.09;
7      double deposit;
8
9      cout << "Enter the amount of your deposit $";
10     cin >> deposit;
11
12     double newBalance;
13     newBalance = deposit + deposit*(RATE/100);
14     cout << "In one year, that deposit will grow to\n"
15         << "$" << newBalance << " an amount worth waiting for.\n";
16
17     return 0;
18 }
```

## 示例运行结果

```
Enter the amount of your deposit $100
In one year, that deposit will grow to
$106.9 an amount worth waiting for.
```

## 算数运算符和表达式

和其他编程语言一样，C++ 允许以变量、常量和算数运算符（包括加、减、乘、除及取余）构造表达式。这些表达式可以用在任何该表达式的值可以用到的地方。

## 类型的混合

所有算数运算符都可以应用于 `int` 类型、`double` 类型以及两种类型的混合。但是，最终结果的类型以及最终的结果依赖于参加运算的数字的类型。如果两个操作数都为 `int` 类型，那么算数运算的结果也为 `int`。如果两个操作数中有一个是 `double` 类型，那么运算的结果就是 `double` 类型。例如，如果变量 `baseAmount` 和 `increase` 为 `int` 类型，那么以下表达式的值类型为 `int`：

```
baseAmount + increase
```

如果这两个操作数中有一个为 `double` 类型，那么结果就为 `double` 类型。将其中的运算符换成 `-`、`*`、`/`，同样也成立。

一般来讲，表达式可以包含任意算数类型，如果表达式中的所有操作数都为 `int` 类型，那么该表达式的最终结果也为 `int` 类型。如果其中有一个子表达式为浮点类型，那么该表达式的最终结果就为浮点类型。C++ 会尽量使表达式的值或者为 `int` 类型，或者为 `double` 类型。不过由于值大小的原因，最终的结果不是以上这些类型时，C++ 会生成一个合适的整型或者浮点类型。

## 优先级顺序

通过添加括号的方式，我们可以改变算数表达式中运算进行的顺序。如果没有括号，计算将按照优先级顺序进行，如加法和乘法谁先执行等。这里的优先级顺序与算数以及其他数学领域中用到的类似。例如：

```
x+y* z
```

上面的算数表达式，首先进行乘法运算，再进行加法运算。除了一些标准情况，如一连串的加法运算或者包含简单乘法运算的加法运算，不用添加括号外；其他情况，即使不会改变运算的顺序，建议在编写程序时总要加上括号，以提高程序的可读性，并减少编程错误。附录 B 给出了 C++ 中完整的优先级顺序。

使用 `const` 修饰符命名常量

在声明变量的同时对其初始化时，可以在变量声明前面添加一个 `const` 修饰符，将该变量标记为不可修改。

## 语法

```
const Type_Name Variable_Name = Constant;
```

## 示例

```
const int MAX_TRIES = 3;
const double PI = 3.14159;
```

## 整数和浮点数除法

### 整数除法

当两个操作数中有一个是 `double` 类型时，除法操作如我们预期得那样运行。但如果两个操作数都为 `int` 类型时，除法操作只返回运算结果的整数部分。也就是说，整数除法运算将小数点后的结果全部丢弃掉了。因此， $10/3$  的结果为 3（而不是  $3.33333\cdots$ ）， $5/2$  的结果为 2（而不是 2.5）， $11/3$  的结果为 3（而不是  $3.6666\cdots$ ）。注意，结果并没有进行四舍五入，小数点后边的结果不管其大小，统统丢弃。

### 求余运算符

求余运算符应用于 `int` 类型的数字，可以弥补整数除法丢失的信息。当对两个 `int` 类型的数字进行运算时，采用在中学学过的长除法，同时采用 `/` 和 `%` 会得出两个数。例如，17 除以 5 得到商 3 和余数 2。整数除法操作 `/` 给出商部分结果，而 `%` 操作给出余数部分结果。例如以下代码：

```
cout << "17 divided by 5 is " << (17 / 5) << "\n";
cout << "with a remainder of " << (17 % 5) << "\n";
```

产生如下结果：

```
17 divided by 5 is 3
with a remainder of 2
```

### 负整数的 除法操作

对于负整数而言，`/` 操作和 `%` 操作的结果会因 C++ 实现的不同而不同。因此，应该在确保两个操作数都为非负整数的情况下，才对其采用 `/` 和 `%` 操作。



## 陷阱：全整数除法

对两个整数采用 `/` 操作，产生的结果也是一个整数，这可能与期望的浮点数结果不太一样。另外，由于这种情况没有任何的提示和警告，很容易被忽略，因此往往看似正确的程序却产生了错误的结果。例如，假设你是一名环境美化工程师，准备美化一条高速公路，每英里收费 5000 美元，同时你也知道该高速公路的总长度（记为 `feet`）。那么该高速公路的最终收费很容易通过以下的代码进行计算：

```
totalPrice = 5000 * (feet/5280.0);
```

由于 1 英里为 5280 `feet`，因此以上代码是正确的。如果该高速公路的长度是 15 000 `feet`，那么总收费将会是：

```
5000 * (15000/5280.0)
```

C++ 通过计算  $15000/5280.0$  得到 2.84，之后 5000 乘以 2.84 得到最终的结果 14200.00。通过该程序，你知道对该高速公路的最终收费是 14 200 美元。

接下来，假设变量 `feet` 的类型是 `int` 类型，并且我们忘记了 5280.0 中的小数点和末尾的 0，这样一来，以上的代码将变为：

```
totalPrice = 5000 * (feet/5280);
```

这样看上去似乎没有问题，但实际上却会导致严重的问题。如果你使用以上的代码进行计算，由于是对两个 `int` 类型的整数做除法运算， $feet/5280$  的结果将是 2，而不是我们所期望的 2.84。最终 `totalPrice` 的值将会是  $5000 \times 2$ ，即 10 000.00。也就是，由于忘记了小数点，导致该工程的收费从 14 200 美元变为 10 000 美元，从而

使得你损失了 4200 美元。请注意, 由于计算结果的错误发生在赋值给变量 `totalPrice` 之前, 因此不管 `totalPrice` 是 `int` 类型还是 `double` 类型, 最终的错误都不可避免。■

### 自测练习题

4. 将以下数学公式转化为 C++ 表达式:

$$3x \quad 3x + y \quad \frac{x + y}{7} \quad \frac{3x + y}{z + 2}$$

5. 以下程序片段, 所有的变量都为 `char` 类型, 那么最终的输出是什么?

```
a = 'b';
b = 'c';
c = a;
cout << a << b << c << 'c';
```

6. 以下程序片段, 变量 `number` 为 `int` 类型, 该程序片段的输出是什么?

```
number = (1/3) * 3;
cout << "(1/3) * 3 is equal to " << number;
```

7. 编写一个可运行的程序, 该程序从键盘读入两个数, 存入两个 `int` 类型的变量。然后第一个变量除以第二个变量, 分别输出结果的商部分和余数部分。
8. 以下程序片段, 用来将摄氏温度转化为华氏温度, 阅读该程序, 回答以下问题:

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
```

- `f` 的值是多少?
- 解释实际得到的结果是什么? 程序员希望得到的结果又是什么?
- 根据程序的意图, 重新编写正确的程序。

## 类型转换

### 类型转换

**类型转换**是指将一种类型的数值转换为另外一种类型的值。类型转换是一种特殊的函数, 它的输入参数是某种类型的数值, 而输出则是另一种类型的数值, 并且这两个值在 C++ 编译器看来是最接近的。C++ 语言中共有 4~6 种不同的类型转换。旧的类型转换有两种使用方式, 而最近的新标准给出了四种新的类型转换方式, 是对旧有使用方式的一种替代。本书重点介绍新标准下的类型转换方法, 对旧有的类型转换方法只进行简单介绍。

首先看一下新标准中的类型转换方法。考虑这样一个表达式:  $9/2$ 。由于参与除法操作的两操作数都为整数, C++ 中, 该表达式的结果为 4。为了得到 4.5 这样的结果, 可以将表达式中的 2 用浮点类型 2.0 进行替换, 将之前的表达式转换为  $9/2.0$ 。考虑一般情况,  $m/n$ , 其中 `m` 和 `n` 为 `int` 类型的整型变量, 当 `m` 和 `n` 分别为 9 和 2 时, 就特殊化为之前的表达式了。为了使  $m/n$  进行浮点除法操作, 就必须进行类型转换, 将 `int` 类型转换为浮点类型。如下所示:

```
double ans = n/static_cast<double>(m);
```

其中 `static_cast<double>(m)` 就是一个类型转换表达式。表达式 `static_cast<double>(m)` 类似一个函数，带有一个 `int` 类型的参数，返回一个“相等”的 `double` 类型值。如果 `m` 的值为 2，表达式 `static_cast<double>(m)` 将返回 `double` 类型的值 2.0。

需要注意的是，表达式 `static_cast<double>(n)` 并不会改变输入变量 `n` 的值。如果该表达式求值之前，`n` 的值为 2，那么表达式求值之后，`n` 的值依然为 2。（如果你了解数学和某些编程语言中函数的作用，那么你应该可以看到，`static_cast<double>` 类似于一个返回“相等” `double` 类型值的函数。）

可以将上式中的 `double` 替换为其他任何数据类型，从而实现从一种类型到另一种类型的转换。这一过程是一种“等价”转换过程，但是对于不同的类型，该等价的含义不尽相同。在将整数类型转换为浮点类型时，其效果是在整数后加上小数点和数字 0。而将浮点数转换为整数时，则是简单地去掉小数点以及其后的小数部分。这里需要注意的是，将浮点数转换为整数时，会直接舍弃小数部分，而不会进行四舍五入。例如 `static_cast<int>(2.9)` 的结果是 2，而不是 3。

`static_cast` 是一类最常用的类型转换方法。除此之外，还有其他三种不常用的类型转换方法。这里我们给出 C++ 中四种类型转换的方法，如下所示：

```
static_cast<Type>(Expression)
const_cast<Type>(Expression)
dynamic_cast<Type>(Expression)
reinterpret_cast<Type>(Expression)
```

如前文所述，`static_cast` 最经常用到，在大多数场合都可以进行一般的类型转换。`const_cast` 用于去除变量的 `const` 属性。`dynamic_cast` 用在继承层次中，将一种类型安全地转换为派生类型。`reinterpret_cast` 很少使用，而且其含义与具体的编译器相关，因此不在本书讨论范围之内。（`reinterpret_cast` 的深入讨论和理解涉及一些本书不会讨论的 C++ 主题。就本书而言，我们只会用到 `static_cast`。）

旧的类型转换方法基本上等价于上文提到的 `static_cast`，共有两种用法。第一种用法将类型名像函数那样使用。例如，`int(9.3)` 返回整数 9，`double(42)` 返回浮点 42.0。第二种用法会将 `double(42)` 写作 `(double)42`。每种用法都可以与变量或者复杂的表达式合用。

虽然 C++ 保留了这种旧的类型转换方法，但是推荐使用新类型转换方法。（虽然目前没有确切消息，但或许某一天旧的类型转换方法就会被淘汰。）

正如之前所说，可以将一个整型的数赋给一个浮点类型的变量，如：

```
double d = 5;
```

这种情况下，C++ 会自动进行类型转换，从而将整数 5 转换为浮点数 5.0，再将其存入变量 `d` 中。如果不进行这种类型转换，整数 5 是不可以存放到变量 `d` 中的。我们将这种自动类型转换称为**强制类型转换**。



## 自增和自减运算符

### 自增运算符 自减运算符

C++ 语言名字中的 ++ 来源于自增运算符 ++。自增运算符给当前的变量值加 1。自减运算符则给当前的变量值减 1。它们可以用于所有的数字类型，但一般用于整数类型。如果 n 是一个数字类型的变量，那么 n++ 将 n 的值增加 1，n-- 将 n 的值减 1。因此以分号结尾的 n++ 和 n--（后面有一个分号）是可以执行的程序语句。例如，以下的语句：

```
int n = 1, m = 7;
n++;
cout << "The value of n is changed to " << n << "\n";
m--;
cout << "The value of m is changed to " << m << "\n";
```

会有如下的输出：

```
The value of n is changed to 2
The value of m is changed to 6
```

表达式 n++ 在返回变量 n 值的同时还将 n 的值加 1。因此 n++ 可以用于算数表达式中，如：

```
2*(n++)
```

上式中，表达式 n++ 首先返回变量 n 的值，然后将 n 的值加 1。考虑如下的代码：

```
int n = 2;
int valueProduced = 2*(n++);
cout << valueProduced << "\n";
cout << n << "\n";
```

将会有如下的输出：

```
4
3
```

注意其中的表达式：2\*(n++)。C++ 在计算该表达式时，首先是 n 的值乘以 2，然后 n 的值再加 1 变为 3。这看起来有点奇怪，但有时这样的结果恰是我们所希望的。另外，下面马上会讲到，如果你希望的是另外一种表达式计算方式，有另外一种方式可以做到这一点。

### v++ 与 ++v

表达式 n++ 返回变量 n 的值，然后变量 n 的值加 1。如果将 ++ 放在变量 n 之前，如 ++n，那么这两个步骤的顺序会发生变化。表达式 ++n 首先将变量 n 的值加 1，然后返回加 1 之后变量 n 的值。请看下面的代码：

```
int n = 2;
int valueProduced = 2*(++n);
cout << valueProduced << "\n";
cout << n << "\n";
```

上面的代码与之前的代码相比，唯一的区别在于 ++ 放在了变量 n 之前。从而输出发生了变化，如下所示：

```
6
3
```

需要注意的是，表达式  $n++$  和  $++n$  都使得变量  $n$  的值加 1，不同的地方在于表达式的值不同。如果  $++$  在变量之前，加 1 在值返回之前进行；如果  $++$  在变量之后，加 1 在值返回之后进行。

我们上面所讲的自增运算符的所有规则都适用于自减运算符，只是变量的值减 1 而不是加 1。请看下面的代码：

```
int n = 8;
int valueProduced = n--;
cout << valueProduced << "\n";
cout << n << "\n";
```

以上代码的输出为：

```
8
7
```

另一方面，代码：

```
int n = 8;
int valueProduced = --n;
cout << valueProduced << "\n";
cout << n << "\n";
```

的输出为：

```
7
7
```

$n--$  先返回变量  $n$  的值，然后减 1。 $--n$  则相反，变量  $n$  先减 1，再返回  $n$  的值。

自增和自减运算符只适用于单个变量，不能用于表达式。例如， $(x+y)++$ 、 $--(x+y)$ 、 $5++$  都为非法的表达式。在一个复杂的表达式中使用自增和自减运算符是危险的，下面对此有详细解释。



### 陷阱：求值顺序

对大多数运算符而言，子表达式的求值顺序是不固定的。尤其要注意的是，求值顺序不一定以从左至右的顺序进行，考虑如下表达式：

```
n + (++n)
```

假设变量  $n$  的初始值为 2，如果按照从左至右的顺序，该表达式的值为  $2+3$ ；如果按照从右至左的顺序，该表达式的值将为  $3+3$ 。由于 C++ 并没有规定子表达式的求值顺序，因此这个表达式的值可能为 5，也可能为 6。这意味着，我们在编写程序时，不能依赖于子表达式的求值顺序，但下面将要介绍的几个运算符除外。

有些运算符明确规定了子表达式的求值顺序是从左到右的。C++ 中，运算符  $\&\&$  (and)、 $\|\|$  (or) 和逗号运算符都是按照从左至右的顺序进行求值的，而这完全是符合常理的。考虑如下的代码：

```
(n <= 2) && (++n > 2)
```

假设变量  $n$  的初始值为 2，由于  $\&\&$  运算符保证了子表达式  $(n \leq 2)$  先求值，因此  $(n \leq 2)$  的值为 `true`， $(++n > 2)$  的值也为 `true`，从而整个表达式的值为 `true`。

这里，一定不要将运算符的优先级顺序与表达式的求值顺序混为一谈。例如：

```
(n + 2) * (++n) + 5
```

总是表示：

```
((n + 2) * (++n)) + 5
```

但是，子表达式 `++n` 和 `(n + 2)` 的计算顺序并不确定。不同的编译器有不同的顺序。

现在你应该明白为什么不建议在复杂的表达式中使用自增和自减运算符了。

如果还是搞不明白，那就简单地记住，编写程序时不要依赖于表达式的求值顺序。■

## 1.3 控制台输入/输出

进来是垃圾，出去也是垃圾。

一句俗语

简单的控制台输入/输出是通过定义在 `iostream` 库中的三个对象，`cin`、`cout` 和 `cerr` 来实现的。要使用控制台输入/输出，程序应该在相关源代码文件的最开始包含如下语句：

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

### 使用 `cout` 输出

**cout** 变量、字符串以及两者的任意组合可以通过 `cout` 输出到屏幕上。例如，考虑示例 1.1 中的部分代码：

```
cout << "Hello reader.\n"
      << "Welcome to C++.\n";
```

该语句输出两行字符串。`cout` 可以输出任意数量的数据项，每一项可以是字符串、变量或者更复杂的表达式，只需要在每一个待输出项的前面加上“<<”符号。

考虑另外一个例子：

```
cout << numberOfGames << "games played.";
```

该语句告诉电脑输出两项内容：变量 `numberOfGames` 的值和字符串 `"games played"`。

注意，不需要在每个待输出的项目前都加上 `cout`，可以在所有待输出的项目前添加 << 符号。前面单行的 `cout` 语句等价于下列两条 `cout` 语句：

```
cout << numberOfGames;
cout << " games played.";
```

**cout 语句  
中的表达式**

`cout` 语句中可以包含算数表达式，如下所示，其中 `price` 和 `tax` 为相关变量。

```
cout << "The total cost is $" << (price + tax);
```

有些编译器会要求表达式加上括号，如 `price+tax`；因此，为了保险起见，最好

所有的算术表达式都用括号括起来。

注意，上面的两个“<”符号之间一定不能有空格。这里 << 符号常常被称作插入运算符。另外，cout 输出语句以分号结束。

#### 输出字符串 中的空格

注意上面例子待输出字符串中的空格。cout 输出语句不会在输出项之前或者之后添加额外的空格，这就是为什么例子中的字符串常常以一个空格作为开头和结尾的原因。这些空格可以避免不同的变量和字符串连在一起。如果你仅仅是想输出空格，那么可以使用只包含空格的字符串，如下所示：

```
cout << firstNumber << " " << secondNumber;
```

另外，如果用 + 运算符将两个 string 类型的变量连接起来，那么结果是产生一个拼接起来的字符串。例如，如下代码：

```
string day1 = "Monday", day2="Tuesday";  
cout << day1 + day2;
```

产生拼接后的字符串：

```
"MondayTuesday"
```

### 换行符

在介绍转义序列的小节中已经提过，\n 告诉计算机在输出时另起一行。除非我们明确地告诉计算机换行，否则计算机会把所有的内容都放在同一行。取决于屏幕的设置，这可能产生任意分行以及整屏显示等各种各样的输出。注意，\n 要包含在括号中。C++ 中，换行符是一个特殊的字符，这个特殊的字符包含在引号中，并且 \ 和 n 之间没有空格。虽然换行符包含两个字符，但 C++ 将其视作单一的字符，称为换行符。

#### 换行符

如果希望在输出时插入一个空行，可以通过输出一个换行符来做到这一点，如：

```
cout << "\n";
```

另外一种输出空行的方式是使用 endl，endl 本质上和 \n 完全相同，因此你可以使用如下代码输出空行：

```
cout << endl;
```

虽然 \n 和 endl 含义相同，但用法不同。\n 必须放在引号中，endl 则不能放在引号中。

#### 决定使用 \n 还是 endl

这里给出一个决定到底是使用 \n 还是 endl 的方法：如果可以在一个长字符串的尾部添加 \n，那么就使用 \n，比如：

```
cout << "Fuel efficiency is "  
    << mpg << " miles per gallon\n";
```

否则，如果 \n 单独出现，那么就应该使用 endl，比如：

```
cout << "You entered " << number << endl;
```

## 输出换行

可以在输出的内容中添加 `\n` 来实现换行，例如：

```
cout << "You have definitely won\n"
      << "one of the following prizes:\n";
```

注意，换行符 `\n` 中，两个字符之间没有空格。

另外，可以通过输出 `endl` 来实现换行，上述 `cout` 语句也可以等价地写为：

```
cout << "You have definitely won" << endl
      << "one of the following prizes:" << endl;
```

提示：以 `\n` 或 `endl` 结束程序

在每个程序的结尾都添加一空行是一个好主意。如果最后输出的是一个字符串，那么在字符串的结尾添加 `\n`。否则就在程序的最后添加 `endl` 作为最后一个输出项。这样做有两个目的：某些编译器只有在程序结尾处包含换行符的情况下才会输出最后一行。在另一些编译器中，没有结尾的换行符程序可能运行没有问题，但紧接着的下一个程序的第一行输出会与上一程序的最后一行混在一起。就算上述两种情况都不会发生，在程序结束时添加空行也会使程序便于移植。■

## 格式化浮点数

浮点数的  
格式化输出

计算机输出 `double` 类型的浮点数时，输出格式或许不是我们想要的。例如以下简单的 `cout` 语句可能会产生多种输出结果：

```
cout << "The price is $" << price << endl;
```

如果 `price` 的值为 78.5，输出可能是：

```
The price is $78.500000
```

也有可能是：

```
The price is $78.5
```

还有可能是下面这种方式：

```
The price is $7.850000e01
```

以下这种输出方式最不可能出现，但却恰恰是最有意义的一种：

```
The price is $78.50
```

魔术语句

为了确保按特定的要求输出，程序必须指定数字输出的格式。

通过在程序中定义 `cout` 的属性，可以输出带有小数点的数字（如 `double` 类型的数字）以及指定小数点后小数的位数。如果希望保留小数点后两位小数，可以采用如下代码：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

### 输出余额

如果在程序中添加了上面的代码，那么其后所有的 `cout` 语句在输出浮点类型数据时，都会保留到小数点后两位。例如，假设下面的输出语句，`value` 的值为 78.5。

```
cout << "The price is $" << price << endl;
```

输出的结果将会是：

```
The price is $78.50
```

可以使用任何非负整数替换 2 以定义小数点后要输出的位数，甚至可以使用一个 `int` 类型的变量去替代 2。

`cout` 属性的详细介绍会在第 12 章。现在只需知道，通过以上的方式可以告诉 `cout` 如何去输出一个浮点数。

如果想改变输出结果中小数点后的位数，以使程序中不同的数字有不同的输出精度，可以用不同的数字代替前面表达式中的数字 2，重新定义 `cout` 的属性。`cout` 的属性更改之后，会影响其后所有的输出代码，直到相关属性再次被重新定义。

```
cout.precision(5);
```

### 输出 `double` 类型的值

如果在程序中添加如下代码，那么之后所有的 `cout` 语句在输出 `double` 类型的数值时都会带有两位小数。

```
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);
```

可以用任何非负整数代替 2，以便重新指定输出中的小数位数，甚至可以用 `int` 类型的变量来代替 2，定义输出中的小数位数。

### 用 `cerr` 输出

`cerr` 对象的使用和 `cout` 类似。只不过是，`cerr` 将待输出的内容发送到标准错误输出流中（一般是控制台屏幕）。由此可以区分两种不同类型的输出：`cout` 用于一般正常的输出，`cerr` 用于错误信息的输出。如果没有做任何特殊处理，`cout` 和 `cerr` 都会输出到控制台屏幕上，此时 `cout` 和 `cerr` 没有区别。

有些系统可以对程序的输出进行重定向。这种重定向方法由操作系统提供，非常有用。通过这种重定向，`cout` 和 `cerr` 的输出可以重定向到不同的文件中。

### 用 `cin` 输入

`cin` 进行数据输入和用 `cout` 进行数据输出多少有点相似。二者语法很接近，用 `cin` 代替 `cout` 且插入运算符的箭头方向取反。例如，示例 1.1 中变量 `numberOfLanguages` 由以下语句进行赋值：

```
cin >> numberOfLanguages;
```

可以在 `cin` 语句中使用多个变量，如下所示：

```
cout << "Enter the number of dragons\n"
    << "followed by the number of trolls.\n";
cin >> dragons >> trolls;
```

此外，根据个人习惯，以上输入代码还可以写作如下两行：

```
cin >> dragons
    >> trolls;
```

注意，与 `cout` 语句一样，一条输入语句最后只有一个分号。

`cin` 如何  
工作  
采用空白  
字符对数字  
进行分隔  
空白字符

程序运行到 `cin` 语句时，会等待键盘的输入。程序将第一个变量的值设为从键盘上输入的值，将第二个变量的值设为第二个输入的值，依此类推。但是只有在用户单击回车键后，程序才会读取用户的输入。这使得用户可以在单击回车键之前，采用退格键修改输入的内容。示例 1.5 展示了采用 `cin` 输入一个 `int` 和 `string` 的例子。

输入中的数字必须用空格或者换行隔开，这些分隔字符被称为空白字符。使用 `cin` 时，计算机跳会跳过所有的空白字符，直到遇到下一个输入内容。因此，输入内容之间是以一个空格、多个空格还是以换行隔开，都无关紧要。输入字符串时同样如此。这表明无法输入一个含有空格的字符串。如示例 1.5 所示，用户想输入“Mr. Bojangles”作为宠物的名字，但是程序只读取到 Mr.，因为下一个是一个空白字符，导致 Bojangles 被当前的 `cin` 输入忽略。但 Bojangles 会被下一个 `cin` 输入符读取。本书的第 9 章介绍了如何输入带有空格的字符串的方法。

通过 `cin` 可以读取整型、浮点型、字符类型以及字符串类型的值。本书后续的章节会介绍通过 `cin` 读取其他类型数据的方法。

### 示例 1.5 使用 `cin` 和 `cout` 处理字符串

---

```
1 // 使用 cin 和 cout 处理字符串的程序。
2 #include <iostream>
3 #include <string> ←————— 该程序用到了 string 类。
4
5 using namespace std;
6 int main( )
7 {
8     string dogName;
9     int actualAge;
10    int humanAge;
11
12    cout << "How many years old is your dog?" << endl;
13    cin >> actualAge;
14    humanAge = actualAge * 7;
15
16    cout << "What is your dog's name?" << endl;
17    cin >> dogName;
18
19    cout << dogName << "'s age is approximately " <<
20        "equivalent to a " << humanAge << " year old human."
21        << endl;
22
23    return 0;
24 }
```

## 示例运行结果1

```
How many years old is your dog?
5
What is your dog's name?
Rex
Rex's age is approximately equivalent to a 35 year old human.
```

## 示例运行结果2

```
How many years old is your dog?
10
What is your dog's name?
Mr. Bojangles
Mr.'s age is approximately equivalent to a 70 year old human.
```

Bojangles 没有读入到变量 dogName 中，因为 cin 以空格结束输入。

**cin 语句**

通过 cin 语句，可以将变量设为从键盘输入的值。

**语法**

```
cin >> Variable_1 >> Variable_2 >> ...;
```

**示例**

```
cin >> number >> size;
cin >> timeLeft
    >> pointsNeeded;
```

**百炼成钢习题**

9. 编写相应的输出语句，使得屏幕显示如下内容：

```
The answer to the question of
Life, the Universe, and Everything is 42.
```

10. 编写一条输入语句，从键盘读取一个 int 类型的值赋给变量 theNumber，在输入数据之前，给出提示信息，要求用户输入一个整数。
11. 普通形式下，要使得在输出一个 double 类型的数值时，保留到小数点后三位，需要在程序中添加什么样的语句？
12. 编写一个完整的 C++ 程序，该程序在屏幕上显示 Hello world。
13. 编写一条输出语句，输出字符 A 之后换行，然后输出字符 B、tab 字符和字符 C。
14. 下面的代码用来输入用户的名字、姓氏和年龄，但是存在一些问题，改正该程序。

```
string fullName;
int age;
cout << "Enter your first and last name." << endl;
cin >> fullName;
cout << "Enter your age." << endl;
cin >> age;
cout << "You are " << age << " years old, " << fullName << endl;
```



## 15. 以下代码会有什么输出？

```
string s1 = "5";
string s2 = "3";
string s3 = s1 + s2;
cout << s3 << endl;
```

## 提示：输入/输出中的换行

可以将输入和输出放在同一行，对用户来讲，这样做有时候会比较美观。为此，只需在提示行的末尾省略 `\n` 和 `endl` 换行符即可，这样用户的输入就会出现提示的同一行。例如，假设采用如下的提示和输入语句：

```
cout << "Enter the cost per person: $";
cin >> costPerPerson;
```

执行 `cout` 语句时，屏幕显示：

```
Enter the cost per person: $
```

当用户输入数字后，所输入的内容将显示在同一行，如：

```
Enter the cost per person: $1.25
```

## 1.4 编程风格

在非常重要的事情上，最要紧的并非真诚，而是格调。

奥斯卡·王尔德，《不可儿戏》

C++ 的编程风格与其他编程语言类似，都是为了便于程序的阅读和修改。下一章我们将会对程序编写中的缩进问题进行介绍。前面我们已经介绍了常量的使用。大多数情况下，程序中的字面值都应该定义为常量。变量名字的选择和适当的缩进可以减少程序中的注释，但仍有一些代码需要做专门的注释。

### 注释

有两种方法可以在 C++ 中添加注释。C++ 中，双斜杠标记着注释的开始，同一行中位于双斜杠之后的所有内容都是注释。编译器将简单地忽略 `/**` 之后的部分。如果需要多行注释，那么需要在各行的注释前面添加 `/**`，符号 `/**` 间没有空格。

另外一种添加注释的方法是使用组合 `/*` 和 `*/`。这两个组合之间的所有内容都是注释，都会被编译器忽略。与 `/**` 不同，这种注释方式可以跨越行的限制，如：

```
/*This is a comment that spans
three lines. Note that there is no comment
symbol of any kind on the second line.*/
```

`/* */` 类型的注释可以用在程序的任何地方。但是它们的使用不应影响程序的阅读和程序的整体设计。一般来讲，注释都放在行的末尾，或者单独成行。到底哪种

注释方式更好，目前尚没有定论。一种方法是在最终的代码中使用 `//` 进行注释，而在代码调试阶段则使用 `/* */` 进行注释。

何时加注释

一个程序应该包含多少注释，这个很难讲，一般而言适可而止。但对于初学者而言，很难把握其中的度，需要有一定的经验才能判断何时应该加入注释。一般来讲，任何重要但却不够明确或难以理解的地方都应添加注释。但是注释也不能太多，否则会适得其反。每一行都有注释的程序将被掩盖在注释之中，导致程序的结构完全被掩埋。例如，下面的注释就属多余：

```
distance = speed * time; //Computes the distance traveled.
```

## 1.5 库与命名空间

C++ 有许多标准库，这些标准库的定义分别放在不同的命名空间中。命名空间其实就是许多定义的集合。本书的后续章节将详细介绍编程库和命名空间的使用技巧。本节详细介绍 C++ 标准库的使用方法。

### 库与include命令

C++ 包含了一系列的标准库。这些标准库在程序的编写过程中不可或缺。事实上，要想编写一个不使用任何标准库的程序，几乎是不可能的。使用标准库的一般方法是采用 `include` 命名，一般形式如下：

#include

```
#include <Library_Name>
```

例如，控制台输入/输出相关的库为 `iostream`。因此，所有用到控制台输入/输出的程序都包含如下语句：

```
#include <iostream>
```

编译器对 `include` 命令中的空格要求非常严格，因此最安全的做法就是在使用 `include` 命令时，不添加多余的空格；`#` 之前和之后不要有空格，尖括号内也不应该有空格。

`include` 命令只是简单地用文件的内容替换当前的 `include` 命令。其中库的名字也仅仅是一个包含了所有定义项的文件名。本书的后续章节会介绍非标准库的引入，但是现在我们只需了解标准库的使用即可。附录 D 给出了常用的 C++ 标准库列表。

预处理器

在程序代码被送交编译器处理之前，会有一个预处理器对程序做一些简单的文本处理。事实上，`include` 命令是由预处理器而非编译器来处理的。大多数编译器在编译程序时，会自动地调用预处理器。

理论上讲，头文件给出的定义仅仅是库定义的一部分，但目前来讲，可以忽略该差别。因为只要使用 `include` 包含了库的头文件，那么 C++ 会自动添加库定义的剩余部分。

### 命名空间

命名空间

命名空间是定义的集合。定义（如函数定义）在不同的命名空间中可能会有相同

的名字。程序可以在不同的地方使用不同命名空间中的定义。本书稍后将对命名空间做详细介绍。现在我们只讨论命名空间 `std`。所有用到的标准库都定义在 `std` 命名空间中。为了能在程序中使用这些定义，需要在程序中插入如下的 `using` 命令：

```
using
namespace
    using namespace std;
```

因此一个典型的使用控制台输入/输出的程序将包含如下代码：

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

如果希望在程序中只引入命名空间的部分内容，`using` 命令提供了引入单个定义的方法。例如，假设程序只希望使用 `std` 命名空间中的 `cin`，可以使用如下的 `using` 命令：

```
using std::cin;
```

同样，如果程序只需使用命名空间中的 `cin`、`cout` 和 `endl`，那么需在程序中加入如下语句：

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

而不是：

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

旧式的 C++ 库的头文件没有将定义放在 `std` 命名空间中，因此在阅读以前的 C++ 代码时，你会发现头文件的名字有稍许不同，并且程序中也并没有包含任何的 `using` 命令。考虑到向后兼容性，这种用法是可以的。但仍然建议使用新的库头文件以 `std` 命名空间命令。



### 陷阱：库名的问题

C++ 语言仍处在不断地完善之中。如果你使用的编译器还没有符合最新的标准，那么可能得使用另外的库名。

如果以下的语句有问题：

```
#include <iostream>
```

那就试试

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

同样，其他的库名也存在同样的问题。附录 E 给出了库名的新旧对照。本书使用的全是新库名，如果你的编译器无法识别某个库名，那就试试相应的旧库名。总之，要么所有的新库名都可以使用，要么都不可用，不可能只有部分新库名可用的情况。

如果使用旧库名（以 `.h` 结尾的文件），就无须使用下面的 `using` 命令。

```
using namespace std; ■
```

## 本章小结

- C++ 区分大小写。例如 `count` 和 `COUNT` 是不同的标识符。
- 使用有意义的变量名。
- 变量在使用之前必须先进行声明。除此之外，变量声明可以在程序的任意位置进行。
- 确保变量在使用之前已经被初始化。变量初始化可以在声明变量的同时进行，也可通过赋值语句在首次使用变量之前进行。
- 可以将一个整数类型的值赋给浮点类型的变量，但是反之则不然。
- 程序中出现的所有数字常量都应该被赋予一个有意义的名字，这样就可以用名字常量来代替数字，增加程序的可读性及可修改性。名字常量可以借助于 `const` 修饰符来实现。
- 在算数表达式中使用括号，明确规定算数运算的顺序。
- 控制台输出采用 `cout` 对象。
- 字符串结尾的 `\n` 或者 `endl` 都可以在控制台输出一个新行。
- 对象 `cerr` 用来输出错误信息，一般情况下，`cerr` 和 `cout` 的作用相同。
- 控制台输入采用 `cin` 对象。
- 为了使用 `cin`、`cout` 或 `cerr`，应该在程序的开头包含以下语句：
 

```
#include <iostream>
using namespace std;
```
- C++ 有两种注释类型：行注释 `//` 以及多行注释 `/* */`。
- 切记不要注释过量。

## 自测练习题答案

1. 

```
int feet = 0, inches = 0;
int feet(0), inches(0);
```
2. 

```
int count = 0;
double distance = 1.5;
int count(0);
double distance(1.5);
public static void main(String[] args)
```
3. 程序的实际输出依赖于系统以及系统的使用历史，如下所示：
 

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

int main( )
```

```

{
    int first, second, third, fourth, fifth;
    cout << first << " " << second << " " << third
        << " " << fourth << " " << fifth << "\n";
    return 0;
}

```

4.  $3 * x$   
 $3 * x + y$   
 $(x + y) / 7$  注意  $x + y / 7$  是错误的。  
 $(3 * x + y) / (z + 2)$

5. bcbb

6.  $(1/3) * 3$  is equal to 0  
 因为 1 和 3 都是 int 类型, 此时运算符 / 执行的是整数除法, 因此 1/3 的值为 0, 而不是 0.3333..., 从而整个表达式的值也为 0。

7. #include <iostream>  
 using namespace std;  
 int main( )  
 {  
 int number1, number2;  
 cout << "Enter two whole numbers: ";  
 cin >> number1 >> number2;  
 cout << number1 << " divided by " << number2  
 << " equals " << (number1/number2) << "\n"  
 << "with a remainder of " << (number1%number2)  
 << "\n";  
 return 0;  
 }

8. a. 52.0  
 b. 9/5 的值为 1。由于分子与分母均为整数, 因此执行的是整数除法, 小数部分被忽略。而程序员期望的可能是浮点除法。  
 c.  $f = (9.0/5) * c + 32.0$ ; 或者  
 $f = 1.8 * c + 32.0$

9. cout << "The answer to the question of\n"  
 << "Life, the Universe, and Everything is 42.\n";

10. cout << "Enter a whole number and press Return: ";  
 cin >> theNumber;

11. cout.setf(ios::fixed);  
 cout.setf(ios::showpoint);  
 cout.precision(3);

12. #include <iostream>  
 using namespace std;  
  
 int main( )  
 {  
 cout << "Hello world\n";  
 return 0;  
 }

13. `cout << 'A' << endl << 'B' << '\t' << 'C';`

还有其他正确答案，如字母可以不加单引号，改加双引号。例如：

```
cout << "A\nB\tC";
```

14. `cin` 以空格作为字符串的边界，因此以空格分割的名字和姓是不能读入到一个变量中的。这里为了方便起见，将姓和名分别读入到不同的变量中：

```
string first, last;
int age;
cout << "Enter your first and last name." << endl;
cin >> first >> last;
cout << "Enter your age." << endl;
cin >> age;
cout << "You are " << age << " years old, " << first <<
    " " << last << endl;
```

15. 运算符“+”将两个字符串连接起来，因此结果是 `s3="53"`。如果 `s1` 和 `s2` 都是数值类型，那么将执行一般的算数操作。

## 编程练习

- 公制的一吨等于 35 273.92 盎司。编写一个程序，读入以盎司表示的一包早餐燕麦的重量，输出以吨表示的重量，以及一吨这样的早餐燕麦需要多少包？
- 某政府实验室研究发现，饮用苏打水中添加的人造甜味剂可导致实验老鼠死亡。你的一个朋友正在努力减肥，但是却嗜苏打水如命。他想知道可以饮用多少苏打水而不会致命。请编写程序来解决这一问题。程序的输入是致死一只老鼠所需的人造甜味剂的量、老鼠的重量以及节食者的体重。为了确保你朋友的安全，一定要注意程序要求的是节食者停止节食时的体重，而不是现在的体重。假定苏打水里有 1/10 为浓度是 1% 的人造甜味剂。采用名字常量来表示这一值。另外可以将百分比表示为 `double` 类型的值，如 0.001。
- 某公司的员工六个月来工资增加了 7.6%。编写一个程序，以员工之前的年收入为输入，输出员工工资增加的数额、新的年收入以及月收入。另外，注意采用名字常量来表示工资增长率 7.6%。
- 客户贷款的发放不总是那么直接。有一种形式的贷款叫做贴现分期贷款，具体实施方法如下：假设某笔贷款的面值为 1000 美元，利率为 15%，贷款期限为 18 个月。那么年利息为面值 1000 乘以 0.15，为 150 美元。年利息 150 再乘以贷款期限 1.5 年得到总的利息 225 美元。总利息将直接从贷款面值中减去，客户实际得到 775 美元。客户每月的偿还额则根据贷款的面值进行计算，也就是每月的偿还额为 1000 美元除以 18，得 55.56 美元。如果客户需要的贷款数目恰好为 775 美元，那么这种计算方法没有任何问题。但如果客户实际需贷款 1000 美元，那么这种计算方法就有问题。编写一个程序，该程序读取三个数字：客户希望的贷款额、利率以及贷款期限。程序应该计算出客户实际应该申请的贷款额度以及每月的偿还额。

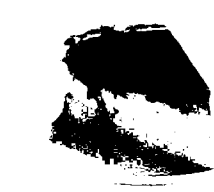
5. 编写一个程序，根据会议室空间的大小来判断是否违反有关的消防法规。程序的输入为会议室空间的大小以及参加会议的人数。如果参加会议的人数小于或等于会议室空间的大小，程序提示这次会议没有违反相关的消防法规，并给出还可以有多少人参加会议。如果人数超过了会议空间的大小，程序提示此次会议违反了相关消防法规，不能举办，同时给出应该减少的人数。
6. 某员工一周的工资收入为每小时 16.78 美元，加班工资是平时的 1.5 倍。员工的总收入中，6% 用来上缴社会保障税，14% 抽取作联邦所得税，5% 抽取作州所得税，另外每周还上缴 10 美元到工会。如果该员工全家人数大于或等于 3，还应支付 35 美元作为医疗保险费。编写一个程序，输入为员工一周工作的小时数和员工的全家人数，输出员工的总收入、应支付的款项以及一周的净收入。
7. 可以使用代谢当量 (MET) 来计算人们在运动中消耗的能量。下面是各种活动对应的 MET：
  - 跑步 (时速 6 英里)：10MET
  - 打篮球：8MET
  - 睡觉：1MET
 每分钟消耗的卡路里可以用以下方式计算：
 
$$\text{Calories/Minute} = 0.0175 \times 1 \text{ MET} \times (\text{Weight in kilograms})$$
 编写一个程序，输入个人的体重 (磅)、某个活动的 MET 数以及花在这个活动上的分钟数，然后计算消耗的卡路里总数。1 千克等于 2.2 磅。
8. 计算数  $n$  平方根的巴比伦算法如下：
  - (1) 设定一个大概值，并记为 `guess` (比如  $n/2$ )。
  - (2) 计算  $r = n / \text{guess}$ 。
  - (3)  $\text{guess} = (\text{guess} + r) / 2$ 。
  - (4) 回到第二步，执行步骤二、步骤三，并尽可能多地迭代这一过程。迭代的过程越多，`guess` 的值就越接近  $n$  的平方根。
 编写一个程序，输入一个 `double` 类型的数，迭代执行巴比伦算法 5 次，最终输出 `double` 类型的计算结果，并保留两位小数。
9. 当地游乐场的电子游戏机会根据你玩游戏的情况赠送一些优惠券。可以用 10 个优惠券换取一个糖果或者用三个优惠券换取一块口香糖。假设你更喜欢吃糖果，因此希望能换取尽可能多的糖果。编写一个程序，输入你获得的优惠券个数，假设你要用掉所有的优惠券，并且希望能得到尽可能多的糖果，请计算最终你可以换取的糖果和口香糖的个数。
10. 编写一个程序，用户输入一个秒数，该程序输出一个自由落体的物体在该时间段内下降的距离。假设没有摩擦和空气阻力，重力加速度为 32 英尺每秒。使用下面的公式：

$$\text{Distance} = \frac{1}{2} \times \text{acceleration} \times \text{time}^2$$

11. 编写一个程序，输入一个秒数，该程序将该秒数转换为对应的小时、分钟、秒。  
例如，用户输入 50391，该程序则输出 13 小时、59 分钟、51 秒。
12. 计算理想体重的一个简单办法是：身高最初的 5 英尺对应 110 英镑；高于 5 英尺的部分，每英寸身高对应体重 5 英镑。编写一个简单的程序，该程序包含两个变量，分别记录身高的英尺部分和英寸部分，程序的输入从键盘获取。另外，为了简单起见，假设所有的身高都大于 5 英尺，根据这些数据计算对应的理想体重。例如，假设一个人的身高为 6 英尺 3 英寸，那么程序中的两个变量分别为 6 和 3。







## 流程控制



### 2.1 布尔表达式 38

创建布尔表达式 38

陷阱：不等式连写 39

布尔表达式求值 40

优先级规则 41

陷阱：整数值用作布尔值 44

### 2.2 分支机制 45

if-else语句 45

复合语句 47

陷阱：用=代替== 47

省略else 49

嵌套语句 49

多分支if-else语句 49

switch语句 50

陷阱：遗漏switch语句中的break 52

提示：在菜单中使用switch语句 52

枚举类型 53

条件运算符 53

### 2.3 循环 54

while和do-while循环 54

再谈自增和自减运算符 57

逗号运算符 58

for语句 59

提示：重复N次的循环 61

陷阱：for语句中额外的分号 62

陷阱：无限循环 62

break与continue语句 65

嵌套循环 67

### 2.4 文件输入简介 67

通过ifstream读取文本文件内容 68

## 第2章 流程控制

“您能告诉我，从现在开始我该往哪个方向走吗？”

“这很大程度上取决于你想得到怎样的结果。”猫说。

刘易斯·卡罗尔，《爱丽丝漫游仙境》

### 概述

与大多数编程语言类似，C++ 通过分支语句和循环语句进行流程控制。C++ 中的分支语句和循环语句与其他编程语言类似。具体来讲，它们与 C 语言中的分支、循环语句完全一致，与 Java 语言的分支、循环语句非常类似。异常处理也是流程控制的一种，这部分内容将在第 18 章提到。

### 2.1 布尔表达式

要想区分真伪，首先必须知道什么是真，什么是伪。

贝尼迪·斯宾诺沙，《伦理学》

布尔表达式

大多数分支语句是通过布尔表达式进行控制的。布尔表达式是值为真或者假的任意表达式。最简单的布尔表达式由两个数字或者变量组成，并且由示例 2.1 中所给出的比较运算符进行连接。需要注意的是，其中一些比较运算符由两个符号构成，例如“==”、“!=”、“<=”、“>=”表示相等，而“!=”表示不等。另外，组成这些比较运算符的两个符号之间不能有空格。

#### 创建布尔表达式

“&&”代表  
“与”运算

可以将两个布尔表达式用“与”（and）运算符进行连接。C++ 中“与”运算符表示为 &&。例如，当 x 的值大于 2 且小于 7 的情况下，如下表达式值为真：

```
(2 < x) && (x < 7)
```

当两个布尔表达式以 && 连接时，如果两个布尔表达式都为真，则整个表达式的值为真，否则为假。

“||”代表  
“或”运算

除了“与”运算符外，还可以用“或”运算符连接两个布尔表达式。C++ 中，或运算符记为 ||。例如，如果变量 y 小于 0 或者大于 12，下面的表达式值为真：

```
(y < 0) || (y > 12)
```

当两个布尔表达式通过或运算符（||）连接时，如果任一布尔表达式的值为真，那么整个表达式的值就为真，否则为假。

可以使用取反运算符“!”对布尔表达式的结果进行取反，只需将取反运算符“!”

放在该布尔表达式的前面即可。例如,  $!(x < y)$  表示  $x$  小于或等于  $y$  时, 该表达式值为假, 否则为真。 $!(x < y)$  等价于  $(x \geq y)$ 。在某些情况下, 表达式中的括号也可以省略而不影响表达式的含义, 本章的“优先级规则”小节中对此有相关介绍。

### “与”运算符 (&&)

通过“与”运算符 (&&), 可以将两个简单的布尔表达式组合成一个复杂的表达式。

#### 语法

```
(Boolean_Exp_1) && (Boolean_Exp_2)
```

#### 示例

```
if ( (score > 0) && (score < 10) )
    cout << "score is between 0 and 10.\n";
else
    cout << "score is not between 0 and 10. \n";
```

如果变量 `score` 的值大于 0 且小于 10, 那么程序将执行第一个 `cout` 语句, 否则将执行第二条 `cout` 语句 (`if-else` 语句的含义显而易见, 本章的后续小节也会对其做详细介绍)。

### 陷阱：不等式连写

切记, 不要使用这样的连写方式,  $x < z < y$ 。如果这样做, 程序虽然可以进行成功编译且运行, 但往往会得出错误的结果。相反, 应该使用运算符“&&”对其连接, 如下所示:

```
(x < z) && (z < y) ■
```

### 示例 2.1 比较运算符

数学符号	名字	C++ 记法	C++ 示例	数学等式
=	等于	==	<code>x+7 == 2*y</code>	$x+7 = 2y$
≠	不等于	!=	<code>ans != 'n'</code>	$ans \neq 'n'$
<	小于	<	<code>count &lt; m+3</code>	$count < m+3$
≤	小于或等于	<=	<code>time &lt;= limit</code>	$time \leq limit$
>	大于	>	<code>time &gt; limit</code>	$time > limit$
≥	大于或等于	>=	<code>age &gt;= 21</code>	$age \geq 21$

**“或”运算符 (||)**

通过“或”运算符 (||)，可以将两个简单的布尔表达式组合成一个复杂的表达式。

**语法**

```
(Boolean_Exp_1) || (Boolean_Exp_2)
```

**示例**

```
if ( (x == 1) || (x == y) )
    cout << "x is 1 or x equals y.\n";
else
    cout << "x is neither 1 nor equals y.\n";
```

如果变量  $x$  的值等于 1，或者  $x$  的值等于  $y$  的值，则执行第一个 `cout` 语句，否则执行第二个 `cout` 语句 (if-else 语句的含义显而易见，本章的后续小节也会对其做详细介绍)。

**布尔表达式求值**

本章后面两节将介绍如何用布尔表达式来控制分支语句和循环语句。一个布尔表达式可以独立于分支语句和循环语句，进行单独求值。`bool` 类型变量的值或者为真或者为假，因此可以用来存储布尔表达式的值。例如：

```
bool result = (x < z) && (z < y);
```

布尔表达式的求值与一般算数表达式的求值相同。唯一的不同是，算数表达式采用 +、-、\*、/ 等运算符，得到的结果是一个数字。而布尔表达式采用关系运算符 (如 == 和 <) 以及布尔运算符 (如 || 和 &&)，得到的最终结果是“真”或者“假”。注意，“==”、“!=”等运算符作用于任何内建数据类型，产生一个布尔类型的值 (真或者假)。

首先，我们复习一下算数表达式的求值过程 (布尔表达式的求值过程类似)。考虑如下的算数表达式：

```
(x + 1) * (x + 3)
```

假设变量  $x$  的值为 2，首先计算加法，分别得到 3 和 5，然后对这两个值做乘法操作，得到该表达式的最终值 15。在对该表达式进行求值时，首先对表达式  $(x+1)$  和  $(x+3)$  分别求值，然后再做乘法操作。

**真值表**

布尔表达式的求值过程完全相同。首先对子表达式进行求值，结果为真或者假，在此基础上再根据示例 2.2 表中的规则进行求值。考虑如下的布尔表达式：

```
!( (y < 3) || (y > 7) )
```

这有可能是某个 if-else 语句的控制表达式。假设  $y$  的值为 8，此时，子表达式  $(y < 3)$  和  $(y > 7)$  的值分别为假和真，因此整个表达式的值等价于：

```
!(false || true)
```

根据 OR 运算规则，计算机发现括号内表达式的值为 true，因此整个表达式等价于：

!(true)

根据!运算规则, 最终得出!(true) 的值为 false, 至此求得最初布尔表达式的值为 false。

## 示例 2.2 真值表

### AND

表达式 1	表达式 2	表达式 1 & 表达式 2
true	true	true
true	false	false
false	true	false
false	false	false

### NOT

表达式	!(表达式)
true	false
false	true

### OR

表达式 1	表达式 2	表达式 1    表达式 2
true	true	true
true	false	true
false	true	true
false	false	false

## 布尔值 true 和 false

true 和 false 是预定义的 bool 常量 (注意是小写)。C++ 中, 如果某个布尔表达式的值为真则取值 true, 如果值为假则记为 false。

## 优先级规则

**括号** 布尔表达式 (以及算数表达式) 在某些情况下可以省略不必要的括号。如果省略括号, 计算的优先级顺序如下: 先进行“!”运算, 接着执行关系操作运算, 如 <, 然后是 &&。最后是 ||。不过强烈建议给表达式加上适当的括号, 以便于其他人理解。在单纯的 && 或 || 连接的表达式中, 完全可以省略掉括号。例如下面的布尔表达式就是一个比较好的例子 (无论是从可读性还是从编译器的可移植性来考虑都是如此)。

```
(temperature > 90) && (humidity > 0.90) && (poolGate == OPEN)
```

优先级顺序

较高的  
优先级

短路求值  
完全求值

由于关系运算符 `>>` 和 `==` 具有更高的优先级，因此可以省略该表达式中的括号，不过加上括号会提高程序的可读性。

表达式省略括号时，编译器会根据**优先级顺序**对该表达式进行理解。C++ 的运算符优先级顺序如示例 2.3 所示。示例 2.3 中的一些运算符会在后续的章节中一一介绍。

如果某种操作在其他操作之前执行，那么这个操作就拥有**较高的优先级**。示例 2.3 中处于同一个方格内的所有运算符拥有相同的优先级；不同方格内越靠前的运算符，其优先级越高。

如果运算符的优先级相同且没有括号的影响，一元运算符和赋值运算符从右到左进行结合。例如，`x = y = z` 意味着 `x = (y = z)`。对于二元运算符，如果优先级顺序相同，则是从左到右进行结合的。例如 `x + y + z` 意味着 `(x + y) + z`。

注意，优先级规则不仅包括算数运算符，如 `+` 和 `*`，同时也包括布尔运算符，如 `&&` 和 `||`。对于同时包含了算数运算符和布尔操作的表达式，如下：

```
(x + 1) > 2 || (x + 1) < -3
```

根据示例 2.3 的规则，上式其实等同于：

```
((x + 1) > 2) || ((x + 1) < -3)
```

因为 `>` 和 `<` 的优先级高于 `||`。事实上，可以去掉上式中的所有括号而该式的运算顺序不受任何影响，只是不便于阅读。虽然不提倡省略所有的括号，但我们可以通过该表达式理解运算符的优先级顺序。对于下面这个省略了括号的表达式：

```
x + 1 > 2 || x + 1 < -3
```

根据规则，首先进行一元运算符运算，其次进行加法运算，然后是 `>` 和 `<`，最后是 `||`。这与括号省略之前的运算顺序完全一样。

前面我们介绍了布尔运算的求值顺序。但是在 C++ 的具体实现中，计算机对布尔表达式求值，通常会走一些捷径。在许多情况下，对于一个布尔表达式来说，实际上只需要计算两个表达式中的第一个。例如，考虑下面的表达式：

```
(x >= 0) && (y > 1)
```

如果 `x` 小于 0，此时 `(x >= 0)` 为 `false`，从示例 2.2 我们知道，`&&` 表达式中，有一个子表达式为假，那么不管其他子表达式是真是假，该 `&&` 表达式的最终结果都为假。因此，如果我们已经求得第一个子表达式的值为假，那么就不需要对第二个子表达式进行求值。对于 `||` 表达式也存在类似的情况，如果第一个子表达式的值为 `true`，那么就可以判定整个表达式的值为真了，而不用计算其他子表达式的值。C++ 运用了这一点，可以在某些情况下避免计算 `&&` 和 `||` 表达式中的第二个子表达式，降低计算开销。C++ 首先会计算 `&&` 和 `||` 表达式中左子表达式的值，如果根据该子表达式可以确定最终的值，那么 C++ 会直接返回最终的值，不对右子表达式进行求值。这种运算方法称为**短路求值**。

有些语言与 C++ 不同，它们采用的是**完全求值**。采用这种方法时，`&&` 表达式和 `||` 表达式中所有的子表达式都会被计算，之后再根据真值表得到最终结果。

## 示例 2.3 运算符的优先级

::	作用域运算符
.	点运算符
->	成员选择符
[]	数组索引
()	函数调用
++	后缀自增运算符 (放在变量后面)
--	后缀自减运算符 (放在变量后面)
++	前缀自增运算符 (放在变量前面)
--	前缀自减运算符 (放在变量前面)
!	取反运算符
-	负号
+	正号
*	反引用
&	取地址符
new	创建 (分配) 内存
delete	删除
delete []	删除数组
sizeof	获取对象大小
()	类型转换
*	乘
/	除
%	取余
+	加法
-	减法
<<	插入运算符 (控制台输入)
>>	输出运算符 (控制台输出)
<	小于运算符
>	大于运算符
<=	小于或等于运算符
>=	大于或等于运算符
==	相等
!=	不等
&&	与
	或
=	赋值
+=	相加然后赋值
-=	相减然后赋值
*=	相乘然后赋值
/=	相除然后赋值
%=	取余然后赋值
? :	条件运算符
throw	抛出异常
,	逗号运算符

优先级最高

优先级最低



## 陷阱：整数值用作布尔值

整型转化为  
布尔类型

C++ 有时会将整数值当作布尔值使用，或者相反。具体来讲，将整数值当作布尔值使用时，整数 1 和 0 分别被转化为 true 和 false；当将布尔值当整数值使用时，布尔值 true 和 false 分别被转化为 1 和 0。实际的编译器可能会采取更激进的方法，任何非零的数都按 true 处理，而 0 按假处理。只要按规则正确地编写布尔表达式，这种转化不会带来任何问题。但是，了解这一点对正确调试程序有很大帮助。

例如，假设我们希望获得一个布尔表达式，当时间还没有到某个点时，其值为真。可以使用如下的语句：

```
!time > limit
```

乍眼一看，该表达式没有任何问题。但实际上，该表达式是错误的，而且编译器不会对该错误做出提示。相反，编译器会根据示例 2.3 的优先级规则，将其解释为：

```
(!time) > limit
```

很明显，该表达式是错误的。如果 time 的值为 36，那么 (!time) 的值将等于“非 36”。C++ 中，任何非零整数都转化为真，而 0 将转化为假。因此，!36 将转化为“非真”，也就是假。此后，布尔值假又会转化为 0，以便与整数 limit 进行比较。

由此可以看到，C++ 对该表达式的计算与我们的期望不符。如果 time 的值为 36，limit 的值为 60，我们希望上述表达式值为真，但实际上，C++ 给出的值却为假：(!time) 的值为假，假转化为 0，整个布尔表达式转化为：

```
0 > limit
```

由于 limit 的值为 60，显而易见，整个表达式计算结果为假。

有两种办法可以纠正此问题，一是正确地使用！运算符。使用！运算符时，一定将参数用括号括起来。因此以上布尔表达式的正确写法如下：

```
!(time > limit)
```

另一种避免此种错误的方法是不使用！运算符。例如，之前的布尔表达式可以表示为：

```
if (time <= limit)
```

我们总有办法避免使用！运算符，以至于有些程序员提倡尽量不要使用！运算符。■

对于布尔表达式，短路求值和完全求值都可以给出正确的最终结果，因此大多数情况下，我们无须关心 C++ 采用了哪种求值方式。但是有一种特殊情况，如果第二个子表达式未定义，此时 C++ 采用的短路求值方式会对程序的执行产生影响。具体来讲，它会隐藏程序中存在的潜在问题。考虑如下的语句：

```
if ( (kids != 0) && ((pieces/kids) >= 2) )
    cout << "Each child may have two pieces!";
```

如果变量 kids 的值不为 0，以上语句不存在任何问题。但如果变量 kids 的值为 0，我们来看看短路求值对程序的执行会产生什么影响。此时，(kids!=0) 的值为 false，采用短路求值方式，计算机将直接给出整个表达式的值为假，跳过第二个子表达式的求值。从而忽略了第二个子表达式中除数为 0 的运行时错误。

1. 假设变量 count 的值为 0，变量 limit 的值为 10，对以下布尔表达式求值。

- a. `(count == 0) && (limit < 20)`
- b. `count == 0 && limit < 20`
- c. `(limit > 20) || (count < 5)`
- d. `!(count == 12)`
- e. `(count == 1) && (x < y)`
- f. `(count < 10) || (x < y)`
- g. `!((count < 10) || (x < y)) && (count >= 0)`
- h. `((limit / count) > 7) || (limit < 20)`
- i. `(limit < 20) || ((limit / count) > 7)`
- j. `((limit / count) > 7) && (limit < 0)`
- k. `(limit < 0) && ((limit / count) > 7)`
- l. `(5 && 7) + (!6)`

2. 有时会碰到如下表示的表达式：

$$2 < x < 3$$

C++ 中，该表达式的值与预期的不同。解释其中的原因，并给出正确的布尔表达式。

3. 考虑如下的二次多项式：

$$x^2 - x - 2$$

请描述，什么情况下该二次多项式的值为正，给出该值与大根（-1）和小根（2）的关系。写出一个布尔表达式，当该表达式为真时，该二次多项式的值大于 0。

4. 考虑如下的二次多项式：

$$x^2 - 4x + 3$$

请描述，什么情况下该二次多项式的值为负。写出一个布尔表达式，当该表达式的值为真时，该二次多项式的值小于 0。

## 2.2 分支机制

如果你碰到一个好的机遇，就请牢牢抓住。

尤吉·贝拉

### if-else 语句

**if-else**  
语句

**if-else** 语句根据布尔表达式的值，在两个语句中进行选择性执行。例如，假定我们想编写一个程序来计算小时工的周薪。假定一周内，工作时间超出 40 小时之后，每小时的工资是之前的 1.5 倍。因此，小时工的收入可以表示为：

$$\text{rate} * 40 + 1.5 * \text{rate} * (\text{hours} - 40)$$

但是，如果该小时工的工作时间少于 40 小时，其收入为：

```
rate * hours
```

采用如下的 if-else 语句，可以表示出该小时工的收入：

```
if (hours > 40)
    grossPay = rate * 40 + 1.5 * rate * (hours - 40);
else
    grossPay = rate * hours;
```

下文的方框中给出了 if-else 语句的语法。如果括号内的布尔表达式为真，那么执行 else 之前的语句；如果布尔表达式的值为假，则执行 else 之后的语句。

### if-else 语句

if-else 语句根据布尔表达式的值，在两个语句片段之间选择执行，相关语法如下。注意，其中的布尔表达式一定要放在括号中。

**语法：每个语句片段都分别包含一条语句**

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

如果 Boolean\_Expression 的值为 true，执行语句 Yes\_Statement。如果 Boolean\_Expression 的值为 false，则执行 No\_Statement。

**语法：每个语句片段包含多条语句。**

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
    ...
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
    ...
    No_Statement_Last
}
```

### 示例

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

注意，if-else 语句会包含一些小的语句，这也是 C++ 中语句组织的基本方式：采用一定的方式将小语句组织起来，从而构成一个更大的语句。

括号

记住，if-else 语句中的布尔表达式必须放在括号当中。

## 复合语句

包含复合  
语句的 if-  
else 语句  
复合语句

if-else 语句的某个分支可以包含多条语句。为此，将某个分支下的语句用大括号括起来。包含在大括号中的语句序列被称为**复合语句**。C++ 对复合语句的处理方法与单语句的处理方法完全相同。任何可以使用单语句的地方，同样也可以使用复合语句（因此第二个名为“if-else 语句”的语法模板其实是第一种的特殊形式）。

下面是 if-else 语句中两种常见的缩进书写方式：

```
if (myScore > yourScore)
{
    cout <<"I win!\n";
    wager = wager + 100;
}
else
{
    cout <<"I wish these were golf scores.\n";
    wager = 0;
}
```

和

```
if (myScore > yourScore){
    cout <<"I win!\n";
    wager = wager + 100;
} else {
    cout <<"I wish these were golf scores.\n";
    wager = 0;
}
```

两者唯一的不同就是大括号的位置。由于第一种方式代码的可读性更好，故很多人都采用这种方式。第二种方式节省空间，因此受到某些程序员的青睐。



### 陷阱：用 = 代替 ==

C++ 存在一些语句，乍一看存在问题，本以为系统会给出错误信息，结果系统并没有报错。这就意味着如果你写错了某些程序语句，系统不但不会报错，反而正确编译并运行这些语句，从而导致错误的运行结果。由于这些错误很难发现，因此往往在不知不觉中就会导致严重的问题。请看下面的例子：

```
if (x = 12)
    Do_Something
else
    Do_Something_Else
```

假设你想测试  $x$  的值是否等于 12，因此实际上应该使用 `==` 而不是 `=`。或许你认为编译器会发现这一错误，并给出提示，原因是表达式 `x = 12` 是赋值语句，不是布尔表达式。但是遗憾的是，情况并非如此。C++ 中，`x = 12` 是一个有返回值的表达

式,如同  $x + 12$  或  $2 + 3$  一样。赋值表达式的返回值就是赋给等式左边变量的值。例如,  $x=12$  的返回值就是 12。之前我们已经介绍过, C++ 会将非零整数转化为布尔值 true。如果将赋值表达式  $x = 12$  放在 if-else 语句的布尔表达式部分, 该布尔表达式的值将永为真。

由于该错误隐藏得很深, 而且编译器没有错误提示, 因此很难发现。如果把 12 放在比较运算符的左边, 如  $12 == x$ , 那么无须任何特殊的指令, 编译器就能发现这个错误, 并给出提示。因此  $12 == x$  是合法的, 但  $12 = x$  却是非法的。■

### 自测练习

5. 下列语句会出现除数为零的情况吗?

```
j = -1;
if ((j > 0) && (1/(j + 1) > 10))
    cout << i << endl;
```

6. 编写一个 if-else 语句, 如果整型变量 score 的值大于 100, 输出 “HIGH”; 如果小于或等于 100 时, 输出 “LOW”。

7. 假设 savings 和 expenses 都是 double 类型的变量, 并已赋初值。编写一个 if-else 语句, 如果 savings 大于或等于 expenses, 则输出 “Solvent”, 并从 savings 减去 expenses, 将 expenses 的值设为 0; 如果 savings 小于 expenses, 则仅输出 “Bankrupt”, 不改变任何变量的值。

8. 编写一个 if-else 语句, 如果变量 exam 的值大于或等于 60 且变量 programsDone 的值大于或等于 10, 输出 “Passed”; 否则输出 “Failed”。其中变量 programDone 和 exam 都为 int 类型。

9. 编写一个 if-else 语句, 如果变量 temperature 的值大于或等于 100, 或变量 pressure 的值大于或等于 200, 输出 “Warning”, 否则输出 “OK”, 其中变量 temperature 和 pressure 均为 int 类型。

10. 给出下列语句的输出结果, 并给出解释。

```
a. if(0)
    cout << "0 is true";
    else
    cout << "0 is false";
    cout << endl;
```

```
b. if(1)
    cout << "1 is true";
    else
    cout << "1 is false";
    cout << endl;
```

```
c. if(-1)
    cout << "-1 is true";
    else
    cout << "-1 is false";
    cout << endl;
```

注意: 本题目仅仅作为练习, 不代表实际编程时应该遵从的编程风格。

## 省略else

有些时候我们需要一个更简单的 if-else 语句：根据判断条件来决定是否执行某些特定语句。C++ 中，只需简单地省略掉 else 部分即可。为了与 if-else 语句区分开，这类语句被称为 **if 语句**。下面是 if 语句的一个例子：

```
if (sales >= minimum)
    salary = salary + bonus;
cout << "salary = $" << salary;
```

如果变量 sales 的值大于或等于 minimum，则执行赋值语句，然后再执行 cout 输出语句。否则，跳过赋值语句，直接执行 cout 输出语句。

## 嵌套语句

if-else 语句和 if 语句的每一分支都可以包含多条子语句。到目前为止，我们知道复合语句和单语句可以用在每一分支中。其实，任何合法的 C++ 语句都可以用在 if-else 语句的分支中。

**对齐** 使用嵌套语句时，一般需要对每一层的嵌套语句进行对齐书写，但也有例外的情況（如多分支 if-else 语句）。

## 多分支if-else语句

**多分支 if-else** 多分支 if-else 语句不是一种新的 C++ 语句类型，它只是在一个普通的 if-else 语句中又嵌入了其他 if-else 语句。但它比较特殊，不是一般的嵌套语句，所以专门拿出来介绍。

下面给出了多分支 if-else 语句的使用方法以及简单的实例。注意，其中的布尔表达式是排成一线的，它们的执行也是依次执行的，因此很容易看出布尔表达式与其行为之间的对应关系。所有布尔表达式依次进行求值，直到碰到一个值为 true 的布尔表达式。接着执行与该布尔表达式对应的行为代码。最后的 else 可有可无。如果最后有 else 语句，且前面所有的布尔表达式都为假，那么执行与 else 对应的行为代码。如果前面所有布尔表达式均为假，且最后没有 else，则不执行任何行为。

### 多分支 if-else 语句

#### 语法

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
    .
    .
    .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

**示例**

```

if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10)
    cout << "Stay home, but call work.";
else if (temperature <= 0)
    cout << "Dress warm.";
else
    cout << "Work hard and play hard.";

```

程序运行时，依次检查布尔表达式，直到某个布尔表达式为真，然后执行对应的行为代码。如果所有的布尔表达式都为假，那么执行 *Statement\_For\_All\_Other\_Possibilities* 语句。

**switch语句**

**switch**  
语句

**switch** 语句是 C++ 中另外一种实现多分支跳转的语句。下面介绍 **switch** 语句的语法以及简单的例子。

控制表达式

执行 **switch** 语句时，可以选择某一支，运行其中的语句。而究竟运行哪一条分支，取决于关键字 **switch** 之后括号中的控制表达式。**switch** 语句的控制表达式必须返回一个 **bool** 值、一个 **enum** 常量、一个整数类型或者一个字符。执行 **switch** 语句时，首先计算控制表达式。计算机将控制表达式的值与其后所有 **case** 标识符之后的常量做对比。如果发现某个常量恰好等于该控制表达式的值，则程序跳转到对应的 **case** 分支，执行相关代码。两个不同的 **case** 分支之后，不能出现同一个常量，否则会产生难以理解的程序。

**break** 语句

程序执行过程中，碰到 **break** 语句或者 **switch** 语句的结尾时，标志该 **switch** 语句执行结束。**break** 语句由关键字 **break** 后跟一个分号组成。当计算机执行某个 **case** 标记之后的语句时，如果遇到 **break** 语句便会停下来，因此常使用 **break** 来终止 **switch** 语句。如果省略了 **case** 代码中的 **break** 语句，那么计算机在执行完一个 **case** 标记中的代码后，会继续执行下一个 **case** 标记后的代码。

注意，同一段代码可以有多个 **case** 标记，如下所示：

```

case 'A':
case 'a':
    cout << "Excellent."
        << "You need not take the final.\n";
    break;

```

由于第一个 **case** 标记中没有 **break** 语句（事实上是没有任何语句），因此效果上就如同一个 **case** 有两个标记，不过 C++ 语法要求每个标记（如 'A' 和 'a'）前都必须有一个关键字 **case**。

**default**

如果没有一个 **case** 标记满足控制表达式的值，那么就执行 **default** 标记之后的语句。**default** 标记不是必需的，如果没有 **default** 标记且所有的 **case** 标记都不满足，那么这个 **switch** 语句就不执行任何操作。一般情况下，最好包含一个

default 标记。如果你认为 case 标记已经列出了所有的可能，那么可以在 default 标记中放入一个错误提示。

## switch 语句

### 语法

```
switch (Controlling_Expression)
{
    case Constant_1:           每个 case 语句后都有一个 break 语句。
        Statement_Sequence_1  如果省略该 break 语句的话，程序会执
        break;                行下一个 case 语句，直到碰到 break
                                语句或者 switch 语句结束。
    case Constant_2:
        Statement_Sequence_2
        break;
        .
        .
        .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

### 示例

```
int vehicleClass;
double toll;
cout << "Enter vehicle class:";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break; ← 如果省略该 break 语句，那么旅客将支
                  付 1.5 美元。
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```



## 自测练习题

11. 下面代码的输出是什么?

```
int x = 2;
cout <<"Start\n";
if (x <= 3)
    if (x != 0)
        cout <<"Hello from the second if.\n";
    else
        cout <<"Hello from the else.\n";
cout <<"End\n";

cout <<"Start again\n";
if (x > 3)
    if (x != 0)
        cout <<"Hello from the second if.\n";
    else
        cout <<"Hello from the else.\n";
cout <<"End again\n";
```

12. 下面代码的输出是什么?

```
int extra = 2;
if (extra < 0)
    cout <<"small";
else if (extra == 0)
    cout <<"medium";
else
    cout <<"large";
```

13. 如果自测练习题 12 中的赋值语句改为下面这句, 程序的输出是什么?

```
int extra = -37;
```

14. 如果自测练习题 12 中的赋值语句改为如下这句, 程序的输出又是什么?

```
int extra = 0;
```

15. 编写一个多分支 if-else 语句, 将 int 变量 n 的值归入下列某一类中并给出恰当的信息。

```
n < 0 or 0 <= n <= 100 or n > 100
```

## 陷阱: 遗漏 switch 语句中的 break

如果遗漏了 switch 语句中的 break, 编译器不会做任何提示——你编写的仍然是一个语法正确的程序, 但程序却不会得到预期的结果。请留意前面“switch 语句”方框中所举例子的注释部分。■

## 提示: 在菜单中使用 switch 语句

多分支 if-else 语句与 switch 语句相比, 其用途更为广泛。任何可以使用 switch 语句的地方同样可以使用多分支 if-else 语句, 但有时用 switch 语句会更

清晰。例如，switch 语句是实现菜单的最佳方法，其中每个 case 语句都可以作为菜单的一个选项。■

## 枚举类型

### 枚举类型

枚举类型的值是由一系列 int 类型的常量定义的，它很像是一系列命名常量。switch 语句中的一系列 case 标识符一般就用枚举类型来表示。

定义枚举类型时，可以使用任意 int 值，且常量的个数不受任何限制。例如，下面的语句为每月的天数定义了一个枚举类型。

```
enum MonthLength { JAN_LENGTH = 31, FEB_LENGTH = 28,
    MAR_LENGTH = 31, APR_LENGTH = 30, MAY_LENGTH = 31,
    JUN_LENGTH = 30, JUL_LENGTH = 31, AUG_LENGTH = 31,
    SEP_LENGTH = 30, OCT_LENGTH = 31, NOV_LENGTH = 30,
    DEC_LENGTH = 31 };
```

正如本例所示，同一枚举类型中的两个或多个命名常量可以拥有同样的值。

如果没有明确地指定数值，枚举类型定义中的标识符默认从 0 开始赋值。例如，下面的类型定义

```
enum Direction { NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3 };
```

等价于：

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

这种没有给出 int 值的形式经常用在只需要一组名字，而不关心其具体值的情况中。假设用一个值初始化了一个枚举常量，如：

```
enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE };
```

那么，ONE 的值为 17；TWO 的值为下一个值，即 18；THREE 的值则为 19；FOUR 的值则为 -3；从而导致 FIVE 的值为 -2。简单来说，在没有具体赋值的情况下，枚举常量的值为之前常量的值加 1。

虽然枚举常量的值是以 int 值给出的，并且在许多场合下都可以当作整数来使用，但切记枚举类型是一种独立的类型，应该与 int 类型区分开。可以将枚举类型当作标记，但要避免用枚举类型的变量进行算术运算。

## 条件运算符

### 条件运算符

可以在一个表达式中嵌入一个条件，这是通过三元运算符进行的，即所谓的**条件运算符**（也称为三元运算符或者算数 if 语句）。三元运算符的用法类似以前的程序设计风格，我们并不推荐这种用法。考虑到本书内容的完整性，我们这里对其做相关的介绍。

条件运算符是另外一种形式的 if-else 语句，请看下面的例子：

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

采用条件运算符，该语句可以改写为：

条件运算符  
表达式

```
max = (n1 > n2) ? n1 : n2;
```

该赋值语句等号右边的表达式就是一个条件运算符表达式：

```
(n1 > n2) ? n1 : n2
```

“?”和“:”共同构成了这个条件运算符的三元运算符。条件运算符表达式以一个带有“?”的布尔表达式开始，然后带有两个用冒号隔开的表达式。如果布尔表达式的值为真，该条件运算符表达式的值为第一个表达式的值，否则为第二个表达式的值。

16. 给出下面的声明和输出语句，这些语句的输出结果是什么？

```
enum Direction { N, S, E, W };
// ...
cout << W << " " << E << " " << S << " " << N << endl;
```

17. 给出如下的声明和输出语句，对应的输出结果是什么？

```
enum Direction { N = 5, S = 7, E = 1, W };
// ...
cout << W << " " << E << " " << S << " " << N << endl;
```

## 2.3 循环

有些任务与其说是家务劳动，不如说是科林斯王的折磨，其无休止地重复：一遍又一遍，一天又一天。

西蒙娜·德·波伏娃

循环体  
迭代

C++ 中的循环和其他高级编程语言类似，共有三种循环语句：while 语句、do-while 语句和 for 语句。C++ 中的相关术语和其他编程语言相同。一个循环中反复执行的代码块称为循环体，其中循环的每次执行称为循环的一次迭代。

### while和do-while循环

while 和  
do-while

下面给出 while 循环与它的变体，以及 do-while 循环的语法。在两种情况下，多语句循环体都是单语句循环体的一个特殊情况。多语句循环体是一个单独的复合语句。示例 2.4 和示例 2.5 分别给出了 while 语句和 do-while 语句的例子。

#### 示例 2.4 while 语句示例

```
1 #include <iostream>
2 using namespace std;

3 int main( )

4 {
5     int countDown;
```

```

6      cout << "How many greetings do you want? ";
7      cin >> countDown;

8      while (countDown > 0)
9      {
10         cout << "Hello ";
11         countDown = countDown - 1;
12     }

13     cout << endl;
14     cout << "That's all!\n";

15     return 0;
16 }

```

### 示例运行结果1

```

How many greetings do you want? 3
Hello Hello Hello
That's all!

```

### 示例运行结果2

```

How many greetings do you want? 0
That's all!

```

函数体没有执行。

---

while 语句和 do-while 语句的一个重大差别在于控制布尔表达式的检查。对于 while 语句，在运行循环体语句之前，首先对布尔表达式进行检查，如果布尔表达式为假，整个循环体就不会执行。而对于 do-while 语句，则先执行循环体语句，然后再检查布尔表达式。因此，do-while 语句至少会执行一次循环体语句。在循环体开始执行以后，while 语句和 do-while 语句就没有什么差别了，每次循环迭代之后，首先检查布尔表达式，如果布尔表达式为真，则继续执行循环体语句，否则终止循环。

## 示例 2.5 do-while 循环示例

---

```

1  #include <iostream>
2  using namespace std;

3  int main()
4  {
5      int countDown;

6      cout << "How many greetings do you want? ";
7      cin >> countDown;

8      do
9      {
10         cout << "Hello ";
11         countDown = countDown - 1;
12     } while (countDown > 0)

13     cout << endl;

```

```

14      cout << "That's all!\n";

15      return 0;

16  }

```

### 示例运行结果1

```

How many greetings do you want? 3
Hello Hello
That's all!

```

### 示例运行结果2

```

How many greetings do you want? 0
Hello
That's all!

```

函数体至少执行1次。

## while 语句和 do-while 语句的语法

### 单语句循环体的 while 语句

```

while (Boolean_Expression)
    Statement

```

### 多语句循环体的 while 语句

```

while (Boolean_Expression)
{
    Statement_1
    Statement_2
    .
    .
    .
    Statement_Last
}

```

### 单语句循环体的 do-while 语句

```

do
    Statement
while (Boolean_Expression);

```

### 多语句循环体的 do-while 语句

```

do
{
    Statement_1
    Statement_2
    .
    .
    .
    Statement_Last
} while (Boolean_Expression);

```

不要忘记末尾的分号。

不执行  
循环体

执行 while 语句时, 首先会对布尔表达式进行求值。如果布尔表达式的值为假, 则循环体语句不执行。不执行循环体语句看起来毫无意义, 但有时这正好是我们希望的操作。例如, while 语句经常用于计算一系列数字的和, 有时这一系列数字恰好为空, 或者更具体些来说, 用支票结算程序计算一个月内你签的支票的总额时, 你却恰巧度过了一个月的假, 根本就没有签过支票。在这种要计算的对象为空时, 循环体语句不应被执行。

### 再谈自增和自减运算符

通常情况下, 我们不鼓励在表达式中使用自增及自减运算符。但许多程序员习惯在 while 或 do-while 语句的布尔表达式中使用它们。如果谨慎使用, 该做法可以收到很好的效果。示例 2.6 给出了一个相关的例子。一定注意, 表达式 `count++ <= numberOfItems` 中, `count++` 的返回值是 `count` 自增之前的值。

#### 自我练习

18. 下面代码的输出是什么?

```
int count=3;
while (count--> 0)
    cout <<count<< " ";
```

19. 下面代码的输出是什么?

```
int count=3;
while (--count > 0)
    cout<< count<< " ";
```

20. 下面代码的输出是什么?

```
int n =1;
do
    cout << n <<" ";
while (n++ <= 3);
```

21. 下面代码的输出是什么?

```
int n=1;
do
    cout << n << " ";
while (++n <= 3);
```

22. 下面语句的输出是什么? (x 为 int 类型)

```
int x =10;
while(x >0)
{
    cout << x << endl;
    x = x - 3;
}
```

23. 如果将前一个练习中的 > 改为 <, 又会有什么样的输出?

24. 下列语句的输出是什么?

```
int x = 10;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

25. 下面语句的输出是什么？（x 为 int 类型）

```
int x = -42;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

26. while 语句和 do-while 语句最重要的区别是什么？

## 逗号运算符

### 逗号运算符

**逗号运算符**是一种对一系列表达式进行计算并返回最后一个表达式值的方法。正如下一节介绍的，逗号运算符一般用在 for 循环中。虽然可以在其他合法的表达式中使用逗号运算符，但我们只建议在 for 循环中使用逗号运算符。

下面的赋值语句给出了一个逗号运算符的例子：

```
result = (first = 2, second = first + 1);
```

### 逗号表达式

**逗号运算符**，如上面的语句所示就是一个逗号，赋值运算符右边的即逗号表达式。逗号运算符有两个作为操作数的表达式。本例中，逗号运算符的两个操作数分别为：

```
first = 2 和 second = first + 1
```

首先计算第一个表达式，然后是第二个表达式。正如第1章介绍的，赋值语句作为表达式时，会返回赋值运算符左边变量的值。因此，本例中逗号表达式将返回变量 second 的值，也就是说将 result 的值设为 3。

由于逗号运算符仅返回第 2 个表达式的值，第一个表达式就仅仅只是为了它的副作用。上例中，第一个表达式的副作用就是改变变量 first 的值。

可以用逗号运算符将一系列表达式连接起来，但只能在表达式的计算顺序不重要时才能这么做。当计算顺序比较重要时，应该使用括号。例如：

```
result = ((first = 2, second = first + 1), third = second + 1);
```

result 的值被设为 4。但作为对比，下面语句中 result 的最终值是不能确定的，因此该表达式的计算顺序不确定。

```
result = (first = 2, second = first + 1, third = second + 1);
```

比如，表达式 third=second+1 有可能在 second=first+1 之前进行计算。<sup>1</sup>

<sup>1</sup> 虽然 C++ 标准规定，逗号运算符连接起来的表达式的计算顺序须按照从左到右的顺序进行，但并不是所有的编译器都遵循这一点。

## 示例 2.6 表达式中的自增运算符

```

1  #include <iostream>
2  using namespace std;

3  int main
4  {
5      int numberOfItems, count,
6          caloriesForItem, totalCalories;

7      cout << "How many items did you eat today?";
8      cin >> numberOfItems;
9      totalCalories = 0;
10     count = 1;
11     cout << "Enter the number of calories in each of the\n"
12         << numberOfItems << " items eaten:\n";

13     while (count++ <= numberOfItems)
14     {
15         cin >> caloriesForItem;
16         totalCalories = totalCalories
17             + caloriesForItem;
18     }

19     cout << "Total calories eaten today ="
20         << totalCalories << endl;
21     return 0;
22 }

```

### 示例运行结果

```

How many items did you eat today? 7
Enter the number of calories in each of the
7 items eaten:
300 60 1200 600 150 1 120
Total calories eaten today = 2431

```

## for 语句

### for 语句

C++ 中第三种循环语句是 **for 语句**。for 语句一般用于以固定的步长遍历某个循环。我们在第 5 章中也将看到，for 语句也经常用于数组的遍历。但是，for 语句本身是一种应用非常广泛的循环方式，任何 while 循环可以完成的工作，for 语句也可以胜任。

例如，下面的 for 循环语句对 1 ~ 10 的整数进行累加：

```

sum = 0;
int n;
for (n = 1; n <= 10; n++)
    sum = sum + n;

```



for 语句以关键字 for 开头，紧接着是包含在括号中的三部分内容，用来指示计算机对控制变量进行何种操作。一个 for 语句的开头一般如下：

```
for (Initialization_Action; Boolean_Expression; Update_Action)
```

第一个表达式给出变量的初始化方式；第二个表达式为一个布尔表达式，用来控制循环何时终止；最后一个表达式给出每次循环迭代之后，控制变量的更新方式。

for 语句开头的三个表达式是以两个且只有两个分号互相隔开的。一定不要习惯性地最后一个表达式后面加上分号（从语法的角度来看，这是因为：这三项为表达式而非语句，因此其结尾处不需要添加分号）。

for 语句通常适用单个的 int 变量去控制循环的执行。但是，for 语句开头的三个表达式可以是任何合法的 C++ 表达式，因此，它们可以包含多个变量，而且可以是任何的类型。

采用逗号运算符，可以对括号中的第一或第三部分添加多个操作。例如，可以将变量 sum 的初始化操作移入 for 循环之内，如下所示：

```
for (sum = 0, n = 1; n <= 10; n++)
    sum = sum + n;
```

也可以将循环体中的代码移入 for 循环括号的第三项中，如下所示：

```
for (sum = 0, n = 1; n <= 10; sum = sum + n, n++);
```

但由于这种方式导致代码很难理解，这里不建议采用。

示例 2.7 给出了 for 语句的语法，并给出了与其等价的 while 语句。需要注意的是，for 语句和对应的 while 语句一样，都是首先检验循环的控制条件，然后根据结果执行循环体的代码。因此，for 循环的循环体运行的次数可能为 0。

## 示例 2.7 for 语句

### for 语句的语法

```
for (Initialization_Action; Boolean_Expression; Update_Action)
    Body_Statement
```

### 示例

```
for (number = 100; number >= 0; number--)
    cout << number
        <<" bottles of beer on the shelf.\n";
```

### 等价的 while 循环语法

```
Initialization_Action;
while (Boolean_Expression)
{
    Body_Statement
    Update_Action;
}
```

### 等价的示例

```
number = 100;
while (number >= 0)
{
```

```

cout << number
    << " bottles of beer on the shelf.\n";
number-- ;
}

```

### 示例运行结果

```

100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
.
.
.
0 bottles of beer on the shelf.

```

for 语句的循环体通常是一个复合语句，如下所示：

```

for (number = 100; number >= 0; number--)
{
    cout << number
        << " bottles of beer on the shelf.\n";
    if (number > 0)
        cout << "Take one down and pass it around.\n";
}

```

for 语句开头括号内的第一个和第三个表达式可以是任何合法的 C++ 表达式，因此可以包含任意数目、任何类型的变量。

for 语句中，可以在初始化变量的同时进行变量的声明，如下所示：

```

for (int n = 1; n < 10; n++)
    cout << n << endl;

```

不同的编译器在处理这类 for 语句时方法有所不同，这点将在 3.3 节中进行讨论。在讲到第 3 章之前，建议尽量避免使用这类 for 语句。

## for 语句

### 语法

```

for (Initialization_Action; Boolean_Expression; Update_Action)
    Body_Statement

```

### 示例

```

for (sum = 0, n = 1; n <= 10; n++)
    sum = sum + n;

```

关于 for 语句操作的解释，参见示例 2.7。

## 提示：重复、次的循环

for 语句可以设定循环体重复执行的次数。例如，下面的 for 语句将重复执行循环体三次：

```
for (int count = 1; count <= 3; count++)
    cout <<"Hip, Hip, Hurray\n";
```

for 循环体中的语句可以完全不使用循环控制变量，如变量 count。■



### 陷阱：for 语句中额外的分号

一定不要在 for 循环开头的括号后添加分号，我们通过一个例子来看这样做会有什么问题：

```
for (int count = 1; count <= 10; count++);
    cout <<"Hello\n";
```

如果没有留意括号之后额外的分号，那么你可能会希望该语句在屏幕上输出 10 次 Hello。就算是留意到了该额外的分号，你也可能会指望编译器给出错误提示信息。但这两种情况都不会发生。实际包含该循环的代码，编译过程中不会出现任何错误，且程序运行时将只会输出一个而不是 10 个 Hello。那么这是什么原因呢？要回答该问题，我们需要了解一些背景知识。

C++ 中创建一个语句的方法就是在其后添加一个分号。如果在表达式 x++ 后面加上分号，那么表达式

```
x++
```

将变为：

```
x++;
```

### 空语句

即使是单个的分号，也可以是一条语句，该语句被称为空语句。空语句不执行任何操作，但仍然是一个语句，因此下面的代码是一个完整且合法的 for 循环语句，它的循环体为空：

```
for (int count = 1; count <= 10; count++);
```

该循环的确重复执行了 10 次，但由于其循环体为空，因此，每次循环它其实什么事情也没做。

while 循环也会出现类似的情况，千万留意，不要在 while 语句开头包含布尔表达式的括号后添加分号。而 do-while 循环正好相反，必须在结尾处添加分号来结束循环。■



### 陷阱：无限循环

while 循环、do-while 循环以及 for 循环在循环控制表达式为真的情况下，会一直执行循环体而不会终止。该循环控制表达式通常会包含一个可被循环体改变的变量，该变量的值以某种方式被循环体改变，并最终为假，从而终止循环的执行。但如果由于失误使得循环控制表达式始终为真，那么循环便会一直运行下去。这种永远运行的循环被称为无限循环。

### 无限循环

遗憾的是，无限循环经常会碰到。首先我们看一个可以正常终止的循环。下面的代码输出小于 12 的正偶数，也就是它会输出 2、4、6、8 和 10，每行一个数字，然

后循环终止。

```
x = 2;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

变量  $x$  的值每次增加 2，直到 12 为止。到达 12 后，while 表达式的值不再为真，因此循环结束。

现在假设希望输出小于 12 的奇数，而不是偶数。乍一看，只需要将初始化语句改为：

```
x = 1;
```

但这么做有很大的问题，将会导致一个无限循环。因为变量  $x$  的值将从 11 直接增加到 13，所以  $x$  永远也不会等于 12。因此布尔表达式永远不会为假，循环将永不停止。

这类错误时有发生，特别是当循环终止条件是用 `==` 或者 `!=` 来判断某些数值时。处理数值时，通过判断其是否超过某个值将会更安全。例如，为了避免这类问题，将以上代码的相关部分改为如下这样：

```
while (x < 12)
```

通过这样的修改，无论  $x$  初始化为何值，循环始终会结束，从而避免无限循环的发生。

处在无限循环状态的程序，除非有外力终止它，否则程序将一直运行。既然无限循环是时有发生的情况，那么就有必要了解一下如何终止一个正在运行的程序。不同的系统终止程序的方法各异。在大多数系统上，按下 `Ctrl + C` 组合键可以终止一个程序的运行。

对于一个简单的程序，无限循环多是一种错误。但有些程序却故意设置为一直运行。例如，航班预定程序的外层主循环，它将直接受用户的预定直到关闭计算机。■

## 自测练习题

27. 下面代码的输出是什么？

```
for (int count = 1; count < 5; count++)
    cout << (2 * count) << " ";
```

28. 下面代码的输出是什么？

```
for (int n = 10; n > 0; n = n - 2)
{
    cout << "Hello";
    cout << n << endl;
}
```

29. 下面代码的输出是什么？

```
for (double sample = 2; sample > 0; sample = sample - 0.5)
    cout << sample << " ";
```

30. 将下面的循环改写为 for 循环。

```
a. int i = 1;
   while(i <= 10)
   {
       if (i < 5 && i != 2)
           cout << 'X';
       i++;
   }

b. int i = 1;
   while(i <= 10)
   {
       cout << 'X';
       i = i + 3;
   }

c. long n = 100;
   do
   {
       cout << 'X';
       n = n + 100;
   } while (n < 1000);
```

31. 下面循环的输出是什么？给出变量 n 与变量 log 之间的关系。

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2)
    log++;
cout << n << " " << log << endl;
```

32. 下面循环的输出是什么？给代码加上注释。

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2);
    log++;
cout << n << " " << log << endl;
```

33. 下面循环的输出是什么？给代码添加注释。

```
int n = 1024;
int log = 0;
for (int i = 0; i < n; i = i * 2);
    log++;
cout << n << " " << log << endl;
```

34. 对于下列每种情况，给出最适合的循环类型 (while, do-while, for)

- 一个序列的求和，如  $1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/10$ 。
- 读入一个学生的各门考试成绩。
- 读入某部门员工请病假的总天数。
- 测试一个函数，看参数不同时该函数的运行表现。

35. 下面语句的输出结果是什么？

```

int x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x + 3;
}

```

### break与continue语句

前面我们已经介绍了 while、do-while 以及 for 循环这些基本的流程控制语句，并给出了这些循环应用的一般形式和方法。但除此之外，我们还可以改变循环的执行流程，这在某些情况下是非常有用且安全的方法。改变循环执行流程的两种方式是：break 语句和 continue 语句。break 语句用来结束循环的执行；continue 语句则用来结束当前的迭代。break 语句可用于任何 C++ 循环语句中。

在介绍 switch 语句时介绍过 break 语句。break 语句由关键字 break 和其后的分号构成。执行时，break 语句终止离它最近的 switch 或循环语句。示例 2.8 给出了当输入内容不恰当时如何去终止循环的一个例子。

#### 示例 2.8 循环中的 break 语句

---

```

1  #include <iostream>
2  using namespace std;

3  int main
4  {
5      int number, sum = 0, count = 0;
6      cout << "Enter 4 negative numbers:\n";

7      while (++count <= 4)
8      {
9          cin >> number;

10         if (number >= 0)
11         {
12             cout << "ERROR: positive number"
13                 << " or zero was entered as the\n"
14                 << count << "th number! Input ends"
15                 << "with the" << count << "th number.\n"
16                 << count << "th number was not added in.\n";
17             break;
18         }

19         sum = sum + number;
20     }

21     cout << sum << " is the sum of the first"
22         << (count - 1) << " numbers.\n";

23     return 0;
24 }

```

## 示例运行结果

```

Enter 4 negative numbers:
-1 -2 -3 4
ERROR: positive number or zero was entered as the
4th number! Input ends with the 4th number.
4th number was not added in
-6 is the sum of the first 3 numbers.

```

**continue 语句** **continue** 语句包含关键字 **continue** 以及其后的分号。执行时，它终止离它最近的循环的迭代。示例 2.9 给出了一个包含 **continue** 语句的循环。

示例 2.9 包含 **continue** 语句的循环

```

1  #include <iostream>
2  using namespace std;

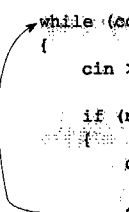
3  int main()
4  {
5      int number, sum = 0, count = 0;
6      cout << "Enter 4 negative numbers, ONE PER LINE:\n";

7      while (count < 4)
8      {
9          cin >> number;

10         if (number >= 0)
11         {
12             cout << "ERROR: positive number (or zero)!\n"
13              << "Reenter that number and continue:\n";
14             continue;
15         }
16         sum = sum + number;
17         count++;
18     }

19     cout << sum << " is the sum of the"
20          << count << " numbers.\n";
21     return 0;
22 }

```



## 示例运行结果

```

Enter 4 negative numbers, ONE PER LINE:
1
ERROR: positive number (or zero)!
Reenter that number and continue:
-1
-2
3
ERROR: positive number!
Reenter that number and continue:

```

```
-3
-4
-10 is the sum of the 4 numbers.
```

在 for 语句中使用 continue 语句时需注意一点, continue 语句会将控制转到循环的更新表达式。因此, continue 语句执行之后所有循环控制变量都会立即更新。

注意, break 语句完全结束整个循环。相反, continue 语句只是结束循环的当前迭代, 下次迭代还将继续进行 (如果有的话)。通过比较示例 2.8 和示例 2.9, 你会对此有更深入的理解, 特别要注意其中控制表达式的变化。

注意, break 语句和 continue 语句并不是必需的, 示例 2.8 和示例 2.9 中的程序都可以重新改写, 既不使用 break 语句也不使用 continue 语句。continue 语句比较复杂, 它的使用可能会导致代码不易阅读。因此, 最好避免 continue 的使用, 或者只在必要的时候使用。

### 嵌套循环

将一个循环语句嵌入另一个循环语句中是没有任何问题的。但这种情况下需注意, 任何 break 语句或者 continue 语句均对应于包含它们的最内的循环。使用嵌套循环时, 最好避免将内循环放入函数定义中或者将函数调用放在外层循环中。本书第 3 章将介绍函数的相关内容。

### 自测练习

36. break 语句有什么作用? 什么地方可以使用 break 语句?

37. 给出下面嵌套循环的执行结果。

```
int n, m;
for (n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m--)
        cout << n << " times " << m
            << " = " << n * m << endl;
```

## 2.4 文件输入简介

你会在美丽的四开纸上看到那些文字, 整齐的文字宛如那小溪蜿蜒穿过草地的边缘。

理查德·布林斯莱·谢立丹, 《造谣学校》

输入流

到目前为止, 通过之前相关示例的介绍, 我们应该很熟悉 cin 的用法了, 通过 cin 我们可以读取键盘的输入。cin 在 C++ 中是一类被称为输入流的对象。同样, 我们也可以将一个输入流关联到磁盘上的一个文件。一旦完成这种关联, 就可以像使用 cin 那样来读取文件的内容。关于文件操作的详细内容会在本书的第 12 章中进行介绍,



这其中会涉及一些我们尚未介绍的编程概念。本节不深究背后的原理，只介绍一些基本的知识，使得读者掌握从文件中读取数据的方法。

### 通过ifstream读取文本文件内容

文本文件是一种存储内容为文本的文件。在 Windows 系统下，可以通过记事本程序创建；在 Mac 系统下，可以通过 TextEdit 创建；或者对于 UNIX 系统，可以通过 vi/nano/emacs 等工具创建。大多数的字处理软件都会将文件保存为文本文件。要从该文本文件中读取数据，需在程序中包含 fstream 类库，并且包含 std 命名空间下的 ifstream 对象。因此，程序一般包含如下语句：

```
#include <fstream>
using namespace std;
```

输入流对象的声明与其他变量的声明没有什么不同。

```
ifstream inputStream;
```

接下来，需将该输入流对象和一个文本文件关联起来。

```
inputStream.open("filename.txt");
```

在指定文件名的同时，也可以给出文件的路径。对于这一点，各个系统的具体用法各不相同。本例中只简单地给出了文件名，此时文件的路径就默认为程序运行的当前目录。

一旦声明了一个输入流对象，并且用 open 将它和一个文件进行关联，程序就可以使用 >> 符号从文件中读取数据了。

例如，可以使用 inputStream >> intVar 从文件中读取一个 int 到变量 intVar 中；inputStream >> strVar 则从文件中读取一个字符串到变量 strVar 中。C++ 从文件的开头读取数据，一直到文件的结尾。当程序完成相关的读取操作后，记得执行 inputStream.close() 以关闭文件，并释放文件打开时分配的相关资源。

示例 2.10 和示例 2.11 给出了一个完整的程序。示例 2.10 显示一个名为 player.txt 文件的内容。该文件是一个普通的文本文件，可以通过其他任何字处理程序生成。作为一个例子，我们假定该文件记录的是最后一个比赛选手的相关信息。文件的第一行记录该选手的最好成绩，100510；文件的第二行记录该选手的名字，Gordon Freeman。示例 2.11 中的程序读取并显示该文件的内容。文件的读取过程就是简单地通过流提取运算符 >> 进行的。

#### 示例 2.10 示例文本文件 player.txt，用来存储选手的最高分和名字

---

```
100510
Gordon Freeman
```

---

### 示例 2.11 读取示例 2.10 文本文件的程序

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>

4  using namespace std;
5  int main()
6  {
7      string firstName, lastName;
8      int score;
9      fstream inputStream;

10     inputStream.open("player.txt");

11     inputStream >> score;
12     inputStream >> firstName >> lastName;

13     cout << "Name:" << firstName << "
14         << lastName << endl;
15     cout << "Score:" << score << endl;
16     inputStream.close();

17     return 0;
18 }
```

#### 示例运行结果

```
Name: Gordon Freeman
Score: 100510
```

我们常常需要判断程序是否读取了文件的所有内容。一个方法是通过流提取运算符 `>>` 进行：如果读取操作成功，流提取运算符 `>>` 的返回值为 `true`，否则返回 `false`。因此，尝试去读取已经到结尾的文件，`>>` 运算符会返回 `false`。这种用法看上去会有点奇怪，因为在同一行代码中同时进行了读取操作并对读取的状态进行检查。示例 2.12 通过一个 `while` 循环读取文件 `player.txt` 的内容，并将其输出。

### 示例 2.12 通过循环读取示例 2.10 文本文件的内容

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>

4  using namespace std;
5  int main()
6  {
7      string text;
8      fstream inputStream;
9
10     inputStream.open("player.txt");
11     while (inputStream >> text)
```

文件读取到结尾时将返回 `false`。

```

12     {
13         cout << text << endl;
14     }
15     inputStream.close();

16     return 0;
17 }

```

### 示例运行结果

```

100510
Gordon
Freeman

```

---

### 自测练习题

38. 下面的代码用来输出文件中的所有单词（注意，仅仅是单词），找出其中存在的问题。

```

bool moreToRead = true;
while (moreToRead)
{
    string text;
    moreToRead = inputStream >> text;
    cout << text << endl;
}

```

39. 考虑一个记录最高分的文件。文件第一行是一个整数，记录文件中包含的记录数。其后的部分是一个个的记录，每个记录由两行组成，第一行为选手的名字，第二行为选手的得分。下面是文件 `scores.txt` 的示例内容，包含三个记录：

```

3
Gordon
500
Mario
550
Illidan
385

```

编写一个程序，每行输出选手的名字和对应的得分，如：

```

Gordon, 500
Mario, 550
Illidan, 385

```

40. 改写练习 39，程序只输出得分最高的选手的记录。

## 本章小结

- 布尔表达式的求值方法与算数表达式类似。
- C++ 分支语句包含 if-else 语句和 switch 语句。
- switch 语句是一个多路分支语句。也可以通过嵌套的 if-else 语句来实现多路分支语句。
- switch 语句特别适合实现用户程序中的菜单。
- C++ 循环语句包括 while、do-while 和 for 循环。
- do-while 循环至少执行一次循环体语句。while 语句和 for 语句的循环体都有可能执行 0 次。
- for 循环可以用来实现“重复循环体  $n$  次”的效果。
- 循环可以被 break 语句提前结束，循环的每次迭代则可以通过 continue 语句提前结束。最好谨慎地使用 break 语句。对于 continue 语句，则应尽量避免它的使用，尽管一些程序员的确在极少的情况下会使用 continue 语句。
- 输入流的使用与标准库中的 cin 类似，只不过输入流是从文件中读取内容，而 cin 是从键盘读取用户的输入。

## 自测练习题答案

- a. 真。
- b. 真。注意表达式 a 和 b 同意。由于运算符 == 和 < 的优先级比 && 高，可以不加括号。不过加上括号有助于提高程序的可读性。大多数人会发现 a 中的表达式更易理解，但本质上它们表达的意思完全一样。
- c. 真
- d. 真
- e. 假。由于第一个子表达式 (count == 1) 为假，因此不管第二个子表达式是真还是假，整个表达式的值都为假，这被称为短路求值。
- f. 真。由于第一个子表达式 (count < 10) 为真，因此不管第二个子表达式取何值，整个表达式的值都为真，这被称为短路求值。
- g. 假。注意 g 中的表达式包含了 f 表达式，并将其作为子表达式。正如 f 中所述，该子表达式的值通过短路求值为真。因此，整个表达式等价于：

```
!( (count == 1) || (x < y) ) && true )
```

而上面的表达式又等价于 !(true && true)，也就是 !(true)，因此最终的结果为假。

- h. 该表达式在求值时会出错。因为第一个子表达式 ((limit/count) > 7) 包含一个除数为零的错误。

- i. 真。由于第一个子表达式 ( $\text{limit} < 20$ ) 的值为真, 根据短路求值, 整个表达式的值为真。此时不会对第二个子表达式进行求值, 因此其中存在的除数为 0 的错误也就不能被计算机发现。
  - j. 该表达式在求值过程中会产生错误, 因为第一个子表达式包含一个除数为零的错误。
  - k. 假。由于第一个子表达式 ( $\text{limit} < 0$ ) 的值为假, 根据短路求值, 整个表达式的值为假。该过程中, 计算机不会对第二个子表达式进行求值, 因此其中存在的除数为 0 的错误也就不能被计算机发现。
  - l. 如果你认为该表达式毫无意义, 那么你是对的。该表达式没有任何实际意义。但 C++ 是将 int 值转换为 bool 值, 然后进行 “&&” 及 “!” 运算。因此, C++ 将逐步计算该表达式。回顾一下, C++ 中任何非零整数都会转化为真, 而零会转化为假, 因此 C++ 将按照如下的规则计算  $(5 \ \&\& \ 7) + (!6)$ :  
在子表达式  $(5 \ \&\& \ 7)$  中, 5 和 7 转化为真, 从而得到 “true && true”, 计算得真, C++ 又会将其转化为 1。表达式  $(!6)$  中, 6 转化为真, 因此整个表达式为假, C++ 又将其转化为 0。因此整个表达式计算为  $1 + 0$ , 也就是最终结果为 1。当然了, 我们没有必要进行这些毫无意义的表达式运算, 不过稍微了解一些这方面的知识有助于你理解为什么当误将数字和布尔表达式混在一起的时候, 编译器不会报错。
2. 表达式  $2 < x < 3$  是合法的, 但它表示的并不是:

```
(2 < x) && (x < 3)
```

它的意思是  $(2 < x) < 3$ 。由于表达式  $(2 < x)$  是一个布尔表达式, 它的值或者为真或者为假, 因此可转换为 1 或者 0, 而任何一个都是小于 3 的。因此表达式  $2 < x < 3$  永远为真, 不管  $x$  取何值。

3.  $(x < -1) || (x > 2)$
4.  $(x > 1) \&\& (x < 3)$
5. 不是。在该布尔表达式中,  $(j > 0)$  为假。由于 && 运算符使用短路求值, 当第一个表达式为假时, 不会对第二个表达式进行求值。
6. 

```
(score > 100)
    cout << "High";

    cout << "Low";
```

根据程序的具体情况, 可以在上面引用字符串的末尾添加 “\n”。

7. 

```
(savings >= expenses)
{
    savings = savings - expenses;
    expenses = 0;
    cout << "Solvent";
}

{
    cout << "Bankrupt";
}
```

根据程序的具体情况，可以在上面引用字符串的末尾添加“\n”。

```
8. if ( (exam >=60) && (programsDone >=10) )
    cout <<"Passed";
```

```
else
```

```
    cout <<"Failed";
```

根据程序的具体情况，可以在上面引用字符串的末尾添加“\n”。

```
9. if ( (temperature >= 100) || (pressure >= 200) )
    cout <<"Warning";
```

```
else
```

```
    cout <<"OK";
```

根据程序的具体情况，可以在上面引用字符串的末尾添加“\n”。

10. 所有非零整数都转化为 true，0 转化为 false。

a. 0 为 false

b. 1 为 true

c. -1 为 true

11. Start

```
Hello from the second if.
```

```
End
```

```
Start again
```

```
End again
```

12. Large

13. Small

14. Medium

15. 以下都正确：

```
if (n < 0)
```

```
    cout << n << " is less than zero.\n";
```

```
else if ((0 <= n) && (n <= 100))
```

```
    cout << n << " is between 0 and 100 (inclusive).\n";
```

```
else if (n >100)
```

```
    cout << n << " is larger than 100.\n";
```

以及

```
if (n < 0)
```

```
    cout << n << " is less than zero.\n";
```

```
else if (n <= 100)
```

```
    cout << n << " is between 0 and 100 (inclusive).\n";
```

```
else
```

```
    cout << n << " is larger than 100.\n";
```

16. 3 2 1 0

17. 2 1 7 5

18. 2 1 0

19. 2 1

20. 1 2 3 4

21. 1 2 3

```
22. 10
    7
    4
    1
```

23. 没有任何输出，循环不会执行。

```
24. 10
    7
    4
    1
```

25. -42

26. 对于 do-while 循环，循环体至少执行一次；而对于 while 循环，可能出现循环根本就不执行的情况。

27. 2 4 6 8

```
28. Hello 10
    Hello 8
    Hello 6
    Hello 4
    Hello 2
```

29. 2.000000 1.500000 1.000000 0.500000

```
30. a. for (int i = 1; i <= 10; i++)
    if (i < 5 && i != 2)
        cout << 'X';
```

```
    b. for (int i = 1; i <= 10; i = i + 3)
        cout << 'X';
```

```
    c. cout << 'X' // 为了保持输出的一致，同时注意 n 初始化的变化。
        for (long n = 200; n < 1000; n = n + 100)
            cout << 'X';
```

31. 输出 1024 10。第二个数是第一个数以 2 为底的对数(如果第一个数不是 2 的幂，则产生一个近似的以 2 为底的对数)。

32. 输出为 1024 1。for 循环第一行后的分号是一个引入的错误。

33. 这是一个无限循环。考虑更新表达式  $i=i*2$ ，它不能改变  $i$  的值，因为  $i$  的初始值为 0。由于 for 循环第一行后面的分号，所以没有任何输出。

34. a.for 循环。

b. 和 c. 都需要 while 循环，因为输入的列有可能为空。

c. do-while 循环，因为执行前至少会有一次检测。

35. 无限循环。输出的前几行结果为：

```
10
13
16
19
21
```

36. break 语句用来退出循环 (while、do-while 以及 for) 或者结束 switch 语句, 除此之外, break 语句不能出现在 C++ 程序中。注意, 在一个嵌套循环中, break 语句只终止一层循环。

37. 输出的模式如下:

```
1 times 10 = 10
1 times 9 = 9
.
.
.
1 times 1 = 1
2 times 10 = 20
2 times 9 = 18
.
.
.
2 times 1 = 2
3 times 10 = 30
.
.
.
```

38. 变量 moreToRead 一直为真, 直到读取到文件的最后一个项目。通过该变量, 我们可以在循环的最后一个迭代中输出一个额外的换行符。为此, 可以先输出读取的值, 再尝试读取下一个字符串, 通过这种方式来解决这个问题。同时, 这种方式也解决了文件为空的情况。

```
string text;
bool moreToRead = inputStream >> text;
while (moreToRead)
{
    cout << text << endl;
    moreToRead = inputStream >> text;
}
```

39. 以下的代码读取第一行, 然后通过循环不断读取。

```
fstream inputStream;

inputStream.open("player.txt");
int numScores;
inputStream >> numScores;
for (int i = 0; i < numScores; i++)
{
    string name;
    int score;
    inputStream >> name;
    inputStream >> score;
    cout << name << ", " << score << endl;
}
inputStream.close();
```



40. 该方法记录当前最高的分值和姓名，待循环执行完成之后输出。

```
fstream inputStream;

inputStream.open("player.txt");
    numScores;
    int highScore = -1;
    string highName = "";
    inputStream >> numScores;
    for (int i = 0; i < numScores; i++)
    {
        string name;
        int score;
        inputStream >> name;
        inputStream >> score;
        if (score > highScore)
        {
            highScore = score;
            highName = name;
        }
    }
    inputStream.close();
    cout << highName << " has the high score of"
        << highScore << endl;
```

## 编程练习

1. 由于物价的变化，要制定一个可持续数年而不变的预算是很困难的。假设你所在的公司每年需要 200 支铅笔，你不能简单地仅凭今年的价格去预算未来两年在该项上的花费。由于通货膨胀的原因，花费通常要比今年的大。编写一个程序，估算在一段年份内的某项开支。程序要求输入该项目的开支、从现在开始此项开支要持续的年数以及对应的通货膨胀率。程序应根据指定的时间和通货膨胀率给出预算值，程序会将百分数转化为小数，如 5.6% 转化为 0.056。借助于一个循环来实现该程序。
2. 假定采用如下的方式来购买一套价值 1000 美元的音响：没有现付，利率为每年 18%（即每月 1.5%），月供为 50 美元。月供的 50 美元首先用来支付利息，余额再用来支付欠款。例如，第一个月的 50 美元中，包含 1000 美元的 1.5%，即 15 美元的利息，剩下的 35 美元用来偿还欠款，即第一个月支付 50 美元后，你的债务变为 965.00 美元。第二个月支付 965.00 美元的 1.5% 作为利息。因此可以从剩余债务中再减去 35.52 美元，依此类推，直到偿还所有债务。  
编写一个程序，计算需要多少个月才能偿还所有的债务，以及你在整个期间所支付的总利息。
3. 假设你可以从自动售货机购买巧克力，一美元可以购买一块。每块巧克力中都包含一张礼券，用七张礼券可以兑换一块巧克力。假设你有  $n$  美元，那么你可以吃到多少块巧克力？

例如, 假设你有 20 美元, 可以先购买 20 块巧克力。这样可以得到 20 张礼券, 可以用 14 张礼券兑换两块巧克力, 而这两块巧克力又包含两张礼券, 这样就有了 8 张礼券。接下来你又可以换得一块巧克力, 最后你共有 23 块巧克力和余下的两张礼券。

编写一个程序, 输入美元数, 输出可以得到的巧克力个数, 以及余下的礼券数。处理该问题最简单的办法就是采用循环。

4. 编写一个程序, 找出所有位于 3 ~ 100 之间的质数。质数是只能被 1 及自身整除的数 (如 3、5、7、11、13、17 等)。

解决问题的一种方法是双重嵌套循环。外循环可以从 3 ~ 100 进行迭代, 而内循环则检查外循环的计数值是否为质数。判定一个数  $n$  是否是质数的方法是从  $2 \sim n-1$  进行循环, 如果当中存在一个数可以整除  $n$ , 那么  $n$  不是质数。如果这些数都不能整除  $n$ , 那么  $n$  必然是质数 (注意, 可以采用多种方法对该过程进行优化)。

5. 密码算数谜题是一种用字母书写的等式。其中每个字母代表一个 0 ~ 9 的数字, 且不同的字母代表不同的数字。例如:

SEND + MORE = MONEY

该谜题的一个解为: S=9, R=8, O=0, M=1, Y=2, E=5, N=6, D=7。编写相关的程序解决如下的密码算数谜题:

TOO + TOO + TOO + TOO = GOOD

对该问题最简单的办法是对每个不同的字母 (本例中为 T, O, G, D) 采用循环, 从而构成一个嵌套循环。具体来讲就是, 对每个字母从 0 ~ 9 分别循环。例如, 程序会首先检查 T = 0, O = 0, G = 0, D = 0 的情况, 之后再检查 T = 0, O = 0, G = 0, D = 1, 之后检查 T = 0, O = 0, G = 0, D = 2, ..., 一直到 T=9, O=9, G=9, D=9。循环体中, 则检查不同的字母是否代表的数字不同以及相关的等式是否满足, 然后输出满足条件的值。

6. 阿基米德定理给出, 一个物体的浮力等于它所排开的液体的重量, 具体公式如下:

$$F_b = V \times r$$

其中  $F_b$  代表物体受到的浮力,  $V$  是物体排开的液体体积,  $r$  是该液体的密度。如果  $F_b$  大于或者等于物体自身的重量, 那么它将漂浮在液体中, 否则会沉没。编写程序判断一个给定的球体是否会漂浮在水中, 该程序的输入包括: 球体的重量 (磅)、球体的半径 (英尺), 其中水的密度采用  $r=62.4 \text{ lb/ft}^3$ , 球体积的计算可以采用公式:  $(4/3)\pi r^3$

7. 编写程序计算  $N$  道课堂练习题的得分情况。用户首先输入  $N$  的值, 紧接着分别输入  $N$  道题目的得分及总分情况, 程序根据这些信息, 计算该用户得分的百分比 (总的得分除以总满分)。例如:

```

How many exercises to input?3

Score received for exercise 1:10
Total points possible for exercise 1:10

Score received for exercises 2:7
Total points possible for exercise 2:12

Score received for exercise 3:5
Total points possible for exercise 3:8

Your total is 22 out of 30, or 73.33%

```

8. 编写程序找出摄氏温度和华氏温度在数值上相等的温度。摄氏温度和华氏温度的转化如下公式所示：

$$\text{Fahrenheit} = \frac{9}{5} \text{Celsius} + 32$$

程序应该包含两个 `int` 类型的变量分别记录摄氏温度和华氏温度。程序开始时将摄氏温度初始化为 100，然后通过循环不断减小摄氏温度，并计算对应的华氏温度，直到两者相等。

9. 计算一个正数  $n$  平方根的巴比伦算法如下：

- (1) 给出一个猜测的大概结果（如  $n/2$ ），记为 `guess`。
- (2) 计算  $r = n / \text{guess}$ 。
- (3)  $\text{guess} = (\text{guess} + r) / 2$ 。
- (4) 重复执行第二步、第三步，直到满足某个给定的精度。

编写程序，输入 `double` 类型的数作为  $n$ ，利用巴比伦算法计算  $n$  的平方根，直到前后两次迭代间的差异小于 1%。最后采用两位小数的精度，输出最终的结果。编写的程序应能正确处理很大的输入值。

10. 创建一个内容为 “I hate C++ and hate programming!” 的文本文件。编写程序读取该文件的内容，将其中出现的 “hate” 全部替换为 “love” 后，输出到控制台。编写的程序应具有通用性，可以正确处理其他的文本文件。
11. 计算理想体重的一个方法是：身高的 5 英尺部分对应体重 110 磅；高于 5 英尺的部分，每英寸对应体重 5 磅。创建一个文件，记录每个人的名字和对应身高的英尺数和英寸数，格式如下：

```

Tom Atto
6
3
Eaton Wright
5
5
Cary Oki

```

5

11

编写程序，读取该文件的内容，根据身高信息计算每个人的理想体重，并输出到终端控制台。程序应采用循环读取不同的记录信息，从而计算各人的理想体重。

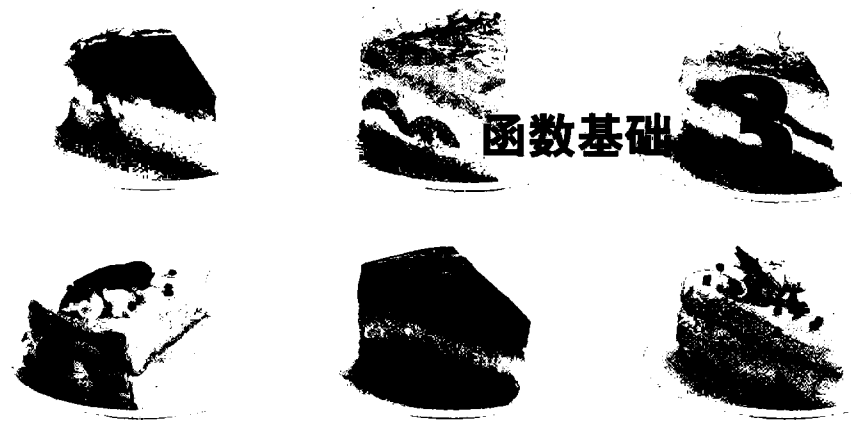
12. 该练习基于 Steve Wolfman 的“俏皮作业” (<http://nifty.stanford.edu/2006/wolfman-pretid>)。

考虑实际生活中的一系列数字，例如选修不同课程的学生数目、Facebook 上不同状态的回复数、不同图书馆的藏书量等。列中每个数字的第一位属于  $1 \sim 9$ ，初一看，大家可能会觉得它们出现的概率完全相同，但是 Benford 定律却指出，数字 1 出现的概率约为 30%，随着数字的增大，出现的概率越来越小，例如数字 9 出现的概率就约为 5%。

编写程序验证 Benford 定律。从实际生活中收集至少包含 100 个数字的序列，并保存到一个文本文件中。程序采用一个循环从该文件中读取数字，并统计各数字出现的概率。

如果将文件中的数字读入到一个名为 `strNum` 的 `string` 类型的变量中，那么就可以通过 `strNum[0]` 来访问该数字的第一位，关于 `string` 的详细介绍可参考第 9 章。





## 函数基础

### 3.1 预定义函数 82

带有返回值的预定义函数 82

预定义的void函数 86

随机数生成器 88

### 3.2 自定义函数 91

定义带有返回值的函数 91

函数声明的另一种形式 93

陷阱：参数顺序的错误 93

调用函数的函数 94

示例：一个四舍五入的函数 94

返回布尔值的函数 96

定义void函数 96

void函数中的return语句 98

前提条件和运行结果 98

main函数 100

递归函数 100

### 3.3 作用域规则 101

局部变量 101

过程抽象 103

全局常量和全局变量 104

语句块 106

嵌套作用域 107

提示：在分支和循环语句中使用函数调用 107

for循环体中的变量声明 107

# 第3章 函数基础

幸福来自点滴的积累。

一句俗语

## 概述

如果你之前使用过其他的编程语言，那么本章的内容对你而言将不会太陌生。通过本章的学习，你可以掌握 C++ 函数的相关术语和语法基础。第 4 章将介绍 C++ 函数不同于其他语言的部分。

一个程序可以看作是由一些子部分构成的，比如读取输入数据、计算输出数据、显示输出数据等。同其他编程语言一样，C++ 有专门的工具来命名并编写每个子部分的代码。C++ 中，这些子部分被称为函数。其他程序设计语言也有函数或者类似函数这样的概念（虽然名称或许不同）。比如其他语言中有 `procedure`（过程）、`subprogram`（子程序）或者 `method`（方法）的概念，这些其实就是函数的同义词。C++ 中，函数可以返回一个值，也可以仅仅执行一些操作而不返回任何值。本章将介绍 C++ 函数的基础知识。在介绍如何编写自定义函数之前，我们将首先介绍如何使用预定义的 C++ 函数。

## 3.1 预定义函数

不要重新发明轮子。

一句俗语

void 函数

C++ 存在一些预定义函数库，可以在程序中直接调用。C++ 中有两类函数：带有返回值的函数和没有返回值的函数。没有返回值的函数又被称为 **void 函数**。我们首先介绍带有返回值的函数，之后介绍 void 函数。

### 带有返回值的预定义函数

参数  
返回值

我们以 `sqrt` 函数为例介绍带有返回值的预定义函数的用法。`sqrt` 函数计算给定数字的平方根。例如，对于数字 9，`sqrt` 函数返回 9 的平方根——3。函数的输入值被称为**参数**，最终计算得到的结果被称为**返回值**。函数可以有一个以上的参数，但所有函数最多只能有一个返回值。

函数的使用非常简单。要将 9.0 的平方根赋给变量 `theRoot`，可以采用以下程序语句：

```
theRoot=sqrt(9.0)
```

函数调用

表达式 `sqrt(9.0)` 被称为**函数调用**。函数调用中的实参可以是常量（如 9.0）、变量或是更复杂的表达式。函数调用作为一个表达式，可以像其他表达式一样使用。例如，`sqrt` 函数的返回值是 `double` 类型，因此以下语句合法：

```
bonus=sqrt(sales)/10;
```

变量 `sales` 和 `bonus` 都是 `double` 类型的。函数调用 `sqrt(sales)` 是一个单独的项，就好像用圆括号括起来一样。因此上面的赋值语句等价于：

```
bonus = (sqrt(sales))/10;
```

任何可以使用某种类型表达式的地方，也都可以使用具有相同类型返回值的函数调用。

示例 3.1 作为一个完整的可运行程序，详细展示了预定义函数 `sqrt` 的使用方法。这个程序根据使用者花费的金额，计算出可以修建的犬舍的面积。程序会询问使用者的预算金额，然后计算出可以购买的地板面积。最后给出方形犬舍的边长。

`cmath` 函数库包含了函数 `sqrt` 的定义以及其他常用的数学函数。如果程序中使用了某些预定义函数，那么程序中一定要使用 **include 指令** 来包含相关的函数库。例如，示例 3.1 的程序调用了 `sqrt` 函数，因此它包含如下的代码：

```
#include <cmath>
```

这个程序含有两个 `include` 指令，`include` 指令的顺序并不重要。本书的第 1 章对 `include` 指令做了详细的介绍。

预定义函数通常放置在 `std` 命名空间中，因此要求如下 `using` 命令，如示例 3.1 所示。

```
using namespace std;
```

### 示例 3.1 带有返回值的预定义函数

```
1 // 根据用户的预算，计算可以购买的犬舍的尺寸大小。
2 #include <iostream>
3 #include <cmath>
4 using namespace std;

5 int main()
6 {
7     const double COST_PER_SQ_FT = 10.50;
8     double budget, area, lengthSide;

9     cout << "Enter the amount budgeted for your doghouse $";
10    cin >> budget;

11    area = budget / COST_PER_SQ_FT;
12    lengthSide = sqrt(area);

13    cout.setf(ios::fixed);
14    cout.setf(ios::showpoint);
15    cout.precision(2);
16    cout << "For a price of $" << budget << endl
17         << "I can build you a luxurious square doghouse\n"
18         << "that is " << lengthSide
19         << " feet on each side.\n";
```



```
20     return 0;
21 }
```

示例运行结果

```
Enter the amount budgeted for your doghouse $25.00
For a price of $25.00
I can build you a luxurious square doghouse
that is 1.54 feet on each side.
```

仅仅使用  
include  
指令还不够

一般情况下，要使用一个函数库，只需在程序中用 include 指令引入该函数库，并采用 using 指令包含 std 命名空间即可。但是，对于某些系统下的函数库，为了程序的正常编译和运行，还需设定相关编译选项和链接选项。具体的细节随系统不同而不同，遇到上述问题时，请查阅相关操作手册或请教专业人士寻求帮助。

abs 和 labs

示例 3.2 给出了一些预定义函数。附录 D 给出了更多的预定义函数。注意，绝对值函数 **abs** 和 **labs** 包含在头文件为 cstdlib 的库中，因此使用这两个函数的程序必须包含如下的 include 命令：

```
#include <cstdlib>
```

带有返回值的函数

对于带有返回值的函数，函数调用是一个由函数名以及带括号的参数列表组成的表达式。如果该函数包含多个参数，则参数之间用逗号隔开。如果该函数有返回值，那么就可以像同类型的表达式一样使用。

语法

```
Function_Name (Argument_List)
```

其中 Argument\_List 是由逗号分隔的参数列表：

```
Argument_1, Argument_2, . . . , Argument_Last
```

示例

```
side = sqrt(area);
cout << "2.5 to the power 3.0 is "
      << pow(2.5, 3.0);
```

fabs

同样要注意共有三个求绝对值函数。如果希望求 int 数值类型的绝对值，使用 abs；如果求 long 数值类型的绝对值，使用 labs；如果求 double 类型的绝对值，使用 fabs。情况更复杂的是，abs 和 labs 包含在头文件 cstdlib 中，而 **fabs** 则包含在头文件 cmath 中。fabs 是浮点绝对值的缩写。这里我们回顾一下，带有小数部分的数字，（如 double 类型的数）通常被称为浮点数。

pow

另一个预定义函数的例子是 pow，包含在头文件 cmath 中。函数 pow 用来做指数运算。例如，将一个变量的值设为  $x^y$ ，可以采用如下的语句：

```
result = pow(x, y);
```

因此，以下三行代码将输出 9.0，因为  $(3.0)^{2.0}=9.0$ ：

```
double result, x = 3.0, y = 2.0;
result = pow(x, y);
cout << result;
```

### 示例 3.2 一些预定义函数

所有预定义函数均要求使用 include 指令和 using std 指令。

函数名	描述	实参类型	返回类型	示例	结果	头文件
sqrt	平方根	double	double	sqrt(4.0)	2.0	cmath
pow	乘方	double	double	pow(2.0, 3.0)	8.0	cmath
abs	int 类型的绝对值	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	long 类型的绝对值	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	double 类型的绝对值	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	向上取整	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	向下取整	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	结束程序	int	void	exit(1)	-	cstdlib
rand	随机数	-	int	rand()	随机数	cstdlib
srand	设置随机数因子	unsigned int	void	srand(42)	-	cstdlib

#### 参数的类型

注意，上面对 pow 函数的调用，得到的结果是 9.0，而不是 9。需要注意的是，pow 函数的返回值为 double 类型。还需要注意的是，pow 函数需要两个参数。函数可以有任意数目的参数，且每个参数都有自己的类型。函数调用时必须满足对应的参数类型。许多情况下，如果使用了不同类型的参数，C++ 会自动进行类型转换，但结果不一定是我们所期望的。但有一个例外，即从 int 类型向 double 类型的转换是自动的。很多情况下，包括 pow 函数调用，即使函数规定的参数类型为 double，也可以安全地使用 int 类型的参数。

#### pow 函数的限制

许多 pow 函数的实现参数都有一定的限制。如果第一个参数为负，那么第二个参数就必须为整数。为安全起见，建议使用 pow 函数时，确保第一个参数为正。

## void 函数

void 函数执行一定的操作，但没有任何返回值。对于 void 函数而言，函数调用就是一个包含函数名、带有括号的参数列表，并以分号结束的语句。如果包含多个参数，参数之间以逗号隔开。对于 void 函数，函数调用可以像其他 C++ 语句那样使用。

### 语法

```
Function_Name (Argument_List);
```

其中 *Argument\_List* 是一个逗号隔开的参数列表：

```
Argument_1, Argument_2, . . . , Argument_Last
```

### 示例

```
exit(1);
```

## 预定义的void函数

void 函数执行某些特定操作，但没有返回值，因此一个 void 函数的调用就是一个语句。void 函数的使用与带有返回值的函数调用类似。唯一不同的是，void 函数调用是被当作一个语句来使用的而非表达式。预定义的 void 函数与带返回值的预定义函数，使用方法相同，也必须包含相关的 include 指令和 using std 指令。

例如，**exit** 函数定义在 `cstdlib` 库中，因此使用它的程序必须包含如下语句：

```
#include <cstdlib>
using namespace std;
```

下面是一个 exit 函数调用的例子：

```
exit(1);
```

## exit 函数

exit 函数是一个预定义的 void 函数，包含一个 int 类型的参数。exit 函数的调用采用如下格式：

```
exit(Integer_Value);
```

exit 函数的调用将结束程序的运行，其参数可以是任何整数。但习惯上，由于错误造成的 exit 调用使用参数 1，其他情况则使用参数 0。

exit 函数包含在 `cstdlib` 库中，并放在 `std` 命名空间之下。因此任何使用 exit 函数的程序均需包含如下语句：

```
#include <cstdlib>
using namespace std;
```

exit 函数的调用将立即结束程序的运行。示例 3.3 展示了 exit 函数的用法。

注意，exit 函数包含一个 int 类型的参数，这个参数被传递给操作系统。就

C++ 程序本身而言，可以使用任何 `int` 数值作为参数。但习惯上，由于错误造成的 `exit` 调用使用参数 1，其他情况则使用参数 0。

### 示例 3.3 预定义 `void` 函数的调用

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
```

这里仅仅是一个例子，如果删除这几行语句，程序的输出不会有任何改变。

```
4 int main()
5 {
6     cout << "Hello Out There!\n";
7     exit(1);

8     cout << "This statement is pointless, \n"
9         << "because it will never be executed. \n"
10        << "This is just a toy program to illustrate exit.\n";

11     return 0;
12 }
```

#### 示例运行结果

```
Hello Out There!
```

同样，`void` 函数也可以有任意数目的参数。`void` 函数的参数与带有返回值的函数的参数完全相同。如果函数调用时采用了不同的参数类型，C++ 会自动进行一些类型转换，但最终的结果可能不是我们所期望的。

### 自测练习题

#### 1. 给出下列算数表达式的值。

<code>sqrt(16.0)</code>	<code>sqrt(16)</code>	<code>pow(2.0, 3.0)</code>
<code>pow(2, 3)</code>	<code>pow(2.0, 3)</code>	<code>pow(1.1, 2)</code>
<code>abs(3)</code>	<code>abs(-3)</code>	<code>abs(0)</code>
<code>fabs(-3.0)</code>	<code>fabs(-3.5)</code>	<code>fabs(3.5)</code>
<code>ceil(5.1)</code>	<code>ceil(5.8)</code>	<code>floor(5.1)</code>
<code>floor(5.8)</code>	<code>pow(3.0, 2)/2.0</code>	<code>pow(3.0, 2)/2</code>
<code>7/abs(-2)</code>	<code>(7 + sqrt(4.0))/3.0</code>	<code>sqrt(pow(3, 2))</code>

#### 2. 将下列数学表达式改写为 C++ 算数表达式。

- $\sqrt{x+y}$
- $x^{y+7}$
- $\sqrt{\text{area} + \text{fudge}}$
- $\frac{\sqrt{\text{time} + \text{tide}}}{\text{nobody}}$

$$e. \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$f. |x - y|$$

3. 编写一个完整的 C++ 程序，计算并输出 1 ~ 10 所有整数的平方根。

4. exit 函数中，int 参数的功能是什么？

## 随机数生成器

随机数生成器是一个可以返回随机数的函数。它与我们前面所讲的函数的不同之处在于，它的返回值不是由参数决定的（事实上，它通常没有参数），而是由某些全局条件来决定的。由于它的返回值是一些随机数，因此可以使用随机数生成器模拟一些随机事件，比如掷色子或者抛硬币。除了可以模拟随机事件之外，随机数生成器还可以用来模拟那些严格来说并非随机，然而表面上是随机的事件，比如经过收费站的车辆的时间间隔。

rand  
RAND\_MAX

C++ 函数库 cstdlib 中包含了一个名为 rand 的随机数生成函数。这个函数没有参数。当程序调用该函数时，函数随机地返回一个位于 0 和 RAND\_MAX 之间的整数，包括 0 和 RAND\_MAX。RAND\_MAX 是一个预定义的整数常量，它的定义包含在头文件为 cstdlib 的库中。具体的值与系统相关，但一般最小为 32767（两字节整数的最大值）。例如，下列语句将输出一列 10 个随机数字，位于 0 到 RAND\_MAX 之间。

```
int i;
for (i = 0; i < 10; i++)
    cout << rand() << endl;
```

有时，我们希望产生的随机数在一个较小的范围内，如 0 ~ 10。为了保证产生的随机数位于 0 ~ 10 之间，可以使用：

```
rand() % 11
```

比例缩放

这叫作比例缩放。下面输出的 10 个随机数就位于 0 ~ 10 之间。

```
int i;
for (i = 0; i < 10; i++)
    cout << (rand() % 11) << endl;
```

随机数生成器（如函数 rand）并不产生真的随机数。函数 rand 的一系列调用，将产生一系列看似随机的数字。也就是说，如果使计算机的状态回到调用开始时的状态，重新调用一遍的话，就可以得到一系列相同的“随机数”。看似随机其实不随机的数，如 rand 函数产生的一系列数，叫作伪随机数。

伪随机数  
种子  
srand

伪随机数的产生通常由一个被称为种子的数决定。如果用同一个 seed 启动随机数生成器，不管调用多少次，得到的都是相同的一组数。可以通过 srand 函数来设定函数 rand 的 seed。srand 函数以一个正整数为参数，设置 seed 的值。例如，以下语句将输出两列完全相同的 10 个伪随机数：

```
int i;
```

```

srand(99);
for (i = 0; i < 10; i++)
    cout << (rand() % 11) << endl;
srand(99);
for (i = 0; i < 10; i++)
    cout << (rand() % 11) << endl;

```

这里函数 `srand` 采用 99 进行 `seed` 的初始化，也可以将 99 替换为其他任何的正整数，程序仍将输出两列完全相同的 10 个伪随机数。

注意，给定一个 `seed` 之后产生的一系列伪随机数是随系统不同而不同的。同一个 `seed`，不同的系统得到的伪随机数是不同的。但是，如果在同一个系统中使用相同的 `seed`，那么总是可以得到相同的伪随机数。

### 伪随机数

函数 `rand` 没有参数，返回一个位于 0 和 `RAND_MAX` 之间（包含 0 和 `RAND_MAX`）的随机整数。`srand` 函数带有一个参数，该参数是随机数生成器的 `seed`。函数 `srand` 的参数类型为 `unsigned int`，因此函数调用的实参须为非负正数。函数 `rand` 和 `srand`，以及常量 `RAND_MAX` 都定义在库 `cstdlib` 中。因此，调用这些函数的程序都必须包含如下语句：

```

#include <cstdlib>
using namespace std;

```

大多数应用中，伪随机数完全可以替代真随机数。事实上，伪随机数有时比真随机数更合适。伪随机数相对真随机数有一大优势：产生的随机数是可以再次重现的。以同一 `seed` 值，再运行一次，就会产生相同的随机数序列。对于许多具体的应用来讲，这十分方便。当发现并纠正一个错误后，程序可以以相同的随机数序列重新运行。因此只要采用的是伪随机数，程序的运行状态总是可以重现的。而对于真随机数生成器，程序的运行状态每次都不一样。

示例 3.4 给出了一个采用随机数生成器预测天气的程序。很明显，这个预测是随机的、不准确的。

注意，在示例 3.4 中，用作 `srand` 函数的参数是月份乘以日期。因此，每次运行程序，只要输入相同的日期，都会产生同样的天气预报结果。

随机浮点数

概率经常用介于 0.0 ~ 1.0 之间的浮点数表示。假如我们希望用随机概率代替随机整数，可以用另一种形式的比例缩放。以下语句产生一个位于 0.0 ~ 1.0 之间的伪随机浮点数：

```

(RAND_MAX - rand()) / static_cast<double>(RAND_MAX)

```

其中的类型转换使得我们可以得到浮点除法而非整数除法。

### 示例 3.4 使用随机数生成器的函数

```

1 #include <iostream>
2 #include <cstdlib>

```

```

3  using namespace std;

4  int main()
5  {
6      int month, day;
7      cout << "Welcome to your friendly weather program.\n"
8          << "Enter today's date as two integers for the month" <<
          "and the day:\n";
9      cin >> month;
10     cin >> day;
11     srand(month * day);
12     int prediction;
13     char ans;
14     cout << "Weather for today:\n";
15     do
16     {
17         prediction = rand( ) % 3;
18         switch (prediction)
19         {
20             case 0:
21                 cout << "The day will be sunny!!\n";
22                 break;
23             case 1:
24                 cout << "The day will be cloudy.\n";
25                 break;
26             case 2:
27                 cout << "The day will be stormy!\n";
28                 break;
29             default:
30                 cout << "Weather program is not"
31                     << "functioning properly.\n";
32         }
33         cout << "Want the weather for the next day?(y/n):";
34         cin >> ans;
35     } while (ans == 'y' || ans == 'Y');
36     cout << "That's it from your 24-hour weather program.\n";
37     return 0;
38 }

```

### 示例运行结果

```

Welcome to your friendly weather program.
Enter today's date as two integers for the month and the day:
2 14
Weather for today:
The day will be cloudy.
Want the weather for the next day?(y/n): y
The day will be cloudy.
Want the weather for the next day?(y/n): y
The day will be stormy!
Want the weather for the next day?(y/n): y
The day will be stormy!
Want the weather for the next day?(y/n): y
The day will be sunny!!
Want the weather for the next day?(y/n): n

```

That's it from your 24-hour weather program.

### 白兔练习

5. 给出一个表达式，产生位于 5 ~ 10 之间（包括两端）的一个伪随机整数。
6. 编写一个程序，要求使用者输入一个 seed，然后根据这个 seed 产生一系列 10 个随机数字。这 10 个随机数字必须位于 0.0 ~ 1.0 之间（包括两端）。

## 3.2 自定义函数

定制西装总是比成衣更适合自己的。

我的叔叔，裁缝师

上一节介绍了预定义函数的使用。本节将介绍自定义函数。

### 定义带有返回值的函数

自定义函数可以放在 main 函数所在的文件中，也可以放在一个单独的文件中以便重用在不同的程序中。两种方式没有本质上的差别，这里我们假设所有的函数定义都在同一个文件中。本节只介绍带有返回值的自定义函数，下一小节介绍无返回值的自定义函数。

示例 3.5 给出了自定义函数的实现以及使用。该函数名为 totalCost，带有两个参数：单个项目的价格以及购买的项目的数目。函数返回总的价格，包括消费税在内。自定义函数的调用方法与预定义函数完全相同。

自定义函数的实现分两部分。第一部分称为函数声明或者函数原型。以下是示例 3.5 中相关的函数声明语句：

```
double totalCost(int numberParameter, double priceParameter);
```

返回值的  
类型

函数声明中的第一个 double 指明了函数返回值的类型。对于 totalCost 函数而言，它的返回值为 double 类型。接下来是函数的名字——totalCost。函数声明给出了编写以及使用该函数的所有信息。它给出函数参数的个数以及类型。本例中，函数 totalCost 共有两个参数：第一个为 int 类型，第二个为 double 类型。标识符 numberParameter 和 priceParameter 称为形参，或简称参数。形参给出了函数调用时实参的数目和类型。在编写函数时，并不知道函数调用时要使用什么样的实参，因此用形参来代替实参。形参可以是任何合法的标识符。注意函数声明以分号结束。

形参

函数定义  
函数头

虽然函数声明给出了编写函数所需的所有信息，但并没有给出函数的返回值。函数的返回值由函数的定义或者实现来决定。示例 3.5 中 24 ~ 30 行为函数 totalCost 的定义部分。函数定义描述了函数如何计算出返回值。函数定义由函数头以及紧接着的函数体构成。函数头的写法与函数声明完全相同，只不过结尾处没有分号。返回值由函数体的语句决定。



## 函数体

函数体紧跟在函数头之后，是函数定义的主体部分。函数体由声明部分以及包含在一对大括号之内的可执行语句构成。因此函数体就像是一个程序的 main 部分。函数调用时，实参传递给形参，然后执行函数体内的语句。当执行到函数体的 return 语句时，函数的返回值得以确定，函数执行结束。

## 示例 3.5 使用随机数生成器的函数

```

1  #include <iostream>
2  using namespace std;

3  double totalCost(int numberParameter, double priceParameter);
4  // 计算总的价格，包括 5% 的销售税。
5  int main()
6  {
7      double price, bill;
8      int number;

9      cout << "Enter the number of items purchased: ";
10     cin >> number;
11     cout << "Enter the price per item $";
12     cin >> price;

13     bill = totalCost(number, price);

14     cout.setf(ios::fixed);
15     cout.setf(ios::showpoint);
16     cout.precision(2);
17     cout << number << " items at "
18          << "$" << price << " each.\n"
19          << "Final bill, including tax, is $" << bill
20          << endl;

21     return 0;
22 }

23 double totalCost(int numberParameter, double priceParameter)
24 {
25     const double TAXRATE = 0.05; //5% 销售税
26     double subtotal;

27     subtotal = priceParameter * numberParameter;
28     return (subtotal + subtotal*TAXRATE);
29 }
```

函数声明，又称函数原型

函数调用

函数头

函数体

函数定义

## 示例运行结果

```

Enter the number of items purchased: 2
Enter the price per item: $10.10
2 items at $10.10 each.
Final bill, including tax, is $21.21
```

**return** 语句由关键字 **return** 和一个表达式组成。示例 3.5 中的函数定义包含如下的 **return** 语句：

```
return (subtotal + subtotal * TAXRATE);
```

执行这个 **return** 语句时，下面表达式的值，将作为函数的返回值。

```
(subtotal + subtotal * TAXRATE)
```

表达式中的括号其实可以省略，不影响程序运行的结果。但如果表达式较长，加上括号可以使 **return** 语句易于阅读。为了统一起见，一些程序员建议哪怕是简单的表达式也要加上括号。在示例 3.5 的函数定义中，**return** 语句之后没有任何语句，但如果有的话，这些语句也不会被执行。**return** 语句的执行标志着函数调用的结束。

注意，函数体可以包含任何合法的 C++ 语句，并在函数调用时执行。因此带有返回值的函数除了返回一个值之外，还可以完成其他操作。但一般来讲，调用带返回值的函数的主要目的就为了获得该返回值。

函数调用之前，必须首先给出该函数的完整定义或者对该函数进行声明。最常见的安排是在程序的 **main** 部分之前进行相关的函数声明，而对应函数的定义则出现在另外的文件中。由于目前我们还没有接触到程序分散在多个文件中的用法，因此这里将对应函数的定义放在程序的 **main** 部分之后。另外，如果完整的函数定义出现在 **main** 部分之前，则可省略对应的函数声明。

### 函数声明的另一种形式

我们可以省略函数声明（函数原型）中的形参名字，而不影响程序的正常执行。下面两种函数声明是等价的：

```
double totalCost(int numberParameter, double priceParameter);
```

和

```
double totalCost(int, double);
```

通常我们使用第一种形式，因为这样可以对函数做详细的注释，如给出每个形参的描述信息。第二种形式多见于相关的编程手册中。

需要注意的是，这种形式仅限于函数声明，函数定义必须给出相关的形参名字。

### 陷阱：参数顺序的错误

函数调用时，计算机用第一个实参代替第一个形参，用第二个实参代替第二个形参，依此类推。虽然计算机检查每个实参的类型，但并不会检查其合理性。如果你将实参的顺序弄混，计算机不会按你希望的那样运行。如果类型不匹配且不能进行相关的类型转换，计算机将报错，从而你可以发现此类错误。如果碰巧类型匹配，或者计算机可以做一些自动类型转换，程序将照常运行，那么其中的错误将很难被发现。■

## 调用函数的函数

函数体可以包含对另一个函数的调用。这种类型的函数调用与程序 main 部分中的函数调用完全相同，唯一的限制在于函数调用之前必须对函数进行声明（或定义）。虽然我们可以在函数定义中包含其他函数调用，但不能在一个函数定义中包含另一个函数定义。

### 示例：一个四舍五入的函数

预定义函数（示例 3.2）中没有包括对某个数进行四舍五入的函数。函数 `ceil` 和 `floor` 有类似的功能，但不是四舍五入函数。`ceil` 函数返回邻近的最大整数。因此 `ceil(2.1)` 返回整数 3.0，而不是 2.0。

函数 `floor` 返回邻近的小于或等于实参的整数，因此 `floor(2.1)` 返回的是 2.0 而不是 3.0。幸运的是，可以很容易地定义一个真正的四舍五入的函数。示例 3.6 中给出了一个具体的例子。`round` 函数返回最接近的整数，例如，`round(2.3)` 返回 2，而 `round(2.6)` 返回 3。

为了验证 `round` 函数是否能够正确执行，我们来看几个例子。考虑 `round(2.4)`，根据 `round` 函数的定义，其返回值如下（转化为一个 `int` 值）：

```
floor(2.4 + 0.5)
```

等于 `floor(2.9)`，即 2.0。事实上，对于任何大于或等于 2 且小于 2.5 的数，加上 0.5 也会小于 3.0，因此加上 0.5 后的 `floor` 函数的返回值为 2.0。由此可见，只要是大于或等于 2.0 且小于 2.5 的数，其 `round` 返回值均为 2（由于 `round` 函数定义中指明了其返回值类型为 `int`，所以计算所得的值都将进行一次类型转换）。

再看一个大于或等于 2.5 的数，如 2.6。`round(2.6)` 的返回值如下（转换为 `int` 值）：

```
floor(2.6 + 0.5)
```

等于 `floor(3.1)`，即 3.0。事实上，对于任何大于或等于 2.5 且小于或等于 3.0 的数，加上 0.5 都会大于 3.0，因此调用 `round` 时，只要是大于或等于 2.5 且小于 3.0 的数，`round` 函数的返回值均为 3。

综上所述，`round` 函数对介于 2.0 ~ 3.0 之间的数运行无误。显然，对于所有的非负数，`round` 函数都没有问题。

### 示例 3.6 round 函数

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;                                round 函数的测试程序

4 int round(double number);
5 // 假设 number >= 0.
```

```

6 // 返回四舍五入的值。
7 int main()
8 {
9     double doubleValue;
10    char ans;
11    do
12    {
13        cout << "Enter a double value: ";
14        cin >> doubleValue;
15        cout << "Rounded that number is " << round(doubleValue)<<
16        endl;
17        cout << "Again? (y/n):";
18        cin >> ans;
19    } while (ans == 'y' || ans == 'Y');
20    cout << "End of testing.\n";
21
22    return 0;
23 }
24 // 使用 cmath。
25 int round(double number)
26 {
27     return static_cast<int>(floor(number + 0.5));
28 }

```

#### 示例运行结果

```

Enter a double value: 9.6
Rounded, that number is 10
Again? (y/n): y
Enter a double value: 2.49
Rounded, that number is 2
Again? (y/n): n
End of testing.

```

#### 自测练习

##### 7. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;
char mystery(int firstParameter, int secondParameter);
int main( )
{
    cout << mystery(10, 9) << "ow\n";
    return 0;
}

char mystery(int firstParameter, int secondParameter)
{
    if (firstParameter >= secondParameter)
        return 'W';
    else
        return 'H';
}

```

8. 编写一个函数的声明及其定义, 该函数有三个 `int` 类型的参数。并返回这三个整数的和。
9. 编写一个函数的声明及其定义, 该函数有一个 `double` 类型的参数。当参数为正值时, 返回字母 'P'; 当参数为负值或 0 时, 返回字母 'N'。
10. 一个函数的定义能出现在另一个函数的定义中吗?
11. 预定义函数的调用和自定义函数的调用有什么异同点?

### 返回布尔值的函数

函数的返回值可以是 `bool` 类型。调用这样的函数会得到一个 `true` 或 `false`, 而且可以当作一般的布尔表达式来使用。例如, 可以当作布尔表达式用在 `if-else` 语句或者循环语句中。这通常可以使程序更易于阅读。通过函数声明, 可以将一个复杂的布尔表达式与一个有意义的名字联系起来, 例如, 如下语句:

```
if ((rate >= 10) && (rate < 20)) || (rate == 0))
{
    ...
}
```

可以写为:

```
if (appropriate(rate))
{
    ...
}
```

只需要定义如下的函数:

```
bool appropriate(int rate)
{
    return ((rate >= 10) && (rate < 20)) || (rate == 0);
}
```

### 自测练习

12. 编写一个函数的定义, 该函数名为 `inOrder`, 有三个 `int` 类型的参数。如果三个参数按照从小到大的顺序排列, 该函数返回 `true`, 否则返回 `false`。例如, `inOrder(1,2,3)` 及 `inOrder(1,2,2)` 都返回 `true`, 而 `inOrder(1,3,2)` 返回 `false`。
13. 编写一个函数的定义, 该函数名为 `even`, 包含一个 `int` 类型的参数, 返回一个 `bool` 类型的值。如果参数为偶数, 函数返回 `true`, 否则返回 `false`。
14. 编写一个函数的定义, 该函数名为 `isDigit`, 包含一个 `char` 类型的参数, 返回值为 `bool` 类型。如果参数为一个数字, 返回 `true`, 否则返回 `false`。

### 定义 `void` 函数

C++ 中 `void` 函数的定义与带返回值的函数的定义完全相同。例如, 下面就是一个 `void` 函数, 它输出华氏温度经计算后对应的摄氏温度。其中温度转换的计算是在

程序的其他部分完成的，该 `void` 函数只负责对该计算结果进行输出。

```
void showResults(double fDegrees, double cDegrees)
{
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << fDegrees
        << "degrees Fahrenheit is equivalent to\n"
        << cDegrees << "degrees Celsius.\n";
}
```

如上面的函数所示，`void` 函数的定义与带返回值的函数仅有两点区别：一是在函数返回值类型的地方用 `void` 代替，这告诉编译器这个函数没有返回值；`void` 的意思就是没有返回值。二是 `void` 函数的函数体中没有 `return` 语句。函数体中最后一条语句的执行代表着函数的结束。

**void  
函数定义**

**void  
函数调用**

`void` 函数的调用是一条可执行语句，例如，之前的 `showResults` 函数可以这样被调用：

```
showResults(32.5, 0.3);
```

程序执行这条语句时，屏幕上将显示如下的内容：

```
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

注意，函数调用以分号结束，这告诉编译器这是一条可执行语句。

`void` 函数被调用时，形参被实参代替，然后执行函数体中的语句。例如，本节前面所给的 `showResults` 函数，调用时，将在屏幕上输出某些内容。理解 `void` 函数调用的另一种方法是，可以认为用函数体中的语句代替了函数调用语句，函数调用时，实参取代形参，就好像函数体列在了程序中（本书第 4 章详细介绍了形参替换的机制，目前我们只采用简单的例子来直观地展示该替换过程）。

### 函数声明（函数原型）

函数声明给出了使用以及定义函数的所有信息。函数声明在代码中必须先于函数调用出现，一般放在 `main` 函数之前。

#### 语法

```
Type_Returned_Or_void FunctionName (Parameter_List);
```

其中 `Parameter_List` 是以逗号分隔的形参列表，如下所示：

```
Type_1 Formal_Parameter_1, Type_2 Formal_Parameter_2,...
Type_Last Formal_Parameter_Last
```

不要忘记这里的  
分号。

#### 示例

```
double totalWeight(int number, double weightOfOne);
// 返回总的重量。
```

```
void showResults(double fDegrees, double cDegrees);
// 显示一条信息，表示 fDegrees 华氏温度等价于 cDegrees 摄氏温度。
```

有些函数没有任何参数，此时，函数声明、函数定义以及函数调用中，都没有任何参数。例如，下面定义的 `void` 函数 `initializeScreen`，仅仅在屏幕上输出一个换行符，除此之外，什么都不做。

```
void initializeScreen( )
{
    cout << endl;
}
```

如果你的程序开始包含如下的函数调用，那么当前程序会先在屏幕上输出一个换行符，然后进行其他操作。

```
initializeScreen( );
```

一定注意，即使函数没有参数，函数声明以及函数调用中也要记得加上括号。

`void` 函数的声明、定义方式与之前介绍过的带返回值的函数并无二异。

### void 函数中的 return 语句

`void` 函数  
和 `return`  
语句

`void` 函数和带有返回值的函数都可以包含 `return` 语句。在带有返回值的函数中，`return` 语句指定了函数的返回值；在 `void` 函数中，`return` 语句不包含任何的表达式，仅仅起到终止函数调用的作用。每个带有返回值的函数都必须以 `return` 语句作为函数的结束，但 `void` 函数可以不包含 `return` 语句。如果没有 `return` 语句，执行完函数体的代码之后，`void` 函数的执行也将终止，就好像在大括号之前有一个隐含的 `return` 语句一样。

虽然 `void` 函数的结尾不用添加 `return` 语句，但这并不代表 `void` 函数不需要 `return` 语句，有时候 `return` 语句还是必不可少的。例如，示例 3.7 中的函数定义，这是一个酒店管理程序的一部分。该程序根据餐桌上客人的人数以及冰淇淋的总数来计算每位就餐的客人可以分到多少。如果就餐的客人人数为 0，那么 `if` 语句中的 `return` 语句将结束函数的执行，避免出现除数为 0 的情况；如果客人人数不为 0，那么函数一直执行到最后的 `cout` 语句，并输出最终计算结果。

### 前提条件和运行结果

前提条件  
运行结果

一个编写函数声明注释的好办法是将其分为两部分：前提条件（`precondition`）和运行结果（`postcondition`）。`precondition` 是函数调用的前提条件，只有前提条件成立，函数才能正常调用；`postcondition` 是函数调用执行的结果，即在前提条件成立的情况下，该函数调用的运行结果。对于带有返回值的函数，`postcondition` 描述的是函数的返回值，对于那些改变实参变量值的函数，`postcondition` 描述的是这些变量的变化情况。

#### 示例 3.7 void 函数中 return 语句的使用

```
1 #include <iostream>
2 using namespace std;

3 void iceCreamDivision(int number, double totalWeight);
4 // 输出冰淇淋分配的结果，
```

```

5 // 如果 number 为 0，输出一个错误信息。

6 int main()
7 {
8     int number;
9     double totalWeight;

10    cout << "Enter the number of customers:";
11    cin >> number;
12    cout << "Enter weight of ice cream to divide (in ounces):";
13    cin >> totalWeight;

14    iceCreamDivision(number, totalWeight);
15    return 0;
16 }
17 void iceCreamDivision(int number, double totalWeight)
18 {
19     double portion;

20     if (number == 0)
21     {
22         cout << "Cannot divide among zero customers.\n";
23         return; ← 如果 number 的值为 0，那么函数将结束执行
24     }
25     portion = totalWeight/number;
26     cout << "Each one receives "
27         << portion << " ounces of ice cream." << endl;
28 }

```

### 示例运行结果

```

Enter the number of customers: 0
Enter weight of ice cream to divide (in ounces): 12
Cannot divide among zero customers.

```

例如，下面就是一个带有 precondition 和 postcondition 的函数声明：

```

void showInterest(double balance, double rate);
// 前提条件：balance 是非负的存款余额。
// rate 是以百分数表示的利率，如 5 代表 5%。
// 运行结果：屏幕上输出相应的利息数。

```

函数 showInterest 的使用者不需要了解该函数定义的具体细节，他只要知道该函数声明中的前提条件和运行结果就可以正常使用了。

当函数的 postcondition 仅仅用来描述函数返回值时，函数注释中通常会省略 Postcondition 一词，如下所示：

```

double celsius(double fahrenheit);
// 前提条件：fahrenheit 是华氏温度。
// 返回相应的摄氏温度。

```

有些程序员在他们编写的函数声明注释中并不使用 precondition 和 postcondition 这两个词。但不管是否使用这两个词，在设计函数及编写函数声明注释的时候，应该按照 precondition 和 postcondition 这种方式进行考虑。



### main函数

我们注意到，一个程序的 main 部分其实是一个名为 main 的函数。程序运行时会自动调用 main 函数，执行函数体中的语句，而且这一过程可能会调用其他函数。虽然你可能认为 main 函数中的 return 语句可有可无，但实际上并非如此。虽然 C++ 标准规定可以省略程序中 main 部分的 return 0 语句，但许多编译器都要求必须包含 return 0，并且几乎所有的编译器都允许其存在。考虑到程序的可移植性，编写程序时应该总是在 main 部分中包含 return 0 语句。可以将程序的 main 部分看作是一个返回值类型为 int 的函数，因此需要包含一个 return 语句。将程序的 main 部分看作是一个返回整数的函数，这似乎不好理解，但许多编译器都这么处理。

虽然有些编译器允许，但在你的代码中不应该对 main 函数进行调用。只有系统才能调用 main 函数，也就是在程序运行的时候。

### 递归函数

C++ 允许定义递归函数。递归函数的相关介绍可参见第 13 章，在这之前无须了解。如果你想提前了解递归函数的相关内容，可以在学习完第 4 章后，阅读第 13 章的 13.1 节和 13.2 节。需要注意的是，main 函数不能被递归调用。

### 自测练习

15. 以下程序的输出是什么？

```
#include <iostream>
using namespace std;
void friendly();
void shy(int audienceCount);
int main()
{
    friendly();
    shy(6);
    cout << "One more time:\n";
    shy(2);
    friendly();
    cout << "End of program.\n";
    return 0;
}
void friendly()
{
    cout << "Hello\n";
}
void shy(int audienceCount)
{
    if (audienceCount < 5)
        return;
    cout << "Goodbye\n";
}
```

16. 假设示例 3.7 中函数 `iceCreamDivision` 的定义省略掉 `return` 语句, 这对程序会有什么影响? 程序可以顺利编译吗? 能正常运行吗? 程序的运行会有什么不同吗?
17. 请给出一个 `void` 函数的定义, 该函数有三个 `int` 类型的形参, 并将这三个数的乘积输出到屏幕上。编写一个完整的程序对该函数进行测试。
18. 你使用的编译器允许 `void main()` 和 `int main()` 吗? 如果采用了 `int main()` 但却没有包含 `return 0` 语句, 编译器会给出什么警告信息? 请编写一个程序进行验证。
19. 预定义函数 `sqrt` 是一个返回实参平方根的函数, 给出该函数的前提条件和运行结果。

### 3.3 作用域规则

让最终的结局合法, 让它在宪法的范围内……

约翰·马歇尔, 美国最高法院首席大法官

《麦克洛克诉马里兰州》(1819)

函数应该是一个自包含的单元, 而不应与其他函数或代码发生干涉。为了保证这一点, 函数一般都有自己的变量, 这些变量与其他任何该函数之外声明的变量都不同 (即使两者名字相同)。这些在函数定义内部声明的变量被称为局部变量, 为本小节将要介绍的内容。

#### 局部变量

回顾一下示例 3.1 中的程序, 它包含一个预定义函数 `sqrt`。程序员不需要了解 `sqrt` 函数的定义就可以正常使用它。特别是, 我们根本不需要了解 `sqrt` 函数定义中声明了什么样的变量。自定义函数也是如此。函数定义中的变量声明与预定义函数或其他程序中的变量声明是一样的。如果在一个函数定义中声明了一个变量, 紧接着又在 `main` 函数中声明了一个同名的变量, 那么这两个变量尽管名字相同, 但却是两个不同的变量。让我们来看一个例子。

示例 3.8 中的程序包含两个名为 `averagePea` 的变量: 一个在函数 `estimateOfTotal` 内部定义和使用; 另一个在程序的 `main` 函数中定义和使用。这两个是完全不同的变量。这就如同 `estimateOfTotal` 是一个预定义的函数一样。这两个名为 `averagePea` 的变量不会互相影响, 就如同它们分别在两个完全不同的程序中一样。`estimateOfTotal` 函数被调用时, 变量 `averagePea` 的赋值丝毫不会影响到 `main` 函数中同名的 `averagePea` 变量。

局部变量  
作用域

函数定义内部声明的变量被称为局部变量, 它的作用域局限于函数的范围。如果一个变量局限于某个函数, 有时简称其为局部变量, 而不特指是哪个函数的局部变量。

示例 3.5 给出了局部变量的另一个例子。程序中 `totalCost` 函数的定义开头如下所示:

```
double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; // 5% 的销售税
    double subtotal;
```

### 示例 3.8 局部变量

```
1 // 计算实验地中的豌豆产量。
2 #include <iostream>
3 using namespace std;

4 double estimateOfTotal(int minPeas, int maxPeas, int podCount);
5 // 返回豌豆总产量的估计值。
6 // 形参 podCount 表示豆荚的数量。
7 // 形参 minPeas 代表豆荚中最少的豌豆数。
8 // 形参 maxPeas 代表豆荚中最多的豌豆数。

9 int main()
10 {
11     int maxCount, minCount, podCount;
12     double averagePea, yield; ← 变量 averagePea 为 main 函
                                数的局部变量。

13     cout << "Enter minimum and maximum number of peas in a pod: ";
14     cin >> minCount >> maxCount;
15     cout << "Enter the number of pods: ";
16     cin >> podCount;
17     cout << "Enter the weight of an average pea (in ounces): ";
18     cin >> averagePea;

19     yield =
20         estimateOfTotal(minCount, maxCount, podCount) * averagePea;

21     cout.setf(ios::fixed);
22     cout.setf(ios::showpoint);
23     cout.precision(3);
24     cout << "Min number of peas per pod = " << minCount << endl
25         << "Max number of peas per pod = " << maxCount << endl
26         << "Pod count = " << podCount << endl
27         << "Average pea weight = "
28         << averagePea << " ounces" << endl
29         << "Estimated average yield = " << yield << " ounces"
30         << endl;

31     return 0;
32 }
33
34 double estimateOfTotal(int minPeas, int maxPeas, int podCount)
35 {
36     double averagePea; ← 函数 estimateOfTotal 的局部变量。

37     averagePea = (maxPeas + minPeas)/2.0;
38     return (podCount * averagePea);
39 }
```

### 示例运行结果

```
Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces
```

其中变量 `subtotal` 就是 `totalCost` 函数的局部变量，名字常量 `TAXRATE` 同样也是该函数的局部变量（名字常量就是一个已经初始化但不能改变其值的变量）。

### 局部变量

函数定义内部声明的变量被称为该函数的局部变量，或该变量的作用域为该函数内部。如果一个变量是局部变量，那么就可以在其他函数定义中声明同名的变量，它们将会是不同的变量，即使名字相同（特别指出的是，即使其中一个是 `main` 函数，也是如此）。

### 过程抽象

使用程序的人并不需要了解程序的实现细节。想象一下，如果你必须要知道并记住编译器的实现代码才能编译程序的话，那将是多么痛苦。程序总是被设计用来完成某一种任务的，比如编译程序或是检查论文中的错误拼写。要使用程序，用户必须知道程序能完成什么样的任务，但不需要了解程序如何完成该任务。函数与程序类似。编程人员在使用函数时必须知道该函数能完成什么样的工作，比如计算平方根或者将华氏温度转化为摄氏温度，但没有必要探究函数实现的细节。这种思想通常是将函数当作一个“黑箱”进行理解的。

黑箱

将某物称为“黑箱”是一个语言上的比喻，它代表一种物理装置的抽象：你知道该装置如何使用，但不知道该装置内部实现的细节，因此其内部被封闭在一个不可透视的黑箱子中。可以将一个设计良好的函数看作是一个黑箱子，程序员只知道该函数的使用方法，而对其内部的实现一无所知。程序员只需要了解的是：给该黑箱子放入正确的实参，它就可以正确地操作，并得出正确的结论。以这种黑箱思想来设计函数，称为“信息隐藏”。程序员使用函数时，只关注函数的使用方法，函数体则如同消失一般。

信息隐藏  
函数抽象

以这种黑箱思想来编写和使用函数又称为过程抽象，C++ 中则称为函数抽象。与函数相比，过程是一个语义更宽泛的词，计算机工作者用它来代指所有“类函数”的指令集合，因此他们更偏向使用过程抽象这一表述。“抽象”表达了一种思想：当以黑箱的方式使用函数时，函数的具体实现代码实际上是被抽走了。这种技巧称为黑箱原则、过程抽象原则或是信息隐藏，这三个术语表达的意思相同。怎么去称呼并不重要，重要的是要在函数的设计和定义中使用这些原则。

### 过程抽象

具体对应于函数定义时，过程抽象原则表示定义的函数要可以像黑箱那样操作。使用该函数的程序员无须了解函数定义体就可以知道如何去使用该函数，函数的声明及相关的注释应提供所有需要了解的信息。为了确保自定义函数具有这些重要性，必须严格遵守如下原则。

#### 如何去编写一个符合黑箱原则的函数

- 函数声明的注释应该告诉程序员参数所要求的所有条件，并且描述函数运行后的结果。
- 函数体内使用的所有变量都应该在函数体中声明（形参除外）。

### 全局常量和全局变量

如第1章所述，对于常量应该使用 `const` 修饰符。例如，示例 3.5 用 `const` 修饰符声明了如下的表示税率的常量：

```
const double TAXRATE = 0.05; // 5% 销售税
```

如果以上语句包含在函数的定义中，如同示例 3.5 一样，那么变量 `TAXRATE` 是一个函数的局部变量，这表示在该函数定义之外，可以用 `TAXRATE` 去命名其他常量或变量，或其他任何可命名的项。

另一方面，如果以上语句包含在程序的开头，在所有函数体之外（包括 `main` 函数），那么该常量被称为一个全局命名常量，且该常量可以用在该常量声明以后的所有函数定义中。

示例 3.9 给出了一个全局命名常量的例子。程序要求输入一个半径，然后分别计算圆的面积以及球的体积：

```
area =  $\pi \times (\text{radius})^2$ 
volume =  $(4/3) \times \pi \times (\text{radius})^3$ 
```

以上两个公式都含有常量  $\pi$ ，近似等于 3.14159。因此程序使用如下的全局命名常量：

```
const double PI=3.14159;
```

该语句出现在所有函数定义之外（包括 `main` 函数）。

编译器允许在一个很大的范围里去放置全局变量的声明。但为了代码的可读性起见，最后将所有的 `include` 指令放在一起，所有的全局命名常量放在一起，所有的函数声明（函数原型）放在一起。一般来讲，所有的全局命名常量放在 `include` 和 `using` 指令之后，函数声明之前。

将所有全局命名常量放在程序的开始位置可以使程序易于阅读，即使这些全局命名常量只被一个函数使用。如果需要修改这些命名常量，则由于它们都放在程序的开始位置，因此很容易就找到。例如，将税率常量的声明放在程序的开始位置，如果税率发生变化，程序将很容易修改。

**全局变量**      声明全局变量时可以不使用 `const` 修饰符，这样的变量被称为**全局变量**，可以被文件中所有的函数定义所使用。这与全局命名常量很相似（除了变量的声明中没有使用 `const` 修饰符外）。但全局变量很少使用，而且它会使程序变得难以理解和维护，因此应尽量避免全局变量的使用。

### 示例 3.9 全局命名常量

---

```

1  // 计算圆的面积和球的体积。
2  // 两个计算都使用同一半径。
3  #include <iostream>
4  #include <cmath>
5  using namespace std;

6  const double PI = 3.14159;

7  double area(double radius);
8  // 计算一个圆的面积。

9  double volume(double radius);
10 // 计算一个球的体积。

11 int main()
12 {
13     double radiusOfBoth, areaOfCircle, volumeOfSphere;

14     cout << "Enter a radius to use for both a circle\n"
15           << "and a sphere (in inches): ";
16     cin >> radiusOfBoth;

17     areaOfCircle = area(radiusOfBoth);
18     volumeOfSphere = volume(radiusOfBoth);

19     cout << "Radius = "<< radiusOfBoth << "inches\n"
20           << "Area of circle = "<< areaOfCircle
21           << "square inches\n"
22           << "Volume of sphere = "<< volumeOfSphere
23           << "cubic inches\n";

24     return 0;
25 }

26
27 double area(double radius)
28 {
29     return (PI * pow(radius, 2));
30 }

31 double volume(double radius)
32 {
33     return ((4.0/3.0) * PI * pow(radius, 3));
34 }

```

## 示例运行结果

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

## 自测练习题

20. 如果在函数定义中使用了一个变量，应该在什么地方声明该变量？在函数定义中？在 main 函数中？还是在其他合适的地方？
21. 假设一个名为 function1 的函数在函数体中声明了一个名为 sam 的变量，同时函数 function2 在函数体中同样也声明了名为 sam 的变量。假设程序的其他部分都没有问题，该程序能顺利编译吗？如果能顺利编译，程序能顺利运行吗？运行过程中会有错误信息提示吗？程序最终能给出正确的结果吗？
22. 函数声明中的注释有什么作用？
23. 对于函数定义而言，过程抽象的原则是什么？
24. 我们常说函数的使用者可以把函数当作黑箱来对待，怎么理解这句话？（此题的答案可以参考前一题。）

## 语句块

一个复合语句（即在一对大括号中的语句）中定义的变量，其作用域限定于该复合语句中。在该复合语句之外，该变量的名字可以用于其他项，比如另外一个变量的名字。

## 语句块

含有声明的复合语句通常被称为语句块，事实上，语句块和复合语句是同一个事物的两个名字。但是，当我们的重点在复合语句中的声明时，一般使用语句块这一术语，而且我们称这些语句块中声明的变量限定于这个语句块。

如果一个变量是在语句块中声明的，那么该变量的作用范围为从声明处的语句到语句块的结尾处。这通常表达为：该声明的作用域为从声明到语句块的结尾处。因此，如果一个变量在语句块的开始处声明，那么它的作用域就是整个语句块。如果变量的声明在语句块的中间，那么，在程序执行到声明语句所在位置之前，该变量的声明是不会发生作用的。

注意，函数定义中的函数体就是一个语句块。因此，函数定义中的局部变量和语句块中的局部变量是相同的。

## 语句块

语句块是包含在一对大括号中的 C++ 代码。语句块中声明的变量局限于语句块的范围，因此该变量名可以在该语句块之外用作其他项的名字（如另外一个变量的名字）。

## 嵌套作用域

假设一个语句块嵌套在另一个语句块中，并且同一个标识符在这两个语句块中都被声明为变量。此时，虽然两个变量同名，却是两个不同的变量。一个变量只能在内层语句块中使用，在内层语句块之外不能使用；另一变量只存在于外层语句块中，不可在内层语句块中使用。这两个变量完全独立，互不影响，因此对一个变量的改动不会影响到另一个变量。

### 嵌套语句块的作用域规则

如果一个标识符在两个语句块中都被声明为变量，那么这两个变量虽同名但实质不同。一个变量只存在于内层语句块中，不能在内层语句块之外的地方使用；另一个变量只存在于外层语句块中，不可在内层语句块中使用。这两个变量是独立的，因此，对其中一个变量的修改不会影响到另外一个变量。

### 提示：在分支和循环语句中使用函数调用

switch 语句和 if-else 语句允许在每个分支中使用不同的语句。但这样会导致 switch 语句和 if-else 语句很难理解。与在分支语句中放置一个复合语句相比，更好的做法是将该复合语句转化为函数定义，在分支语句中放置函数调用。与此类似，如果循环体过大，可以将复合语句转化为函数定义，使函数体成为函数调用。■

### for 循环体中的变量声明

可以在 for 语句的头部声明变量，这样就可以在声明的同时进行相应的初始化。例如：

```
for (int n = 1; n <= 10; n++)
    sum = sum + n;
```

ANSI/ISO C++ 标准要求，所有符合标准的编译器在处理 for 循环中的初始化声明时，这些声明将被视为在该循环体内部。早期的 C++ 编译器并非如此，程序员自己可以决定编译器如何处理 for 循环初始值中声明的变量。考虑到移植性方面的原因，最好不要这样编写代码。幸运的是，几乎所有常用的 C++ 编译器都遵从这条规则，但不久以后的编译器有可能遵从，也有可能不遵从这条规则。

### 自测练习

25. 虽然我们不建议使用这种风格来实际编写程序，但可以用这种含有嵌套语句块的练习来帮助你理解作用域的规则。如果以下代码位于一个完整且正确的程序中，程序的输出是什么？

```
{
    int x = 1;
```



```

cout << x << endl;
{
    cout << x << endl;
    int x = 2;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 3;
        cout << x << endl;
    }
    cout << x << endl;
}
cout << x << endl;
}

```

## 本章小结

- C++ 中有两种函数：带有返回值的函数和 void 函数。
- 函数应该合理定义，从而可以当作黑箱来使用。使用函数的程序员无须知道函数具体是如何编码的。程序员只需要了解函数声明及相应的描述返回值的注释。该原则通常被称为过程抽象原则。
- 编写函数声明注释的一种方法是使用前提条件和运行结果。前提条件描述的是函数调用时应满足的先决条件。运行结果描述的是函数调用的效果；也就是在满足前提条件的情况下，函数调用后将要返回什么样的值。
- 函数定义中声明的变量被称为该函数的局部变量。
- 形参是一种占位符，函数调用时，实参将填充形参。具体的“填充”过程将在第4章中进行介绍。

## 自测练习题答案

- 4.0 4.0 8.0
  - 8.0 8.0 1.21
  - 3 3 0
  - 3.0 3.5 3.5
  - 6.0 6.0 5.0
  - 5.0 4.5 4.5
  - 3 3.0 3.0
- $\sqrt{x+y}$
  - $\text{pow}(x, y + 7)$
  - $\sqrt{\text{area} + \text{fudge}}$
  - $\sqrt{\text{time} + \text{tide}} / \text{nobody}$
  - $(-b \pm \sqrt{b^2 - 4*a*c}) / (2*a)$

```

f. abs(x - y) or labs(x - y) or fabs(x - y)
3. #include <iostream>
   #include <cmath>
   using namespace std;
   int main( )
   {
       int i;
       for (i = 1; i <= 10; i++)
           cout << "The square root of " << i
               << " is " << sqrt(i) << endl;
       return 0;
   }

```

4. 实参是传递给操作系统的。对于 C++ 程序而言，你可以将任何 int 值用作实参。但习惯上，1 代表因错误而导致的 exit 调用，0 则代表其他情况。

5. (5 + (rand( ) % 6))

```

6. #include <iostream>
   #include <cstdlib>
   using namespace std;

   int main( )
   {
       cout << "Enter a nonnegative integer to use as the\n"
           << "seed for the random number generator: ";
       unsigned int seed;
       cin >> seed;
       srand(seed);

       cout << "Here are ten random probabilities:\n";
       int i;
       for (i = 0; i < 10; i++)
           cout << ((RAND_MAX - rand( ))/static_cast<double>(RAND_MAX))
               << endl;
       return 0;
   }

```

7. Wow

8. 函数声明为：

```

int sum(int n1, int n2, int n3);
// 返回 n1、n2 和 n3 之和。

```

函数定义为：

```

int sum(int n1, int n2, int n3)
{
    return (n1 + n2 + n3);
}

```

9. 函数声明为：

```

char positiveTest(double number);

```

// 如果 number 为正数, 返回 'P' , 否则返回 'N'。

函数定义为：

```
char positiveTest(double number)
{
    if (number > 0)
        return 'P';
    else
        return 'N';
}
```

10. 不可以。函数定义不能包含在另一个函数定义中。

11. 预定义函数与自定义函数的调用方法相同。

```
12. bool inOrder(int n1, int n2, int n3)
{
    return ((n1 <= n2) && (n2 <= n3));
}
```

```
13. bool even(int n)
{
    return ((n % 2) == 0);
}
```

```
14. bool isDigit(char ch)
{
    return ( '0' <= ch) && (ch <= '9');
}
```

```
15. Hello
    Goodbye
    One more time:
    Hello
    End of program.
```

16. 如果省略示例 3.7 中函数 iceCreamDivision 定义中的 return 语句, 程序可以正常编译和运行。但如果顾客的人数为 0, 那么由于除 0 操作, 程序将会发生运行时错误。

```
17. #include <iostream>
using namespace std;
void productOut(int n1, int n2, int n3);

int main( )
{
    int num1, num2, num3;
    cout << "Enter three integers: ";
    cin >> num1 >> num2 >> num3;
    productOut(num1, num2, num3);
    return 0;
}

void productOut(int n1, int n2, int n3)
{
    cout << "The product of the three numbers "
        << n1 << ", " << n2 << ", and "
```

```

    << n3 << " is " << (n1*n2*n3) << endl;
}

```

18. 答案取决于具体的系统。

19. `double sqrt(double n);`

// 前提条件:  $n \geq 0$

// 运行结果: 返回  $n$  的平方根

如果喜欢, 可以将 `sqrt` 函数的声明注释改写为如下的形式。但对于带返回值的函数, 其声明注释推荐采用以上形式。

// 运行结果: 返回  $n$  的平方根

20. 如果函数定义中使用了一个变量, 那么应该在该函数的定义中对该变量进行声明。
21. 一切都会正常进行。程序可以正常编译、运行。运行时也不会出现错误提示信息, 且最后将输出正确结果。
22. 注释描述了函数将会进行什么样的操作, 包括函数的返回值以及其他为了使用该函数需要了解的信息。
23. 过程抽象原则要求可以将函数看作一个黑箱进行操作。这意味着, 使用该函数的程序员无须了解函数具体是怎么实现的, 函数声明以及相关的注释已经给出了使用该函数所需的一切信息。
24. 我们常说可以将某个函数看作是一个黑箱来使用, 这表明: 使用该函数的程序员无须了解函数的具体实现, 函数声明和相关的注释信息已经给出了使用该函数所需的一切信息。
25. 将下面的代码片段做少许修改就可以很容易地理解变量声明和使用的对应关系。这段代码包含三个名为  $x$  的不同变量, 下面我们将它们重新命名为  $x1$ 、 $x2$  和  $x3$ , 注释中给出了相应的输出。

```

{
    int x1 = 1; // output in this column
    cout << x1 << endl; // 1<new line>
    {
        cout << x1 << endl; // 1<new line>
        int x2 = 2;
        cout << x2 << endl; // 2<new line>
        {
            cout << x2 << endl; // 2<new line>
            int x3 = 3;
            cout << x3 << endl; // 3<new line>
        }
        cout << x2 << endl; // 2<new line>
    }
    cout << x1 << endl; // 1<new line>
}

```

## 编程练习

1. 升等于 0.264179 加仑。编写一个程序，读入用户购买的汽油升数以及汽车行驶过的英里数，输出汽车每加仑汽油能跑多少英里。程序可根据用户的需要多次计算，程序应采用函数来计算每加仑汽油跑的英里数，并且采用一个全局常量记录每加仑等于的升数。
2. 编写一个程序来估算去年的通货膨胀率。程序读入某物品去年和今年的同期价格，通过价格差除以去年同期价格来计算通货膨胀率。程序可根据用户的需要多次计算。定义一个函数来计算通货膨胀率，通货膨胀率为 double 类型的值，以百分率表示。例如，5.3 表示 5.3%。
3. 对上一题编写的程序进行扩充，使其能够计算该物品自计算之日起一年或者两年之后的价格。每年增加的钱数是通货膨胀率乘以该年年初时的价格进行计算的。定义一个函数来计算一定年份里该物品的价格，该函数的参数分别为现在的价格和当前的通货膨胀率。
4. 相距为  $d$ ，质量分别为  $m_1$  和  $m_2$  的物体之间的万有引力为：

$$F = \frac{Gm_1m_2}{d^2}$$

其中  $G$  为万有引力常数：

$$G = 6.673 \times 10^{-8} \text{ cm}^3 / (\text{g} \cdot \text{s}^2)$$

编写一个函数，计算两物体之间的万有引力，该函数的参数为：两物体的质量及两者之间的距离，返回它们之间万有引力的大小。由于采用上面的公式进行计算，那么万有引力的单位就是达因（1 达因等于  $1\text{g} \cdot \text{cm}/\text{s}^2$ ）。程序应采用一个全局名字常量来表示万有引力常数。将该函数添加到一个完整的程序中，输入相关的参数后计算它们之间的万有引力。程序可根据用户的需要多次进行计算。

5. 编写一个程序，输入用户的身高、体重及年龄，然后根据如下的规则计算用户的服装尺寸：
  - 帽子尺寸 = 体重（磅）除以身高（英寸），再乘以 2.9。
  - 上衣尺寸（英寸）= 身高乘以体重除以 288。30 岁以上每增加 10 岁增加 1/8 英寸（注意，该调整以 10 岁为单位，例如 30 岁到 39 岁没有任何调整。对于 40 岁，则增加 1/8 英寸）。
  - 腰围（英寸）= 体重除以 5.7。28 岁以上每增加 2 岁增加 1/10 英寸（注意，该调整以 2 岁为单位，例如 29 岁没有任何调整，对于 30 岁则增加 1/10 英寸）。

编写函数完成以上计算，程序可以根据用户的需要多次进行计算。

6. 编写一个函数，计算四个得分的平均值与标准差。标准差的定义是四个得分值的平均数的平方根： $(s_i - a)^2$ ，其中  $a$  是得分  $s_1$ 、 $s_2$ 、 $s_3$ 、 $s_4$  的平均值。函数包含六个参数，并且会调用其他两个函数。将该函数放到一个具体的程序中进行不断测试，直到用户告诉程序结束为止。
7. 寒冷的天气里，气象学家会计算一个名为风寒因子的指标，该指标考虑了风速及温度。该指标给出了一个定量的方式来评估一定温度下风对天气寒冷程度的影响。具体公式如下：

$$w = 33 - \frac{(10\sqrt{v} - v + 10.5)(33 - t)}{23.1}$$

其中  $v$  代表风速，单位为 m/s。

$t$  代表气温，且  $t \leq 10$ 。

$w$  为风寒因子（摄氏度）。

编写一个计算风寒因子的函数，你的程序应对输入温度的范围进行检查。从图书馆的报纸上查找相关的天气报道，比较其给出的风寒因子和程序计算出来的风寒因子。

8. 编写一个程序，输出诗歌 “Ninety-Nine Bottles of Beer on the Wall”，程序应该用英文而不是数字输出瓶子数：

Ninety-nine bottles of beer on the wall,

Ninety-nine bottles of beer,

Take one down, pass it around,

Ninety-eight bottles of beer on the wall.

...

One bottle of beer on the wall,

One bottle of beer,

Take one down, pass it around,

Zero bottles of beer on the wall.

程序不可使用 99 个不同的输出语句来完成该题目。

9. 双骰子游戏是一种掷两个骰子的赌博游戏，游戏规则如下：如果第一掷掷出 7 或者 11 即刻获胜，如果第一掷掷出 2、3 或者 12 即输。若第一掷掷出其他任何一个点数，即 4、5、6、8、9 或 10，玩家则需继续投掷骰子，直到掷出同样的点数或者 7 点。若首先掷出同样的点数，则玩家获胜；若首先掷出 7 点，则玩家失败。

编写一个程序根据以上的规则来玩双骰子游戏。程序不要求下赌注，但应能计算玩家是否会输或者赢。创建一个函数模拟掷两个骰子并返回它们的点数和。

编写一个循环使程序玩 10 000 次游戏，增加计数器统计玩家输赢的次数。在 10 000 次游戏结束以后，以  $(Wins)/(Wins+Losses)$  计算赢的概率，并输出。

10. 估算孩子身高的一个方法是采用如下的公式，公式中用到了父母的身高：

$$H_{\text{male\_child}} = ((H_{\text{mother}} \cdot 13/12) + H_{\text{father}})/2$$

$$H_{\text{female\_child}} = ((H_{\text{father}} \cdot 12/13) + H_{\text{mother}})/2$$

其中，身高均为英寸。编写一个函数，输入参数为孩子的性别、母亲的身高、父亲的身高，输出为孩子的估计身高。将函数放入一个实际的程序中进行不断测试，直到用户让程序结束为止。用户应输入以英尺和英寸为单位的身高，程序输出孩子的估计身高（以英尺和英寸为单位），使用整型变量存储身高。

11. 有一种名叫“pig”的二人掷色子游戏，规则很简单，谁先到 100 点谁就胜出，采用回合制。每一轮一名选手先掷色子。

- 如果掷出 2 ~ 6，那么有两种选择：

(a) 重新掷一次。

(b) 结算本轮的点数，本轮所掷的所有点数将加到选手的总点数中，然后轮换另一个选手。

- 如果选手掷出 1，那么本次点数作废，且轮换另一个选手。

编写程序模拟该游戏，进行人机对战。选手输入“r”表示重新掷色子，输入“h”表示结算本轮点数。

电脑按照如下规则玩游戏：电脑不断重新掷色子，直到本轮的点数达到 20 或更多，然后结算本轮点数。当然，如果电脑取胜或者掷出点数 1，那么立即轮换回合。人机对战中，选手先掷。

程序应该包含如下的函数：

```
int humanTurn(int humanTotalScore);
int computeTurn(int computeTotalScore);
```

以上两个函数分别负责电脑以及选手每轮点数的计算，函数的参数分别为电脑和选手的总点数，函数的返回值为本轮的总点数。例如，如果选手两次分别掷出点数 3 和 6，那么函数 humanTurn 应返回 9；但如果选手三次掷出点数 3、6 和 1，那么该函数将返回 0。

12. 编写程序，输入一个具体的日期（如 2008 年 7 月 4 日），输出该日期对应的星期。采用的算法参见 [http://en.wikipedia.org/wiki/Calculating\\_the\\_day\\_of\\_the\\_week](http://en.wikipedia.org/wiki/Calculating_the_day_of_the_week)。程序的实现会用到如下的函数：

```
bool isLeapYear(int year);
```

该函数判断是否是闰年，判断闰年的伪代码如下：

`leap_year=((year divisible by 400) or (year divisible by 4 and year not divisible by 100))`

`int getCenturyValue(int year);`

该函数将年份的头两位数字除以 4，然后从 3 中减去该余数，得到的差乘以 2 并返回。例如，对于 2008 年， $(20/4)=5$ ，余数为 0。 $3-0=3$ ，返回  $3*2=6$ 。

`int getYearValue(int year);`

该函数根据对应的年份计算出一个值，首先取出年份的后两位数字，例如对于 2008，该数字为 08；然后用 4 除上一步得到的结果，舍弃余数。函数最终返回这两个数字的和。例如，对于年份 2008，首先我们得到 08，然后  $(8/4)=2$ ，返回  $2+8=10$ 。

`int getMonthValue(int month, int year);`

该函数根据下表返回相应的值，实现中会用到函数 `isLeapYear`。

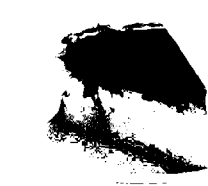
月份	返回值
January	0 (闰年返回 6)
February	3 (闰年返回 2)
March	3
April	6
May	1
June	4
July	6
August	2
September	5
October	0
November	3
December	5

最后，将日期的天数加上函数 `getMonthValue`、`getYearValue` 及 `getCenturyValue` 的返回值，然后结果除以 7，得到最终的余数。其中余数 0 对应星期天，余数 1 对应星期一，依次类推。例如，日期 2008 年 7 月 4 号，对应的星期为： $(\text{日期的天数}) + (\text{getMonthValue}) + (\text{getYearValue}) + (\text{getCenturyValue}) = 4 + 6 + 10 + 6 = 26$ ；然后  $26 / 7 = 3$ ，且余数为 5。也就是说，2008 年 7 月 4 号那天是星期五。



注意，程序应能正确处理各种合法日期，且最终结果以英文形式输出。

13. 现在有 4 个相同的奖项和 25 名进入决赛的选手，选手编号分别为 1 ~ 25。编写程序随机选择 4 名获奖选手，但需注意同一名选手不能被选中两次。例如，3、15、22 和 14 为一个有效的选择结果，但 3、3、31 和 17 就是一个无效的结果。
14. 第 2 章的编程练习 9 要求实现巴比伦算法来计算一个数字的平方根。编写一个完整的程序来测试该算法，程序的输入和输出类型都为 `double` 类型。



## 函数与重载

# 4



### 4.1 参数 118

传值参数 118

初识引用参数 120

引用传递调用机制详解 122

常量引用参数 124

示例：函数swapValues 124

提示：一种操作，而不是代码 125

混合参数列表 126

提示：使用何种参数 126

陷阱：无意的局部变量 128

提示：选择形参的名字 129

示例：买比萨 129

### 4.2 重载与默认实参 132

重载简介 132

陷阱：自动类型转换与重载 134

重载解析的规则 135

示例：改写“买比萨程序” 136

默认参数 138

### 4.3 函数的测试和调试 140

assert宏 140

占位程序和驱动程序 141

# 第4章 函数与重载

只是填补空白。  
常规说明

## 概述

本章详细介绍 C++ 函数调用中实参如何传递给形参的内部实现，还将介绍重载机制。重载是一种函数名相同但函数体不同的函数定义方法。最后，本章还将给出一些函数测试的小技巧。

## 4.1 参数

你不能将一个方形的栓放在一个圆形的洞里。  
一句俗语

值传递  
引用传递

本节详细介绍 C++ 函数调用时，实参传递给形参的具体操作方法。C++ 有两种类型的参数，因此也有两种基本的传递机制，分别为值传递和引用传递。本书之前所出现的所有函数调用都是值传递。对于值传递，仅仅传递了实参的值，而对于引用传递，实参是一个变量，是变量本身被传递过去，因此函数的调用可以改变变量的值。引用传递是通过在形参类型后边加一个“&”符号来表示的，如下所示：

```
void getInput(double& variableOne, int& variableTwo);
```

而值传递是没有“&”符号的，接下来的几节将详细介绍这两种类型的参数。

### 传值参数

传值参数不仅仅是由函数的实参进行填充的占位符，它实际上是一个局部变量。函数调用时，首先计算出实参的值，然后该值成为传值参数（实际上就是一个局部变量）的初始值。

大多数情况下，可以将传值参数看作在函数调用过程中被相应实参填充的一个空白，或者占位符。但在某些情况下，可以将传值参数当作一个局部变量，并且可以在函数体中对其做不同的修改。例如，在示例 4.1 程序中，有一个用作局部变量的传值参数，并且其值在函数体中进行了修改。注意，函数 fee 的形参 minutesWorked，在函数的定义中被当作一个局部变量，并且通过以下的语句对其值做了修改：

```
minutesWorked = hoursWorked*60 + minutesWorked;
```

### 示例 4.1 用作局部变量的传值参数

```
1 // 法律事务所记账程序。  
2 #include <iostream>
```

```

3 using namespace std;

4 const double RATE = 150.00; // 每15分钟的美元数。

5 double fee(int hoursWorked, int minutesWorked);
6 // 返回法律服务的费用。
7 // (即 hoursWorked 小时数和 minutesWorked 分钟数)。

8 int main()
9 {
10     int hours, minutes;
11     double bill;

12     cout << "Welcome to the law office of\n"
13           << "Dewey, Cheatham, and Howe.\n"
14           << "The law office with a heart.\n"
15           << "Enter the hours and minutes"
16           << "of your consultation:\n";
17     cin >> hours >> minutes;

18     bill = fee(hours, minutes);
19
20     cout.setf(ios::fixed);
21     cout.setf(ios::showpoint);
22     cout.precision(2);
23     cout << "For " << hours << " hours and " << minutes
24           << " minutes, your bill is $" << bill << endl;

25     return 0;
26 }
27 double fee(int hoursWorked, int minutesWorked)
28 {
29     int quarterHours;
30     minutesWorked = hoursWorked*60 + minutesWorked;
31     quarterHours = minutesWorked/15;
32     return (quarterHours*RATE);
33 }

```

函数 fee 的调用并不会改变 minutes 的值。

minutesWorked 是一个局部变量。

### 示例运行结果

```

Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00

```

作为局部变量使用的传值参数与其他在函数体中单独声明的变量的用法一致，只是不需要进行单独的声明。在函数头中列出了形参 minutesWorked，就可以视作变量的声明了。下面这种定义 fee 函数的方法是错误的，因为它对参数 minutesWorked 声明了两次。

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;
    int minutesWorked;
    ...
}
```

当 minutesWorked 为函数的形参时，不能再声明同名的局部变量。

## 自测练习

1. 详细描述传值参数的原理。
2. 下面函数的参数分别是以英尺及英寸表示的长度，最终返回的是总的英寸数。例如，totalInches(1,2) 返回 14，因为 1 英尺 2 英寸等于 14 英寸。该函数能正常运行吗？如果不能，给出原因。

```
double totalInches(int feet, int inches)
{
    inches = 12*feet + inches;
    return inches;
}
```

## 初识引用参数

到目前为止，我们一直使用的值传递调用并不能满足所有任务的需求。例如，函数经常要做的一件事情就是：接受用户的输入值，并将该值赋给一个实参变量。到目前为止我们使用的是值传递调用参数，函数调用中的对应实参可以是一个实参，但函数只能使用该实参的值，而不能以任何方式对该实参做出改变。对于值传递调用，仅仅是用实参的值对形参做了一个简单的替换。而对于一个输入函数，我们所希望的是用实参变量（而不是值）来替代形参。引用传递调用就是用来解决这类问题的，对于引用传递调用，函数调用中的相应实参必须是一个变量，并且这个实参将代替函数的形参。就好像是将实参变量复制到函数体中，取代了形参。实参被代入后，通过函数体代码的执行，就可以改变实参变量的值。

& 符号

引用传递调用参数必须以某种方式标记起来，这样就可以与值传递调用参数区别开来。C++ 是通过在形参列表的类型名称之后加上一个符号“&”来实现的。在函数声明和函数定义的函数头中都应该如此。例如，在下面函数的定义中，形参 receiver 就是一个引用传递调用参数：

```
void getInput(double& receiver)
{
    cout << "Enter input number:\n";
    cin >> receiver;
}
```

在包含该函数定义的程序中，下面的函数调用将把 double 类型的变量 inputNumber 设为从键盘读入的值：

```
getInput(inputNumber);
```

C++ 允许将“&”符号与类型名称放在一起，也可以与参数名字放在一起，因此有时候会看到如下的表达方式：

```
void getInput(double &receiver);
```

这等价于：

```
void getInput(double& receiver);
```

示例 4.2 演示了引用传递调用的使用，程序读入两个数字，并最终以相反的顺序输出这两个数。

函数 `getNumbers` 和 `swapValues` 中的参数都是引用传递调用参数。输入由以下函数调用实现：

```
getNumbers(firstNum, secondNum);
```

变量 `firstNum` 和 `secondNum` 的值由以上的函数调用来设定。然后通过如下的函数调用，完成变量顺序的颠倒：

```
swapValues(firstNum, secondNum);
```

## 示例 4.2 引用传递调用参数

---

```
1 // 演示引用传递调用参数的程序。
2 #include <iostream>
3 using namespace std;

4 void getNumbers(int& input1, int& input2);
5 // 从键盘读入两个整数。

6 void swapValues(int& variable1, int& variable2);
7 // 交换两个变量的值。

8 void showResults(int output1, int output2);
9 // 显示变量 output1 和 output2 的值。

10 int main()
11 {
12     int firstNum, secondNum;

13     getNumbers(firstNum, secondNum);
14     swapValues(firstNum, secondNum);
15     showResults(firstNum, secondNum);
16     return 0;
17 }

18 void getNumbers(int& input1, int& input2)
19 {
20     cout << "Enter two integers:" ;
21     cin >> input1
22         >> input2;
23 }

24 void swapValues(int& variable1, int& variable2)
25 {
26     int temp;
```

```

27     temp = variable1;
28     variable1 = variable2;
29     variable2 = temp;
30 }
31
32 void showResults(int output1, int output2)
33 {
34     cout << "In reverse order the numbers are:"
35         << output1 << " " << output2 << endl;
36 }

```

#### 示例运行结果

```

Enter two integers: 5 6
In reverse order the numbers are: 6 5

```

下面几个小节将进一步介绍引用传递调用的细节，并解释示例 4.2 中所使用的一些特殊函数。

#### 引用传递调用参数

为了使一个形参成为引用传递调用参数，只需在形参的类型名称之后添加“&”符号。此后函数调用时，相应的实参就应该是一个变量，而不是常量或者其他表达式了。函数调用时，相应的实参变量（而非其值）将替代对应的形参，函数体中任何对形参的改动都将反映到实参变量上。本章的后续章节将详细介绍替代机制的具体内容。

##### 示例

```
void getData(int& firstInput, double& secondInput);
```

#### 引用传递调用机制详解

大多数情况下，引用传递调用机制就如同使用函数实参名完全替代引用传递调用的形参。但是具体过程可能比这还微妙。在某些情况下，这种微妙的细节是很重要的，因此我们需要了解引用传递调用过程的更多细节。

**地址** 程序变量都是放在内存单元中的，每个内存单元都有自己独一无二的一个地址。编译器为每个变量都分配了一个内存单元。例如，示例 4.2 中的程序编译完成之后，变量 firstNum 可能会被分配到地址为 101 的内存单元，变量 secondNum 则有可能被分配到地址为 1012 的内存单元。简单来看，可以将这些内存单元看成不同的变量。例如，考虑示例 4.2 中的如下函数声明：

```
void getNumbers(int& input1, int& input2);
```

引用传递调用形参 input1 和 input2 是函数调用中实参的占位符。

再考虑如下的函数调用：

```
getNumbers(firstNum, secondNum);
```

该函数调用执行时，传递给函数的并不是实参名 `firstNum` 和 `secondNum`，而是与各个名字对应的内存单元列表。本例中，该内存列表包括：

```
1010
1012
```

这也就是实参 `firstNum` 和 `secondNum` 对应的内存单元。也正是这些内存单元最终与函数形参一一对应：第一个内存单元与第一个形参对应，第二个内存单元与第二个形参对应，依此类推。

换一种方式来看，本例中的对应关系如下：

```
firstNum → 1010 → input1
secondNum → 1012 → input2
```

执行函数语句时，不管函数体对形参执行什么操作，实际上都是对相应的内存单元进行操作。本例中，`getNumbers` 函数要求使用 `cin` 语句将用户输入的值存放在形参 `input1` 中，其实也就相当于将该值存储在地址为 1010 的内存单元中（也就是变量 `firstNum`）。同样，函数还要求将用户输入的第二个值通过 `cin` 语句存放在形参变量 `input2` 中，即存放在地址为 1012 的内存单元中（变量 `secondNum`）。因此，无论函数体对形参变量 `input1` 和 `input2` 进行了什么操作，本质上都是对变量 `firstNum` 和 `secondNum` 进行操作。

这看起来似乎很不直接，或者说至少有一层多余的环节。如果变量 `firstNum` 就是内存单元 1010 中的变量，为什么我们不简单地称为“`firstNum`”，而非要说“内存单元 1010 中的变量”呢？事实上，如果实参和形参的名字恰好相同，在这种易混淆的情况下，这层看似多余的环节就很有用了。例如，函数 `getNumbers` 包含形参 `input1` 和 `input2`，假定我们需要对示例 4.2 中的程序做些修改，使函数 `getNumbers` 实参的名字也分别为 `input1` 和 `input2`。假设我们要进行一些比较复杂的操作，如将用户输入的第二个值存放在 `input2` 中，而将用户输入的第二个值存放在 `input1` 中——这也是因为第二个数需要首先处理，或者它比较重要的原因。现在我们假设程序的 `main` 函数中声明了变量 `input1` 和 `input2`，它们对应的内存单元地址分别为 1014 和 1016。此时函数调用如下所示：

```
int input1, input2;
getNumbers(input2, input1);
```

← 请注意实参的顺序。

这种情况下，如果简单地说“`input1`”，那么其他人根本不清楚指的是 `main` 函数中声明的 `input1`，还是形参 `input1`。但如果 `main` 函数中声明的变量 `input1` 分配的内存单元为 1014，那么“内存单元 1014 中的变量”就很明确了，没有任何歧义。接下来，我们详细地看看本例中替换机制的各个环节。

在该函数调用中，对应形参 `input1` 的实参是变量 `input2`，对应形参 `input2` 的实参是变量 `input1`。对我们而言，这可能容易混淆，但对计算机来说这一点问题都没有。因为计算机根本就不会理解“用 `input1` 替换 `input2`”或者“用 `input2` 替换 `input1`”。计算机只处理内存单元，计算机用“内存单元 1016 中的变量”代替形参 `input1`，用“内存单元 1014 中的变量”代替形参 `input2`。



## 常量引用参数

本小节的内容在这里以备参考。如果你是按照先后顺序阅读本书的话，可以跳过此节。本书的后续章节会有更详细的介绍。

如果在引用参数的类型前面加上 `const` 修饰符，就可以得到一个不可更改的引用参数。对于目前我们碰到的类型来讲，这种用法没有任何优势。但是，对于类和数组类型的参数，这将有助于提高处理效率。在我们讲到数组及类的内容时会再对这部分做详细介绍。

### 示例：函数 `swapValues`

示例 4.2 中定义的函数 `swapValues` 交换两个变量的值。该函数的声明及相应的注释如下：

```
void swapValues(int& variable1, int& variable2);
// 交换变量 variable1 和 variable2 的值。
```

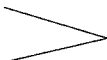
为了解该函数的工作原理，假设变量 `firstNum` 的值为 5，变量 `secondNum` 的值为 6，并考虑如下的调用方式：

```
swapValues(firstNum, secondNum);
```

该函数调用完成后，变量 `firstNum` 的值将为 6，而变量 `secondNum` 的值将为 5。

正如示例 4.2 所示，函数 `swapValues` 的定义使用了一个局部变量 `temp`。这个局部变量是必需的，可能有些人会认为该函数可以简化为：

```
void swapValues(int& variable1, int& variable2)
{
    variable1 = variable2;
    variable2 = variable1;
}
```



该函数的实现是错误的！

为了搞清楚该定义为什么是错误的，先来看看该函数调用后会产生什么结果：

```
swapValues(firstNum, secondNum);
```

变量 `firstNum` 和 `secondNum` 将代替形参 `variable1` 和 `variable2`，因此基于这个不正确的函数定义，此调用等价于：

```
firstNum = secondNum;
secondNum = firstNum;
```

这段代码不能得到预期的结果，变量 `firstNum` 的值将等于 `secondNum` 的值，但 `secondNum` 则等于已经改变了的 `firstNum` 的值，也就是 `secondNum` 最开始的值。因此，`secondNum` 的值根本就没有改变。程序需要做的是，在给 `firstNum` 赋值之前将原来的值保存起来，这也正是局部变量 `temp` 的用途。示例 4.2 中给出的函数定义是正确的，使用该正确定义的函数，并且函数对应的实参分别为 `firstNum` 和 `secondNum`，函数调用等价于如下的代码：

```
temp = firstNum;
```

```
firstNum = secondNum;
secondNum = temp;
```

### 提示：一种操作，而不是代码

尽管可以将函数调用理解为一种代码的替换，但这不是我们通常看待函数调用时所应采取的方式。应该将函数调用视为一个操作。例如，考虑示例 4.2 中的函数 `swapValues` 以及它的一个调用：

```
swapValues(firstNum, secondNum);
```

把该调用看成是两个实参交换值的操作，比将该函数调用看作如下的代码要简单明了得多：

```
temp = firstNum;
firstNum = secondNum;
secondNum = temp; ■
```

### 自测练习

#### 3. 下面程序的输出是什么？

```
#include <iostream>
using namespace std;
void figureMeOut(int& x, int y, int& z);

int main()
{
    int a, b, c;
    a = 10;
    b = 20;
    c = 30;
    figureMeOut(a, b, c);
    cout << a << " " << b << " " << c << endl;
    return 0;
}

void figureMeOut(int& x, int y, int& z)
{
    cout << x << " " << y << " " << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << " " << y << " " << z << endl;
}
```

- 示例 4.2 中，如果省略了函数 `swapValues` 声明以及定义中第一个参数中的“&”符号，但保留第二个参数中的“&”符号，该程序的输出将是什么？假设用户输入的值如示例 4.2 所示。
- 定义一个名为 `zeroBoth` 的 `void` 函数，该函数包含两个类型为 `int` 的引用传递参数，

函数运行的结果是将这两者的值都设为 0。

6. 定义一个名为 `addTax` 的 `void` 函数，该函数包含两个形参：`taxRate`，是以百分比表示的营业税；`cost`，是一个物品的税前价格。该函数计算该物品的营业税，并将其记入 `cost` 中。

## 混合参数列表

一个形参是值传递调用参数还是引用传递调用参数，这取决于参数声明的类型后面是否包含“&”符号。如果有“&”符号，这个形参就是引用传递调用参数，否则就是值传递调用参数。

值传递调用  
参数和引用  
传递调用参  
数混合使用

同一个函数中可以同时包含值传递调用参数和引用传递调用参数。例如，下面的函数声明中，第一个和最后一个形参是引用传递调用参数，中间的是值传递调用参数：

```
void goodStuff(int& par1, int par2, double& par3);
```

引用传递调用参数不只局限于 `void` 函数，也可以在有返回值的函数中使用。因此，包含引用传递调用参数的函数可以在改变实参的同时返回一个值。

## 形参和实参

所有与形参和实参有关的不同术语都容易让人混淆。但是，如果记住如下几点，就很容易掌握它们：

1. 函数的形参列在函数的声明中，并在函数定义的函数体中使用。当函数调用发生时，形参可以被看作是一类将被填充的空白或者占位符。
2. 实参是用来填充形参的。当编写函数调用语句时，实参列在函数名之后的括号内，函数调用执行时，实参被传递给对应的形参。
3. 值传递调用和引用传递调用指的是参数传递过程中的一种机制。在值传递调用中，只使用了实参的值，在此过程中，形参是一个局部变量，它被初始化为相应实参的值。在引用传递调用中，则是用形参变量代替了实参变量，因此任何发生在形参上的修改都会反映到对应的实参变量中。

## 提示：使用何种参数

示例 4.3 解释了编译器对待值传递调用和引用传递调用的不同之处。参数 `par1Value` 和参数 `par2Ref` 都是在函数体内被赋值的。但由于它们是不同的参数，所以效果完全不同。

`par1Value` 是值传递调用参数，因此它是函数内的一个局部变量，当函数调用发生时：

```
doStuff(n1, n2);
```

局部变量 `par1Value` 被初始化为 `n1`。也就是，局部变量 `par1Value` 被初始化为 1，同时实参 `n1` 就被程序所忽略。正如程序运行结果所示，形参 `par1Value` 在函数体中

被设为 111，并输出到屏幕上。但是，变量 n1 的值没有发生变化，如同程序运行的结果所示，n1 的值仍然为 1。

另一方面，par2Ref 为一个引用传递调用参数，函数调用时，实参 n2 代替了形参 par2Ref。因此当下面的这句代码运行时：

```
par2Ref = 222;
```

等价于：

```
n2 = 222;
```

因此，函数体中的代码导致了变量 n2 的值发生了变化。正如程序的运行结果所示，n2 的值由 2 变为 222。

如果你能理解示例 4.3，就很容易决定使用何种参数传递机制。如果希望函数改变变量的值，那么对应的形参就应为引用传递调用参数，就应该加上“&”符号。否则，就应该使用值传递调用参数。■

### 示例 4.3 比较不同的参数传递机制

```
1 // 演示值传递调用参数与引用传递调用参数之间的区别。
2 #include <iostream>
3 using namespace std;

4 void doStuff(int par1Value, int& par2Ref);
5 //par1Value 是一个值传递调用参数。
6 //par2Ref 是一个引用传递调用参数。

7 int main()
8 {
9     int n1, n2;
10
11     n1 = 1;
12     n2 = 2;
13     doStuff(n1, n2);
14     cout << "n1 after function call = " << n1 << endl;
15     cout << "n2 after function call = " << n2 << endl;
16     return 0;
17 }

18 void doStuff(int par1Value, int& par2Ref)
19 {
20     par1Value = 111;
21     cout << "par1Value in function call = "
22         << par1Value << endl;
23     par2Ref = 222;
24     cout << "par2Ref in function call = "
25         << par2Ref << endl;
26 }
```

## 示例运行结果

```

par2Value in function call = 111
par2Ref in function call = 222
n1 after function call = 1
n2 after function call = 222

```



## 陷阱：无意的局部变量

如果希望函数改变某变量的值，那么对应的形参必须为引用传递调用参数，因此必须在其类型名之后加上“&”符号。如果由于疏忽漏掉了“&”符号，那么实际上该参数就变成了值传递调用参数。函数调用发生时，你将发现该函数调用没有改变对应实参的值，原因就在于值传递调用参数是一个局部变量。在函数体内部对该参数进行了修改，但作为局部变量，该参数的变化在函数体外面不会体现出来。由于代码看上去没有问题，因此这种错误很难被发现。

例如，示例 4.4 中的程序与示例 4.2 中的程序几乎一样，除了函数 swapValues 中的“&”被漏掉以外。结果导致形参 variable1 和 variable2 变成局部变量，实参变量 firstNum 和 secondNum 不会替代形参 variable1 和 variable2，相反 variable1 和 variable2 被初始化为 firstNum 和 secondNum 的值。此后，variable1 和 variable2 的值进行了交换，但变量 firstNum 和 secondNum 的值没有发生任何改变。漏掉的两个“&”符号就使得整个程序完全错误，但它看上去与原本正确的程序几乎一模一样，而且可以进行编译和运行而不会给出任何错误信息。■

## 示例 4.4 无意的局部变量

```

1 // 演示引用传递调用参数的程序。
2 #include <iostream>
3 using namespace std;

4 void getNumbers(int& input1, int& input2);
5 // 从键盘读入两个整数。
6 void swapValues(int variable1, int variable2);
7 // 交换变量 variable1 和 variable2 的值。
8 void showResults(int output1, int output2);
9 // 以此顺序显示变量 output1 和 output2 的值。

10 int main()
11 {
12     int firstNum, secondNum;

13     getNumbers(firstNum, secondNum);
14     swapValues(firstNum, secondNum);
15     showResults(firstNum, secondNum);
16     return 0;
17 }

```

```

18 void swapValues(int variable1, int variable2)
19 {
20     int temp;
21     temp = variable1;
22     variable1 = variable2;
23     variable2 = temp;
24 }
25
26

```

忘记了这里的 &。

无意的局部变量

getNumbers 函数与 showResults 函数的定义与示例 4.2 中的一样。

### 示例运行结果

```

Enter two integers: 5 6
In reverse order the numbers are: 5 6

```

—— 无意的局部变量导致的错误

### 提示：选择形参的名字

函数应该是自包含的模块，独立于程序的其他部分。大型的程序设计项目中，不同的程序员会负责编写不同的函数。程序员应该给函数的形参参数选择一个有意义的名字，代替这些形参的实参是其他函数或者 main 函数里的变量，这些变量同样也应给予有意义的名字，但一般是由其他人来命名的。这种情况下，就有可能使某些或者所有的实参与某些形参重名，但这是完全可以接受的。不管这些将用作实参的变量的名字是什么，它们的名字都不会与形参的名字发生混淆。■

### 示例：买比萨

买东西时，“个儿大”的并不一定都比“个儿小”的好，特别是在买比萨的时候。比萨的大小是以比萨的直径来表示的，但比萨的份量则是由比萨的面积决定的，且面积与直径并不成比例。大多数人都很难分辨出 10 英寸的比萨与 12 英寸的比萨在面积上到底差多少，也就很难决定应该购买哪一个——即每平方英寸的价格最低。示例 4.5 给出了一个程序，顾客可以使用该程序来判断应该购买两种比萨中的哪一个。

注意，函数 getData 与 giveResults 具有相同的参数，但由于 getData 会改变实参的值，因此它的参数是引用传递调用参数。另一方面，giveResults 函数只需要实参的值，因此它的参数是值传递调用参数。

同样需要注意的是，函数 giveResults 包含两个局部变量，而且它的函数体中包含对 unitPrice 函数的调用。最后还需注意，unitPrice 函数同时包含多个局部变量和一个局部常量。

### 自测练习题

7. 如果将示例 4.3 程序中函数 doStuff 的声明及定义的函数头修改为如下的形式，从而使得形参 par2Ref 变为一个值传递调用参数，此时程序的运行结果是什么？

```
void doStuff (int par1Value, int par2Ref);
```

### 示例 4.5 买比萨

```
1 // 判断哪种尺寸的比萨更划算。
2 #include <iostream>
3 using namespace std;

4 void getData(int& smallDiameter, double& priceSmall,
5             int& largeDiameter, double& priceLarge);

6 void giveResults(int smallDiameter, double priceSmall,
7                 int largeDiameter, double priceLarge);

8 double unitPrice(int diameter, double price);
9 // 返回每平方英寸比萨的价格。
10 // 前提条件：参数 diameter 是比萨的直径，
11 // price 是比萨的价格。
12 int main()
13 {
14     int diameterSmall, diameterLarge;
15     double priceSmall, priceLarge;

16     getData(diameterSmall, priceSmall, diameterLarge, priceLarge);
17     giveResults(diameterSmall, priceSmall, diameterLarge, priceLarge);

18     return 0;
19 }

20 void getData(int& smallDiameter, double& priceSmall,
21             int& largeDiameter, double& priceLarge)
22 {
23     cout << "Welcome to the Pizza Consumers Union.\n";
24     cout << "Enter diameter of a small pizza (in inches): ";
25     cin >> smallDiameter;
26     cout << "Enter the price of a small pizza: $";
27     cin >> priceSmall;
28     cout << "Enter diameter of a large pizza (in inches): ";
29     cin >> largeDiameter;
30     cout << "Enter the price of a large pizza: $";
31     cin >> priceLarge;
32 }
33
34 void giveResults(int smallDiameter, double priceSmall,
35                 int largeDiameter, double priceLarge)
36 {
37     double unitPriceSmall, unitPriceLarge;
```

变量 diameterSmall、diameterLarge、priceSmall 和 priceLarge 从函数 getData 中获取数据并传递到函数 giveResults。

```

38     unitPriceSmall = unitPrice(smallDiameter, priceSmall);
39     unitPriceLarge = unitPrice(largeDiameter, priceLarge);
40     cout.setf(ios::fixed);
41     cout.setf(ios::showpoint);
42     cout.precision(2);
43     cout << "Small pizza:\n"
44         << "Diameter = " << smallDiameter << " inches\n"
45         << "Price = $" << priceSmall
46         << " Per square inch = $" << unitPriceSmall << endl
47         << "Large pizza:\n"
48         << "Diameter = " << largeDiameter << " inches\n"
49         << "Price = $" << priceLarge
50         << " Per square inch = $" << unitPriceLarge << endl;
51     if (unitPriceLarge < unitPriceSmall)
52         cout << "The large one is the better buy.\n";
53     else
54         cout << "The small one is the better buy.\n";
55     cout << "Buon Appetito!\n";
56 }

57 double unitPrice(int diameter, double price)
58 {
59     const double PI = 3.14159;
60     double radius, area;

61     radius = diameter/static_cast<double>(2);
62     area = PI * radius * radius;
63     return (price/area);
64 }

```

函数内部调用其他函数。

### 示例运行结果

```

Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!

```

---



## 4.2 重载与默认实参

“……那就意味着你有三百六十四天可能会得到一件不是生日礼物的礼物。”

“当然了。”爱丽丝说。

“并且你知道，只有其中一件生日礼物，将会是你的荣耀！”

“我不懂你说的‘荣耀’是指什么。”爱丽丝说。

矮胖子轻蔑地笑着：“你当然知道了，除非我告诉你。我是指这里有一个对你来说具有压倒性的好理由。”

“但是‘荣耀’的意思并不是具有压倒性的好理由。”爱丽丝反驳道。

“当我用一个词的时候，”矮胖子用一种更加轻蔑的语气说，“它只是在表达我想让它表达的意思，不多也不少。”

“问题是，你是否能用词汇表达那么多重的意思。”爱丽丝说。

“问题是，谁才是主人。就这样！”矮胖子说。

刘易斯·卡罗尔，《爱丽丝镜中奇遇记》

C++ 允许给一个函数名以两个或更多不同的定义，这意味着可以重复使用那些具有直观感召力的名字。例如，可以将三个函数都命名为 `max`：一个用来计算两个数字中的最大值，一个用来计算三个数字中的最大数，还有一个计算四个数字中的最大数。给两个函数定义以相同的函数名，这就叫作**重载**。

### 重载简介

假设你正在编写一个用来计算两个数的平均值的函数，你可能会使用如下的函数定义：

```
double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

再假设该程序还需要一个用来计算三个数的平均值的函数，你或许会定义一个新的名为 `ave3` 的函数，定义如下：

```
double ave3(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

这样做是没有问题的，在许多程序设计语言中除此之外没有其他的选择。但是，C++ 提供了更简洁的解决办法——重载。C++ 中，可以将这两个函数都定义为相同的名字 `ave`，可以用如下的函数定义来取代 `ave3` 函数的定义：

```
double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3) / 3.0);
}
```

因此，函数 `ave` 就有了两个定义。这就是一个重载的例子。本例中，我们重载了函数 `ave`，示例 4.6 将这两个名为 `ave` 的函数放到了一个完整的程序中。但一定要注意，每个函数都有其各自的函数声明。

编译器可以通过检查函数调用时参数的类型及数目来决定应该调用哪个函数定义。在示例 4.6 的程序中，一个 `ave` 函数包含两个实参，另一个 `ave` 函数包含三个实参。对于包含两个实参的函数调用，`ave` 函数的第一个定义被使用；对于包含三个实参的函数定义，`ave` 函数的第二个定义被使用。

### 函数名重载

如果给同一个函数名以两个或更多的函数定义，这种用法就被称为重载。重载函数名时，每个函数定义的形参必须在类型或者数量上有所不同。当函数调用发生时，编译器根据参数的类型以及数目来决定具体使用哪个函数定义。

决定采用哪个函数定义

进行函数名重载时，每个函数定义的形参必须在数目或类型上有所不同。也就是说，任何使用同一函数名的函数定义，其形参的数目或类型必须不同（或都不相同）。注意，当你重载一个函数时，除了函数定义必须不同外，函数的声明也必须在形参上有所差异。如果两个函数仅仅是返回值不同，这种情况是不能重载的。不可以根据形参的数目和类型之外的其他差异来对函数进行重载，或者也不能仅仅以 `const` 或值传递调用与引用传递的差别进行函数重载<sup>1</sup>。

本书的第 1 章已经涉及了重载的概念，就是除法运算符的重载。如果操作数都是 `int` 类型（如 `13/2`），则返回值也是一个整数，本例中为 6；如果两个操作数都为 `double` 类型，那么返回值也为 `double` 类型，例如 `13/2.0` 的结果为 6.5。在此，对于除法运算符有两个定义，这两个定义的不同之处在于操作数的类型不同。除法运算符“/”的重载与函数的重载唯一的区别是：C++ 的设计者预先将“/”重载了，而函数名的重载必须由程序员自己来完成。本书的第 8 章介绍了运算符“+”、“-”等的重载。

#### 示例 4.6 函数名重载

```
1 // 演示函数 ave 的重载。
2 #include <iostream>
3 using namespace std;

4 double ave(double n1, double n2);
5 // 返回 n1 和 n2 的平均值。
6
7 double ave(double n1, double n2, double n3);
8 // 返回 n1、n2 和 n3 的平均值。
```

<sup>1</sup> 事实上，一些编译器允许你根据变量的 `const` 或非 `const` 特性进行函数的重载，但你不能依赖于此。因为 C++ 标准不允许这种用法。

```

9  int main()
10 {
11     cout << "The average of 2.0, 2.5, and 3.0 is"
12         << ave(2.0, 2.5, 3.0) << endl;

13     cout << "The average of 4.5 and 5.5 is"
14         << ave(4.5, 5.5) << endl;

15     return 0;
16 }

17 double ave(double n1, double n2)
18 {
19     return ((n1 + n2)/2.0);
20 }

21 double ave(double n1, double n2, double n3)
22 {
23     return ((n1 + n2 + n3)/3.0);
24 }

```

两个参数

三个参数

### 示例运行结果

```

The average of 2.0, 2.5, and 3.0 is 2.5
The average of 4.5 and 5.5 is 5.0

```

## 签名

函数的签名由函数名以及函数参数列表的类型构成，但不包括 `const` 关键字及“&”符号。函数重载时，须保证不同的函数定义具有不同的函数签名（注意，某些专家也将 `const` 或“&”符号算作函数签名的一部分，但这里为了解释函数重载将其排除在外）。

## 陷阱：自动类型转换与重载

假设程序中有如下的函数定义，且函数名 `mpg` 未被重载（也就是说这是名为 `mpg` 函数的唯一一个定义）。

```

double mpg(double miles, double gallons)
// 返回英里每加仑。
{
    return (miles / gallons);
}

```

如果使用 `int` 类型的实参来调用函数 `mpg`，C++ 将会自动地将该实参转换为 `double` 类型的值。因此，下面的语句在屏幕上的输出将是“22.5 miles per gallon”：

```
cout << mpg(45, 2) << " miles per gallon";
```

C++ 将 45 和 2 分别转换为 45.0 和 2.0，然后执行  $45.0/2.0$ ，最终得到结果 22.5。

### 函数重载与 类型转化的 相互影响

如果函数要求的实参是 `double` 类型，而实际上给出的却是 `int` 类型的实参，那么 C++ 会自动地将该实参转换为 `double` 类型。这一特点非常有用并且顺理成章。但是，函数重载可能会干扰这种自动类型转换。我们先来看一个例子。

假设你重载了名为 `mpg` 的函数，除了之前给出的定义外，程序中还包含如下的定义：

```
int mpg(int goals, int misses)
// 返回净进球数，计算方法为 (goals - misses)。
{
    return (goals - misses);
}
```

在同时包含这两个 `mpg` 函数定义的程序里，下面的语句将输出 “43 miles per gallon”（因为  $45-2$  等于 43）。

```
cout << mpg(45, 2) << " miles per gallon";
```

当 C++ 看到函数调用 `mpg(45, 2)` 时，其中的两个实参都为 `int` 类型。C++ 会首先寻找两个形参类型为 `int` 的 `mpg` 函数定义。如果这样的函数定义存在，C++ 会使用它。C++ 不会优先将 `int` 实参转换为 `double` 类型的值，除非它没有找到两个形参类型为 `int` 的 `mpg` 函数定义。

`mpg` 示例函数说明了函数重载的另外一点：不要让两个毫无关系的函数使用同一个函数名，因为这种用法往往会导致混淆。■

### 自测练习

8. 假设两个函数定义的声明如下：

```
double score(double time, double distance);
int score(double points);
```

下面的函数调用（`x` 为 `double` 类型）将使用哪一个函数定义？给出你的理由。

```
double finalScore = score(x);
```

9. 假设两个函数定义的声明如下：

```
double theAnswer(double data1, double data2);
double theAnswer(double time, int count);
```

下面的函数调用（`x` 和 `y` 都为 `double` 类型）将使用哪一个函数定义？给出理由。

```
x = theAnswer(y, 6.0);
```

### 重载解析的规则

如果采用重载技术为同一个函数名产生两个函数定义，且这两个函数有着类似（但不相同）的参数列表，那么重载和自动类型转换之间的相互干扰就会容易让人混淆。编译器使用如下的准则来决定，对于某一个具体的函数调用应该使用哪个函数定义：

1. 精确匹配：如果实参的数目和类型与某一个函数定义完全吻合（不包括任何自动类型转换），那么就使用这个定义。
2. 自动类型转换后匹配：如果没有精确匹配的函数定义，但经过自动类型转换后可以匹配，则使用该函数定义。

如果使用准则 1 的过程中产生多个匹配的函数定义，或者使用准则 1 没有匹配的定义，但使用准则 2 有多个匹配的定义，那么就会出现定义的混淆，编译器此时会报错。

例如，下面的重载虽然写得比较混乱，但完全合法：

```
void f(int n, double m);
void f(double n, int m);
```

但是，如果出现了这样的函数调用：

```
f(98, 99);
```

编译器就不知道应该将这两个 int 参数中的哪一个转换为 double 类型了，此时就会产生一个错误提示信息。

为了说明这种情况可以造成多大的混淆甚至严重危险，假设我们增加了 f 函数的第三个定义：

```
void f(int n, int m);
```

加上这个函数定义之后，就不会再有错误提示信息了，因为第三个定义符合精确匹配原则。显然，应该尽量避免这种容易混淆的函数重载。

上述两个原则在大多数情况下都有效。事实上，与其依赖编译器的解析规则，不如直接把代码写得更加准确。实际上，C++ 的解析准则比之前提到的两个规则还要复杂得多。为了便于参考，这里给出了完整的准则。其中的某些准则可能需要深入阅读本书才能理解。但不用担心，上面给出的两条准则在绝大多数情况下已经足够了。

1. 精确匹配，如前所述。
2. 整数类型或浮点类型通过提升以实现匹配。例如，short 变成 int，float 变成 double（注意，bool 到 int 以及 char 到 int 的转换也被看作整数类型的提升转换）。
3. 通过预定义类型的其他转换实现匹配，如 int 到 double。
4. 通过用户定义的类型进行转换实现匹配（参见第 8 章）。
5. 使用省略号进行匹配（本书未涉及，因为实际中使用得很少）。

同样，如果出现多个函数定义匹配的情况，则出现了混淆，这将产生错误信息。

---

### 示例：改写“买比萨程序”

比萨的购买者从示例 4.5 的程序中获益匪浅。事实上，现在每个人都能买到性价比最高的比萨。有一家声名狼藉的比萨店过去常常糊弄客户购买比较昂贵的比萨来从中渔利，但是我们的程序完全遏制了此类欺骗行为的发生。但是比萨店的老板不死心，他想出了另外一个花招。他们现在既卖圆形比萨，也卖方形比萨，因为他们知道我们的程序不能处理方形的比萨。示例 4.7 是该程序的一个改进版本，可以比较方形比萨和

圆形披萨。注意，函数 `unitPrice` 已经被重载，从而它既可以处理圆形披萨，又可以处理方形披萨。

#### 示例 4.7 改写后的买披萨程序

```

1 // 判断应该购买圆披萨还是方披萨。
2 #include <iostream>
3 using namespace std;

4 double unitPrice(int diameter, double price);
5 // 返回圆披萨每平方英寸的价格。
6 // 参数 diameter 是圆披萨的直径。
7 // 参数 price 为圆披萨的价格。

8 double unitPrice(int length, int width, double price);
9 // 返回方披萨每平方英寸的价格。
10 // 参数 length 和 width 为方披萨的长和宽。
11 // 参数 price 为方披萨的价格。
12 int main()
13 {
14     int diameter, length, width;
15     double priceRound, unitPriceRound,
16         priceRectangular, unitPriceRectangular;

17     cout << "Welcome to the Pizza Consumers Union.\n";
18     cout << "Enter the diameter in inches"
19         << "of a round pizza: " ;
20     cin >> diameter;
21     cout << "Enter the price of a round pizza: $";
22     cin >> priceRound;
23     cout << "Enter length and width in inches\n"
24         << "of a rectangular pizza: " ;
25     cin >> length >> width;
26     cout << "Enter the price of a rectangular pizza: $";
27     cin >> priceRectangular;
28     unitPriceRectangular =
29         unitPrice(length, width, priceRectangular);
30     unitPriceRound = unitPrice(diameter, priceRound);

31     cout.setf(ios::fixed);
32     cout.setf(ios::showpoint);
33     cout.precision(2);
34     cout << endl
35         << "Round pizza: Diameter ="
36         << diameter << " inches\n"
37         << "Price = $" << priceRound

38         << "Per square inch = $" << unitPriceRound
39         << endl
40         << "Rectangular pizza: Length ="
41         << length << " inches\n"

```

```

42         << "Rectangular pizza: Width = "
43         << width << " inches\n"
44         << "Price = $" << priceRectangular
45         << " Per square inch = $" << unitPriceRectangular
46         << endl;
47     if (unitPriceRound < unitPriceRectangular)
48         cout << "The round one is the better buy.\n";
49     else
50         cout << "The rectangular one is the better buy.\n";
51     cout << "Buon Appetito!\n";

52     return 0;
53 }
54 double unitPrice(int diameter, double price)
55 {
56     const double PI = 3.14159;
57     double radius, area;
58
59     radius = diameter/double(2);
60     area = PI * radius * radius;
61     return (price/area);
62 }
63 double unitPrice(int length, int width, double price)
64 {
65     double area = length * width;
66     return (price/area);
67 }

```

### 示例运行结果

```

Welcome to the Pizza Consumers Union.
Enter the diameter in inches of a round pizza: 10
Enter the price of a round pizza: $8.50
Enter length and width in inches
of a rectangular pizza: 6 4
Enter the price of a rectangular pizza: $7.55

Round pizza: Diameter = 10 inches
Price = $8.50 Per square inch = $0.11
Rectangular pizza: Length = 6 inches
Rectangular pizza: Width = 4 inches
Price = $7.55 Per square inch = $0.31
The round one is the better buy.
Buon Appetito!

```

### 默认参数

#### 默认实参值

可以给函数的一个或者多个值传递调用参数指定**默认实参值**。函数调用时，如果相应的实参被漏掉了，将以默认实参代替。例如，示例 4.8 中的函数 `volume` 根据一个箱子的长、宽、高来计算箱子的容积。如果没有给出具体的高度值，那么高度将被设定为 1。如果高和宽都没有给出，那么两者都会被指定为 1。

注意示例 4.8 中，默认值是放在函数的声明中，而不是函数的定义中的。默认值应该在函数的首次声明中给出（或者函数的定义中，如果函数的定义首先出现）。其后的函数声明以及函数定义都不能再次指定默认实参。因为在某些编译器中，即使后边指定的默认实参与前面指定的完全一致，它也会报错。

可以指定多个默认实参，但所有的默认实参都必须放在最右边的位置上。因此，对于示例 4.8 中的 volume 函数，我们可以为最后一个、最后两个或者所有的参数指定默认实参，但除此之外的其他组合都是不允许的。

如果函数包含多个默认实参，该函数调用时，必须从右开始省略实参。例如，示例 4.8 中有两个默认实参。当省略一个默认实参时，它假定是最后一个实参。不可能直接省略第二个实参而不省略第三个实参。

默认实参的值是固定的，它们有时候可以反映出实参的某些意图。默认实参只能是值传递调用参数，对于引用传递调用参数没有任何意义。任何可以用默认实参做的事情都可以用重载来完成，不过使用默认实参比使用重载简单得多。

#### 示例 4.8 默认实参

```

1
2 #include <iostream>
3 using namespace std;

4 void showVolume(int length, int width = 1, int height = 1);
5 // 返回箱子的容积。
6 // 如果高度没给出，默认为 1。
7 // 如果高和宽都没给出，则默认都为 1。

8 int main()
9 {
10     showVolume(4, 6, 2);
11     showVolume(4, 6);
12     showVolume(4);

13     return 0;
14 }

15 void showVolume(int length, int width, int height)
16 {
17     cout << "Volume of a box with \n"
18         << "Length = " << length << ", Width = " << width << endl
19         << "and Height = " << height
20         << " is " << length*width*height << endl;
21 }

```

默认参数

默认参数不应该出现第二次。

#### 示例运行结果

```

Volume of a box with
Length = 4, Width = 6
and Height = 2 is 48
Volume of a box with
Length = 4, Width = 6
and Height = 1 is 24

```



```
Volume of a box with
Length = 4, Width = 1
and Height = 1 is 4
```

## 自我练习

10. 本题目与之前的“改写后的买比萨程序”有关。假设喜欢糊弄顾客的店主又推出了一个方形比萨,是否可以通过unitPrice函数的重载来处理这个新问题?如果可以,为什么?如果不可以,又是为什么?

## 4.3 函数的测试和调试

我看见了那个怪物——那个由我创造出来的可怜的怪物。

玛丽·沃斯通克拉夫特·雪莱,《科学怪人》

本节介绍一些函数测试和调试的基本方法。

### assert宏

**assertion** **assertion (断言)** 是一个或为 true 或为 false 的语句。assertion 用来检验程序的正确性。我们在第3章提到的前提条件和运行结果就是 assertion 的一个例子。C++ 中 assertion 语句是一个简单的布尔表达式。如果将一个 assertion 语句转化为一个布尔表达式,那么可以使用预定的 assert 宏来检验代码是否符合该 assertion (宏和内联函数非常类似,而且用法也非常接近)。

assert 宏在使用时就如同一个带有 bool 类型传值参数的 void 函数。由于 assertion 是一个布尔表达式,这也就意味着 assert 的实参也是一个 assertion。assert 宏被调用时,会计算 assertion 实参的值。如果值为 true,那么不会有任何事情发生;如果值为 false,程序就会终止并且产生一个错误信息。因此,调用 assert 宏是在程序中加入错误检查的一种简洁方法。

例如,下面的函数声明来自示例 4.3:

```
void computeCoin(int coinValue, int& number, int& amountLeft);
// 前提条件: 0 < coinValue < 100; 0 <= amountLeft < 100
// 运行结果: number 的值等于应找还的各种硬币的最大数量,由 coinValue 和
// amountLeft 得出。amountLeft 减去硬币的总面值,即 number * coinValue 的值。
```

可以用如下的方法检验前提条件是否满足:

```
assert((0 < currentCoin) && (currentCoin < 100)
&& (0 <= currentAmountLeft) && (currentAmountLeft < 100));
computeCoin(currentCoin, number, currentAmountLeft);
```

如果前提条件未能满足,程序终止并抛出错误信息。

assert 宏的定义包含在库 cassert 中,因此使用 assert 宏的程序必须包含如下语句:

```
#include <cassert>
```

关闭对  
assert  
的调用  
  
#define  
NDEBUG

使用 assert 的一个好处是，你可以随时关闭对 assert 的调用。可以在程序中使用 assert 来调试程序，然后在调试完毕后再关闭它们，这样用户就不会得到一些莫名其妙的错误提示了，而且还可以提高程序的执行性能。要关闭程序中所有的 assert，只需在 include 命令前添加 #define NDEBUG 语句，如下所示：

```
#define NDEBUG
#include <cassert>
```

因此，如果程序包含了 #define NDEBUG 语句，那么所有的 assert 都将被关闭。如果稍后又对程序做了修改，再次需要调试程序时，可以删除 #define NDEBUG 这行来恢复 assert 的调试作用。

并非所有的注释 assertion 都能转化为 C++ 布尔表达式。例如，前提条件就比运行结果更容易转化为布尔表达式。因此 assert 不是调试程序的“万能武器”，但确实很有用。

### 占位程序和驱动程序

每个函数都应该作为一个独立的单元进行设计、编码和测试。当我们将每个函数看作独立单元时，一个大型的工程就可以转化为一系列小的、可操作的任务。但如何在程序之外对一个本应包含在程序中的函数进行测试？一个方式是编写专门的测试程序来测试函数。例如，示例 4.9 展示了一个用来测试示例 4.5 中 unitPrice 函数的测试程序。这种程序我们称为**驱动程序**。驱动程序是一种临时的工具，而且可以很小。它们不需要复杂的输入程序，也不需要完成最终程序要完成的运算。它们需要做的只是尽可能简单地获取函数实参的值——一般是从用户那里获取，然后运行该函数并显示结果。如示例 4.9 所示的循环，可以以不同的实参反复测试函数，而无须重新运行程序。

驱动程序

如果对每个函数都单独测试，就可以发现程序中存在的大多数错误，而且还可以发现错误出在哪个函数。如果只是对整个程序进行测试，也许仍然可以发现程序中的问题，但并不知道哪儿出错了。更糟糕的是，你所认为出错的地方其实没有错。

一旦完全测试了某个函数之后，就可以将该函数用到测试其他函数的驱动程序中。而且驱动程序中，除了要测试的函数外，其他函数都是已经被测试过的。在测试其他函数的时候，最好使用已经经过完全测试的函数，这样一旦出现问题，就可以知道问题出在尚未测试的函数中。

占位程序

有时，完全不使用未测试函数去测试另一个函数是不可能或者比较麻烦的。这种情况下，可以先使用一个简化版本的未测试函数，这种简化了的函数被称为**占位程序**。这些占位程序无须进行完全的计算，但可以产生足以用于测试的值，而且它们非常简单，很容易确保它们的正确性。例如，以下代码便是函数 unitPrice 可能的一个占位程序：

```
// 一个占位程序。但仍需写出最终的函数定义。
double unitPrice(int diameter, double price)
{
    return (9.99); // 虽然不正确, 但对一个占位程序而言, 已经足够。
}
```

### 示例 4.9 驱动程序

```
1
2 // 函数 unitPrice 的驱动程序。
3 #include <iostream>
4 using namespace std;

5 double unitPrice(int diameter, double price);
6 // 返回每平方英寸披萨的价格。
7 // 前提条件: 参数 diameter 为披萨的直径, 参数 price 为披萨的价格。

8 int main()
9 {
10     double diameter, price;
11     char ans;

12     do
13     {
14         cout << "Enter diameter and price:\n";
15         cin >> diameter >> price;
16         cout << "Unit Price is $"
17             << unitPrice(diameter, price) << endl;

18         cout << "Test again? (y/n)";
19         cin >> ans;
20         cout << endl;
21     } while (ans == 'y' || ans == 'Y');

22     return 0;
23 }
24
25 double unitPrice(int diameter, double price)
26 {
27     const double PI = 3.14159;
28     double radius, area;

29     radius = diameter/static_cast<double>(2);
30     area = PI * radius * radius;
31     return (price/area);
32 }
```

### 示例运行结果

```
Enter diameter and price:
13 14.75
Unit price is: $0.111126
Test again? (y/n): y
Enter diameter and price:
```

2 3.15

Unit price is: \$1.00268

Test again? (y/n): n

使用占位程序可以方便地测试程序的基本情况，而不用编写一个完整的程序来测试每个函数。也正是因为这个原因，使用占位程序进行测试通常是进行测试的有效方法。一个通用的方法是使用驱动程序测试一些基本函数，像是输入、输出函数，然后再使用带占位符的程序去测试其他函数。每次使用一个函数来代替相应的占位程序：首先使用一个完整的函数定义来代替占位程序，一旦该函数测试完毕，就用其他的函数定义来代替相应的占位函数，依此类推，直到产生最终的程序。

### 测试函数的基本准则

任何函数都应该放在一个程序中进行测试，而且该程序用到的其他函数都是已经充分测试过的函数。

### 自测练习题

11. 测试函数的基本准则是什么？为什么这是测试函数的好方法？
12. 什么是驱动程序？
13. 什么是占位程序？
14. 根据如下的函数声明，编写一个占位程序。不需要给出整个程序，只需给出相应的占位程序。

```
double rainProb(double pressure, double humidity, double temp);  
// 前提条件：参数 pressure 是以英寸水银柱表示的气压机显示的压力。  
// 参数 humidity 是以百分比表示的湿度。  
// 参数 temp 是华氏温度。  
// 函数返回降水概率，介于 0 ~ 1 之间。  
// 0 表示不会降水，1 表示一定会降水。
```

### 本章小结

- 形参是一种占位符，函数调用时，由函数的实参来替换。C++ 中有两种方式来实现这种替换：值传递调用和引用传递调用。相应的参数也有两种基本类型：值传递调用参数和引用传递调用参数。
- 值传递调用参数是一个局部变量，函数调用时，它被初始化为对应的实参值。偶尔将值传递调用参数当作局部变量使用也是有效的。
- 在引用传递调用中，实参必须是一个变量，然后形参被整个实参变量所代替。
- 函数中引入引用传递调用参数很简单，只需在参数类型的后面加上“&”符

号（没有“&”符号就表示是一个值传递调用参数）。

- 对应于一个值传递调用参数的实参在函数调用过程中不会被改变，而对应于引用传递调用参数的实参是可以被改变的。如果希望函数改变一个变量的值，那么记得使用引用传递调用参数。
- 可以给同一个函数名多个函数定义，但前提条件是这些函数定义必须具有不同的参数数目或者不同的参数类型。这种用法被称为函数名的重载。
- 可以在一个函数中为多个值传递调用参数设置默认实参，默认实参均位于参数列表中的最右边。
- 通过检查 `assertion` 的值，`assert` 宏可以用来调试程序。
- 每个函数都必须在程序中进行测试，而且这个程序中所有其他的函数都应该是已经完全测试过的。

### 自测练习题答案

1. 值传递调用参数是一个局部变量。函数调用时，首先计算出实参的值，然后对应的值传递调用参数通过该值进行初始化。
2. 函数可以正常运行。但补充一点：形参 `inches` 是一个值传递调用参数，是一个局部变量。因此实参的值是不会发生变化的。

```
3. 10 20 30
   1 2 3
   1 20 3
```

```
4. Enter two integers: 5 10
   In reverse order the numbers are: 5 5
```

```
5. void zeroBoth(int& n1, int& n2)
   {
       n1 = 0;
       n2 = 0;
   }
```

```
6. void addTax(double taxRate, double& cost)
   {
       cost = cost + (taxRate/100.0)*cost;
   }
```

除以 100 是将百分数转化为分数。例如，10% 就是 10/100.0，或者说是成本的 1/10。

```
7. par1Value in function call = 111
   par2Ref in function call = 222
   n1 after function call = 1
   n2 after function call = 2
```

不同点

8. 应该使用第二个函数定义。因为函数调用中只有一个参数，因此相应的函数定

义也应该只有一个参数。

9. 应该使用第一个函数定义。因为第一个符合精确匹配的准则，即两个 double 类型的参数。
10. 不可以（至少不会非常正常地进行）。原因在于方形披萨和圆形披萨的描述方法是一样的：它们都是用一个数字来描述的。对于圆形披萨，该数字表示披萨的直径；对于方形披萨，该数字则是披萨的边长。在这两种情况下，unitPrice 函数都只需要一个 double 类型的参数用来表示价格，以及一个 int 类型的参数用来表示披萨的大小（不管是半径还是边长）。因此，这两个函数声明的形参不仅数目相同，而且类型也都一致（都是一个 double 类型的参数和一个 int 类型的参数）。所以函数调用发生时，编译器无法判断应该使用哪一个函数定义。不过可以通过定义两个名字不同的函数来阻止狡猾的比萨店老板。
11. 函数测试的基本准则是：每个函数都应该在程序中进行测试，而且该程序中用到的其他函数都应该是已经充分测试过的函数。这是测试一个函数的好方法，因为当发生问题时，很容易就知道是哪个函数出问题了。
12. 驱动程序是一种专门用来测试函数的程序。
13. 占位程序是一个函数的简化版本，通过使用占位程序，可以很方便地对其他函数进行测试。
14. // 这是一个占位程序。

```
double rainProb(double pressure,
                 double humidity, double temp)
{
    return 0.25; // 虽然不正确，但对测试而言足够了。
}
```

## 编程练习

1. 编写一个程序，将 24 小时制的时间转化为 12 小时制的时间，例如将 14:25 转化为 2:25 P.M.，要求输入的是两个整数。程序至少包含三个函数：一个用于输入，一个用于用户转换，一个用于输出。A.M./P.M. 可以存储在一个 char 类型的变量中，'A' 代表 A.M.，'P' 代表 P.M.。因此，执行转换任务的函数还必须包含一个引用传递调用的 char 类型的形参，用来记录是 A.M. 还是 P.M.（这个函数同样会有其他参数）。设计一个函数，使用户可以根据需要多次执行计算，直到用户主动结束程序为止。
2. 任意三角形的面积可以用如下的公式进行计算：

$$Area = \sqrt{s(s-a)(s-b)(s-c)}$$

其中， $a$ 、 $b$ 、 $c$  是三条边的长度， $s$  是半周长， $s=(a+b+c)/2$ 。

编写一个 void 函数，包含五个参数：三个值传递调用参数用来表示三条边的长度，两个引用传递调用参数用来表示面积和周长。函数要处理一些异常情况，例如，不是任何  $a$ 、 $b$ 、 $c$  值都可以组成一个三角形。如果输入的值合法，程序

可以给出正确的结果, 否则程序也应该给出适当的响应。

3. 编写一个找零钱的程序, 处理的范围是 1 ~ 99 美分之间。例如, 如果要找零的钱数为 86 美分, 那么输出应是:

```
86 cents can be given as
3 quarter(s) 1 dime(s) and 1 penny(pennies)
```

采用的硬币面值如下: 25 美分为一个 quarter, 10 美分为一个 dime, 1 美分为一个 penny。不要使用 5 美分和半美元的硬币。程序应包含如下的函数:

```
int computeCoin(int coinValue, int& number, int& amountLeft);
// 前提条件: 0 < coinValue < 100; 0 <= amountLeft < 100
// 运行结果: number 值等于要找还硬币的最大数量, 由 coinValue 和 amountLeft 计
// 算得出。amountLeft 中将减去硬币的总面值, 即减去 number * coinValue。
```

例如, 假设 amountLeft 的值为 86, 那么下面的函数调用完成之后, 变量 number 的值应该是 3, 而 amountLeft 的值应该是 11(因为如果你从 86 美分中拿走 3 个 25 美分, 剩下的就是 11 美分):

```
computeCoins(25, number, amountLeft);
```

设计一个循环, 使得用户可以根据需要反复进行计算, 直到用户主动结束程序为止。(提示: 用整数除法和取余操作来完成这个函数。)

4. 编写一个程序, 读入以英尺、英寸表示的长度, 将其转换为等值的米、厘米表示的形式。该程序至少应包括三个函数: 一个用于输入, 一个用于转换, 一个用于输出。设计一个循环, 使得用户可以根据需要反复进行计算, 直到用户主动结束程序为止。其中 1 英尺为 0.3048 米, 100 厘米为 1 米, 12 英寸为 1 英尺。
5. 仿照上面的题目, 编写一个从米、厘米转换为英尺、英寸的程序。使用函数来实现各个子任务。
6. 编写一个程序, 将前两题中的函数结合起来。程序首先询问用户是要将英尺、英寸转换为米、厘米, 还是要将米、厘米转换为英尺、英寸, 然后根据用户的选择进行相应的转换。用户输入 1 表示执行前一种转换, 输入 2 表示执行后一种转换。程序读入用户的选项后, 执行一个 if-else 语句。if-else 语句的每一分支都是一个函数调用, 这些函数的定义与前两题中的函数非常类似, 这些函数会比较复杂, 可能会嵌套其他的函数调用。设计一个循环, 根据用户的需要不断进行计算, 直到用户主动结束程序为止。
7. 编写一个程序, 读入以磅、盎司表示的重量, 转换为等值的千克、克的表示形式。程序应至少包含三个函数: 一个用于输入, 一个用于转换, 一个用于输出。设计一个循环, 可以根据用户的需要反复进行计算, 直到用户主动结束程序的运行。其中, 2.2046 磅为 1 千克, 1000 克为 1 千克, 16 盎司为 1 磅。
8. 仿照上题, 编写一个从千克、克到磅、盎司的转换程序。使用函数来完成各个子任务。

9. 编写一个程序，将前两题中的函数结合起来。程序首先询问用户是要将磅、盎司转换为千克、克，还是要将千克、克转换为磅、盎司，然后根据用户的选择执行选定的转换。用户输入 1 表示前一种转换，输入 2 表示后一种转换。程序读入用户的选项后，执行一个 if-else 语句，if-else 语句的每一个分支都是一个函数调用，这些函数与前两题中的函数调用类似。设计一个循环，可以根据用户的需要反复进行计算，直到用户主动结束程序为止。
10. 编写一个程序，将第 6 题和第 9 题中的函数结合起来。程序首先询问用户是要进行长度单位的转换还是重量单位的转换。如果用户选择是长度单位的转换，紧接着询问是要将英尺、英寸转换为米、厘米，还是要将米、厘米转换为英尺、英寸；如果用户选择的是重量单位的转换，则继续询问用户是要将磅、盎司转换为千克、克，还是要将千克、克转换为磅、盎司。用户输入 1 表示前一种转换，输入 2 表示后一种转换。程序读入用户的输入后，进入一个 if-else 语句。if-else 语句的每个分支都是一个函数调用，这些函数的定义与第 6 题和第 9 题中的函数非常类似。这些函数会比较复杂，其中可能会嵌套其他的函数调用，不过，可以参照第 6 题和第 9 题中的函数定义，因此很容易实现。注意程序中会出现 if-else 语句嵌套在另一个 if-else 语句中的情况（不过是以一种间接的方式）。外层的 if-else 语句包含两个函数调用作为它的两个分支，这两个函数调用再各自包含一个 if-else 语句。不过我们不需要了解这些细节。这里，我们只需将函数看成是一个隐藏了实现细节的黑盒子就可以了。因此，程序应该设计为一个两路分支，而不是四路分支（尽管程序的执行会出现四种情况）。设计一个循环，使用户可以反复进行转换，直到用户主动结束程序为止。
11. 假设你参加一个节目并获得了一个抽取大奖的机会。你面前有三道门，百万美元的现金大奖就放在其中的一道门后边，另外两道门后边只放有洗碗机专用洗涤剂作为安慰奖。现在主持人让你选择，假设你随机选择了一道门。但是，在揭晓最终结果之前，主持人打开了另外两道门中的一道，并且发现其后为洗碗机专用洗涤剂。此刻，主持人征求你的意见：是坚持原先的选择，还是改为选择剩下的一道门？

编写一个函数模拟该节目。首先该函数随机地将大奖安放在某道门后边，然后让选手随机选择一道门，该函数通过相关计算决定选手是应该继续坚持早先的选择还是应该选择剩下的一道门。除该函数之外，还可以编写被该函数调用的其他帮助函数。

紧接着，对你编写的程序做出修改，模拟该节目 10 000 次。分别统计坚持早先的选择和改变选择两种情况下的中奖次数。假设你是参加节目的选手，为了获得百万美元现金大奖，你应该怎么做？

12. Aaron、Bob 和 Charlie 都想成为最好的解密大师，为此他们展开了无休止的争论。为了结束该争论，他们三人同意生死决斗。Aaron 枪法不是很好，打中目标的



概率只有 1/3。Bob 稍微好一点，击中目标的概率为 1/2。Charlie 则是一个专业的神枪手，百发百中。被击中的人都会死掉，也就意味着无缘最好的解密大师称号。

由于每人的枪法水平不同，为了保证决斗的公平，他们同意按如下的顺序进行：首先是 Aaron，然后是 Bob，最后是 Charlie。比赛持续进行，直到最后剩下一个人，最后一个人将获得最好的解密大师称号。

一个明显且合理的策略是：每个人都尽力杀死当前剩余决斗选手中枪法最好的那位，也只有这样，他自己存活下来的概率才最大。

编写一个程序模拟该决斗，该程序采用以上的决斗策略。程序应该使用随机数来模拟每个决斗选手击中目标的概率。为了实现该程序，你可能需要编写多个子程序或函数。一旦你编写的程序可以模拟一次决斗，为该程序添加一个循环，使其执行 10 000 次。统计每位选手获胜的次数并输出其获胜的概率（例如，对于选手 Aaron，你的程序可能会输出“Aaron won 3595/10,000 duels or 35.95%.”）。

另外一个决斗策略是：Aaron 在第一枪的时候总是故意不击中目标。采用该决斗策略，对你的程序做出修改，最后输出每位决斗选手获胜的概率。对于 Aaron 来讲，哪种策略是最有利的？

13. 假设你了解自己的奔跑速度（英里/小时）。不过，跑步机给出的速度是每英里需要的分钟数和秒数（如 5:30/英里），或者千米/小时。

编写一个名为 `convertToMPH` 的重载函数。第一个函数定义包含两个整型参数，分别代表每英里需要的分钟数和秒数，函数最终返回一个 `double` 类型的速度（单位为英里/小时）。第二个函数定义包含一个 `double` 类型的参数，表示以千米/小时为单位的速度，函数最后返回以英里/小时为单位的速度。注意，1 英里约为 1.61 千米。最后，编写完整的程序，对该函数进行测试。

14. 假设你有一台时光机，可以穿梭到未来的某个时间，不过最多只能是未来的 24 小时。时光机以分钟为单位进行未来的穿梭。为了向时光机输入一个合适的分钟数，你需要编写一个程序。该程序读取一个开始时间和一个结束时间，返回这两个时间点之间的分钟数，其中结束时间和开始时间的间隔总是在 24 小时之内。采用军事符号来记录开始时间和结束时间（例如，0000 代表半夜 0 点，2359 代表 0 点的前一分钟）。

编写一个函数，该函数包含两个 `int` 类型的参数，分别代表以军事符号表示的开始时间和结束时间，函数返回以整数表示的两个时间点之间的分钟数。编写一个完整的程序对该函数进行测试。

15. 编写一个名为 `convertToLowestTerms` 的函数，该函数包含两个名为 `numerator` 和 `denominator` 的整型引用参数。此函数应将这两个数分别看成分数的分子和分母，并输出化简之后的分数的分子和分母。例如，假设

numerator 为 20, denominator 为 60, 该函数最终分别将 numerator 和 denominator 的值改为 1 和 3。这一过程会涉及计算 numerator 和 denominator 的最大公因数, 并用该最大公因数分别除 numerator 和 denominator。如果 denominator 的值为 0, 该函数返回 false, 否则返回 true。编写一个测试程序, 对 convertToLowestTerms 函数进行测试。

16. 考虑如下一个包含游戏选手得分情况的文本文件 scores.txt, 其中 Ronaldo 的最高得分为 10400, Didier 的最高得分为 9800, 等等:

```
Ronaldo
10400
Didier
9800
Pele
12300
Kaka
8400
Cristiano
8000
```

编写一个名为 getHighScore 的函数, 该函数带有一个字符串类型的引用参数和一个整型的引用参数。函数扫描整个文本文件, 并将字符串类型的引用参数设置为得分最高的选手名字, 将整数类型的引用参数设置为该选手的对应得分。

17. 编写一个名为 sort 的函数, 函数包含三个整数类型的引用参数。该函数对输入的三个整数进行排序, 并将第一个参数的值设置为最小的值, 第二个参数的值设置为次小的值, 第三个参数设置为最大的值。例如, 给出如下的赋值: a=30, b=10, c=20; 函数调用 sort(a,b,c) 完成后, a=10, b=20, c=30。注意, 第 5 章介绍的数组概念可以解决任意个数整数的排序问题, 而不仅仅是对三个整数进行排序。





# 数组

## 5.1 数组简介 152

数组的声明和引用 152

提示：对数组使用for循环 154

陷阱：数组的索引始终是从0开始的 154

提示：使用已定义的常量作为数组的大小 154

内存中的数组 155

陷阱：数组越界 157

数组的初始化 157

## 5.2 函数中的数组 159

索引变量作为函数实参 159

整个数组作为函数实参 160

const修饰符 163

陷阱：const参数的不一致使用 164

返回数组的函数 164

示例：生产图表 165

## 5.3 用数组编程 169

部分填充的数组 169

提示：不要吝啬形参的使用 169

示例：查询数组 172

示例：给数组排序 174

## 5.4 多维数组 178

多维数组基础 178

多维数组参数 179

示例：使用二维数组的记分程序 179

# 第5章 数组

在未掌握数据之前就得出结论是最大的错误。  
柯南·道尔爵士，《波西米亚丑闻》（福尔摩斯）

## 概述

数组是包含一系列相同类型数据的集合，例如一组温度值或一系列的名字等。本章介绍 C++ 数组的基本概念和用法，同时还会介绍使用数组进行算法和程序设计的一些基本技巧。

本章的内容与后续章节的关系不大，可以略过本章的内容，直接学习第 6 章和第 7 章中有关类的内容。第 7 章中只有介绍向量的 7.3 节用到了本章的内容。

## 5.1 数组简介

假设要编写一个程序来处理五个测验的得分，比如取得这五次测验的最高分以及计算其他得分比最高分少了多少。要取得最高分，首先得知道这五次测验的得分，因此这五次的得分都必须首先存储起来。得到最高分后，剩下的四个分值再分别与最高分进行比较。为了存储这五个得分，需要可以存储五个 int 变量的东西，我们可以采用五个 int 类型的变量来分别记录。但如果后边需要存储 100 个分值时，这种方法就不是彻底的解决之道了。C++ 中的数组就是解决该类问题的最佳方法。数组就像是使用统一的命名机制声明的五个分值的变量，可能分别名为 score[0], score[1], score[2], score[3], score[4]。这些变量中，保持不变的是数组的名称，这里是 score，所不同的是 [ ] 中的整数。

数组

### 数组的声明和引用

C++ 中，如下语句声明了一个包含五个整型变量的数组：

```
int score[5];
```

这一声明就好像是分别声明了如下的五个 int 变量：

```
score[0], score[1], score[2], score[3], score[4]
```

索引变量  
下标变量  
数组元素  
声明大小

组成数组的一系列变量有多种名称，可以称为索引变量、下标变量或者数组元素。中括号中的数字称为索引或者下标。C++ 中，索引从 0 开始，而不是从 1 或者其他任何数字开始的。数组包含的元素个数称为数组的声明大小，或者简称为数组的大小。声明数组时，数组名后边方括号中的数字就是数组的大小。这样数组中的索引变量（也使用方括号）就被分别编号了，从 0 开始，直到比数组大小少 1 的整数结束。

本例中，数组中的索引变量为 int 类型，但实际上数组索引变量可以是任何数据类型。例如，声明一个索引变量为 double 类型的数组，只需在数组声明时将类型名

## 基本类型

称 `int` 替换为 `double` 即可。不过需要注意的是，同一数组中的所有索引变量都为同一类型，这称为数组的基本类型。本例中，数组 `score` 的基本类型为 `int`。

可以在声明普通变量的同时声明数组类型。例如，下面的代码在声明两个 `int` 类型的变量 `next` 和 `max` 的同时，也声明了一个名为 `score` 的数组：

```
int next, score[5], max;
```

数组的索引变量，如 `score[3]`，与普通变量的作用完全等价，任何可以使用普通变量的地方就可以使用同类型的索引变量。

一定不要混淆方括号的两种用法，在数组声明中，方括号用在数组名之后，如：

```
int score[5];
```

此时方括号中的数组表明该数组的大小，也就是该数组包含多少个索引变量。在其他情况下，方括号中的数字表明这是哪一个索引变量。例如，`score[0] ~ score[4]` 就是上述数组的五个索引变量。

方括号中的整数不一定必须是整数常量，方括号内可以使用各种表达式语句，只要该表达式的值是一个大于或等于 0 且小于数组大小的整数即可。例如，下面的语句将索引变量 `score[3]` 的值设为 99：

```
int n=2;
score[n+1] =99;
```

虽然看上去不同，但由于 `n+1` 等于 3，所以 `score[3]` 和 `score[n+1]` 其实是该数组的同一个索引变量。

一个索引变量（如 `score[i]`）的值是由它的索引值来决定的，在这里该索引值就是 `i`。因此可以在程序中写“对第 `i` 个索引变量进行某操作”，其中 `i` 的值是由程序计算而得出的。例如，示例 5.1 中的程序读入得分值，然后按照本章开始所提出的方法对这些得分进行处理。

### 示例 5.1 使用数组的程序

```
1 // 读入五个分值，分别计算其与最高分的差值。
2 #include <iostream>
3 using namespace std;

4 int main()
5 {
6     int i, score[5], max;

7     cout << "Enter 5 scores:\n";
8     cin >> score[0];
9     max = score[0];
10    for (i = 1 ; i<5; i++)
11    {
12        cin >> score[i];
13        if(score[i] > max)
14            max = score[i];
15        //max 是所有分值中的最大值。
16    }
```

```

17     cout << "The highest score is " << max << endl
18         << "The scores and their\n"
19         << "differences from the highest are:\n";
20     for (i=0; i<5; i++)
21         cout << score[i] << "off by "
22             << (max - score[i]) << endl;
23     return 0;
24 }

```

### 示例运行结果

```

Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4

```

### 提示：对数组使用 for 循环

示例 5.1 中的第二个 for 循环给出了一个遍历数组的常用方法：

```

for (i = 0; i < 5; i++)
    cout << score[i] << " off by "
        << (max - score[i]) << endl;

```

for 语句非常适合对数组进行各种操作。■

### 陷阱：数组的索引始终是从 0 开始的

数组的索引从 0 开始，并以比数组大小小 1 的整数结束。■

### 提示：使用已定义的常量作为数组的大小

再看一下示例 5.1 中的程序，它只适用于有 5 个学生的班级。但现实中的班级学生数往往大于 5。那如何让我们的程序能适应其他的情况呢？其中的一个方法就是用定义的常量作为数组的大小。例如，示例 5.1 中的程序可以用如下定义的常量进行改写：

```
const int NUMBER_OF_STUDENTS = 5;
```

数组声明的相关语句此时就应为：

```
int i, score[NUMBER_OF_STUDENTS], max;
```

当然，程序中所有用到数组大小的地方都应该将 5 改为 NUMBER\_OF\_STUDENTS。程序一旦进行了这样的修改（或者你一开始就是这样写的程序），那么后续碰到另外一个班级的时候，只需要改变 NUMBER\_OF\_STUDENTS 的值即可。

数组的大小不能是一个变量，考虑如下有问题的语句：

```
cout << "Enter number of students:\n";
cin >> number;
int score[number]; // 对大多数编译器, 该语句非法。
```

可能有些编译器允许用一个变量来声明数组, 但考虑到可移植性, 最好不要这么做 (在第 10 章中, 我们会介绍另一种可以在程序运行过程中定义大小的数组)。■

## 数组定义

### 语法

```
Type_Name Array_Name[Declared_Size];
```

### 示例

```
int bigArray[100];
double a[3];
double b[5];
char grade[10], oneGrade;
```

以上方式声明的数组会含有 *Declared\_Size* 个索引变量。具体来讲, 这些索引变量从 *Array\_Name[0]* 到 *Array\_Name[Declared\_Size-1]*, 每个索引变量的类型都为 *Type\_Name*。

数组 *a* 包含索引变量 *a[0]*、*a[1]* 以及 *a[2]*, 均为 *double* 类型。数组 *b* 包含索引变量 *b[0]*、*b[1]*、*b[2]*、*b[3]* 及 *b[4]*, 同样都为 *double* 类型。数组的声明和简单变量的声明可以放在一起, 如上面 *grade* 变量所示的那样。

## 内存中的数组

地址

在讨论数组在计算机内存中的表示方式之前, 先让我们看看单一的变量, 如一个 *int* 或者 *double* 类型的变量是如何在计算机内存中表示的。计算机的内存由一系列数字编码的内存单元构成, 每个内存单元大小为一个字节。<sup>1</sup> 其中内存单元的编码被称为该内存单元的地址。一个单一的变量是由一些连续的内存单元组成的, 内存单元的数量由变量的类型决定。因此, 对单一变量的描述包含两方面的信息: 内存中的地址 (该变量第一个字节的内存地址) 以及该变量的类型, 变量的类型表明该变量由多少字节组成。当我们说变量的地址时, 指的就是这个意思。程序在一个变量中存储一个值, 实际上是这个值被存放在该变量对应的内存单元中。相应地, 当某一变量以引用传递的方式赋给函数的实参时, 真正传递给该函数的其实是该变量的地址。下面, 让我们看看数组是如何存储在内存中的。

内存中的  
数组

数组索引变量在内存中的存储方式与一般的变量没有什么区别, 但对于数组, 稍微有一点不同。数组中的一系列索引变量是连续存放在内存中的。例如, 考虑下面的数组:

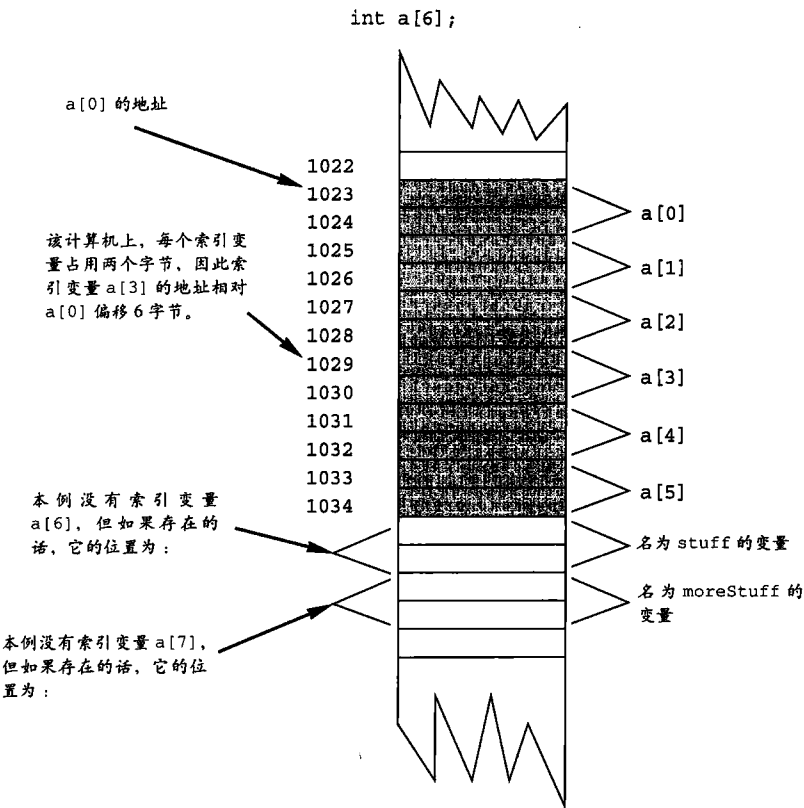
```
int a[6];
```

<sup>1</sup> 1字节包含8比特, 不过这里不关心字节的大小。



声明这个数组时，计算机为该数组分配足够的内存单元来存放 6 个 `int` 类型的变量，并且计算机总是在一段连续的内存单元中存放这 6 个变量。然后，计算机会计录索引变量 `a[0]` 的地址，但不会记录其他索引变量的地址。当程序需要使用该数组其他索引变量的时候，计算机会根据该变量与索引变量 `a[0]` 的距离来计算相应的地址。例如，从索引变量 `a[0]` 开始，向后数 3 个 `int` 类型变量的内存单元数，就可以得到 `a[3]` 的地址。也就是说，为了获取 `a[3]` 的地址，计算机从索引变量 `a[0]` 的地址开始，加上 3 个 `int` 类型变量的字节数，就可以得到 `a[3]` 的地址，实例 5.2 对这一过程做了相应的图解。有关 C++ 数组的更多细节只能在学习了计算机内存的相关内容之后再慢慢体会。例如，在下一个“陷阱”中，我们会解释当程序使用一个非法的数组索引时将碰到什么问题。

示例 5.2 内存中的数组





## 陷阱：数组越界

使用数组时最常碰到的一个错误就是访问一个不存在的索引变量，例如，考虑如下的数组声明：

```
int a[6];
```

非法数组  
索引

使用数组 `a` 时，每个索引表达式的值都应该满足大于或等于 0 且小于或等于 5。例如，如果程序包含索引变量 `a[i]`，那么 `i` 的值必须属于 0、1、2、3、4、5 中的一个，否则就会出现错误。这种索引超出数组索引范围的错误，我们称为访问越界。在大多数系统中，数组越界会导致程序出现错误，而且往往不会给出任何警告。

例如，假设程序包含了如下的语句，数组 `a` 的声明如前面所示：

```
a[i] = 238;
```

现在，假定 `i` 的值被错误地设定为 7，计算机会把 `a[7]` 当作一个合法变量而继续运行。计算机计算出 `a[7]` 的地址后，将 238 放在该地址对应的内存单元中。但不幸的是，索引变量 `a[7]` 是不存在的，存放 238 的那部分内存单元可能属于其他变量，也许是一个名为 `moreStuff` 的变量。此时，变量 `moreStuff` 的值发生了错误的改动，整个过程如示例 5.2 的图解。

操作数组的循环在第一次以及最后一次迭代时，最容易发生此种错误，因此花些时间仔细检查这些循环，确保对数组的合法访问。■

## 数组的初始化

数组可以在声明的同时进行初始化。初始化数组时，所有索引变量的值依次放在大括号里，以逗号隔开。例如，考虑如下的语句：

```
int children[3] = {2, 12, 1}
```

该语句等价于下面的代码：

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

如果大括号中的初始化值少于数组索引变量的个数，那么就按照顺序依次初始化各索引变量，未被初始化的索引变量就默认初始化为数组基本类型对应的 0 值。本例中，未提供初始化值的索引变量被默认初始化为 0。但在函数定义内部（包括程序的 `main` 函数）声明的未提供初始化值的数组以及其他变量，不会进行默认初始化。虽然有些情况下数组的索引变量会自动初始化为对应的 0 值，但还是建议手动初始化每个索引变量，而不要指望自动初始化。

如果数组在声明的同时就进行了初始化，那么该数组的大小就可以省略，并且给出了多少个初始化值，数组的大小就会是多少。例如，下面的语句：

```
int b[] = {5, 12, 11};
```

等价于：

```
int b[3] = {5, 12, 11};
```

### 自测练习

1. 说明 `int a[5]` 与 `a[4]` 表达的意义有何不同, 其中的 `[5]` 和 `[4]` 分别代表什么意思?

2. 从以下数组的声明中:

```
double score[5];
```

我们可以获取以下的哪几项?

- 数组名
- 数组的基本数据类型
- 数组的声明大小
- 数组的索引值的范围
- 数组的一个索引变量

3. 以下数组声明中有什么问题, 请指出。

a. `int x[4] = { 8, 7, 6, 4, 3 };`

b. `int x[] = { 8, 7, 6, 4 };`

c. `const int SIZE = 4;`

```
int x[SIZE];
```

4. 以下代码的输出是什么?

```
char symbol[3] = {'a', 'b', 'c'};
for (int index = 0; index < 3; index++)
    cout << symbol[index];
```

5. 以下代码的输出是什么?

```
double a[3] = {1.1, 2.2, 3.3};
cout << a[0] << " " << a[1] << " " << a[2] << endl;
a[1] = a[2];
cout << a[0] << " " << a[1] << " " << a[2] << endl;
```

6. 以下代码的输出是什么?

```
int i, temp[10];
for (i = 0; i < 10; i++)
    temp[i] = 2*i;
for (i = 0; i < 10; i++)
    cout << temp[i] << " ";
cout << endl;
for (i = 0; i < 10; i = i + 2)
    cout << temp[i] << " ";
```

7. 以下代码存在什么问题?

```
int sampleArray[10];
for (int index = 1; index <= 10; index++)
    sampleArray[index] = 3*index;
```

8. 假设我们希望数组 `a` 中的元素按照以下排序:

```
a[0] ≤ a[1] ≤ a[2] ≤ ...
```

但为了确保顺序,需编写一个检查程序。该检查程序能在元素顺序不对时给出警告,下面的代码为该检查程序的实现,请找出其中存在的问题。

```
double a[10];
    <Some code to fill the array a goes here.>
for (int index = 0; index < 10; index++)
    if (a[index] > a[index + 1])
        cout << "Array elements " << index << " and "
            << (index + 1) << " are out of order.";
```

9. 编写一个 C++ 程序,该程序从键盘读入整数并保存在大小为 20 的 int 整型数组中。

只需给出相关的程序片段即可,记得给出数组以及其他相关变量的声明。

10. 假设程序包含如下数组声明:

```
int yourArray[7];
```

另外,假设 C++ 中每个 int 类型占两字节内存。程序运行时,该数组占多少内存?

假设程序运行时系统将地址 1000 指定给索引变量 yourArray[0],那么索引变量 yourArray[3] 的地址是多少?

## 5.2 函数中的数组

数组的索引变量以及整个数组都可以作为函数的实参,这里首先讨论数组的索引变量作为函数实参的情况。

### 索引变量作为函数实参

索引变量与其他同类型的一般变量一样,都可以作为函数的实参。例如,假设一个程序包含了如下的声明:

```
double i, n, a[10];
```

如果函数 myFunction 可以使用一个 double 类型的实参,那么下面的调用没有问题:

```
myFunction(n);
```

由于数组 a 的索引变量同样也是 double 类型,那么下面的调用同样合法:

```
myFunction(a[3]);
```

索引变量既可以作为传值调用实参,也可以作为引用调用实参。

而且使用索引变量作为函数实参还有一个灵活的地方。例如,考虑如下的函数调用:

```
myFunction(a[i]);
```

如果 i 的值为 3,则函数的实参为 a[3];如果 i 的值为 0,那么这个调用就等价于:

```
myFunction(a[0]);
```

函数在调用之前,首先会对索引表达式进行求值,从而确定函数的实参具体是哪个索引变量。

### 自测练习题

11. 考虑如下的函数定义：

```
void tripler(int& n)
{
    n = 3 * n;
}
```

以下哪些函数调用是正确的？

```
int a[3] = {4, 5, 6}, number = 2;
tripler(a[2]);
tripler(a[3]);
tripler(a[number]);
tripler(a);
tripler(number);
```

12. 以下代码是否有错，如有，请指出。其中函数 tripler 的定义在自测练习题 11 中。

```
int b[5] = {1, 2, 3, 4, 5};
for (int i = 1; i <= 5; i++)
    tripler(b[i]);
```

### 整个数组作为函数实参

函数的形参可以是一个数组，在函数调用时，整个数组可以作为一个参数传递给数组。但是，用数组作为函数形参既不是值传递调用也不是引用传递调用，它是一种新的形参类型，称为**数组参数**。我们通过下面的例子来说明。

数组参数

示例 5.3 中定义的函数带有一个数组参数 *a*，当该函数被调用时，它会被一个数组所代替。函数里还有一个普通的传值调用参数（即 *size*），它的值为一整数，表示数组的大小。该函数将用户键盘输入的值存入数组参数 *a* 中（也即数组的索引变量中），并在屏幕上输出最后一个被使用的数组索引值。

形参 `int a[]` 是一个数组参数，其中方括号中没有索引表达式，表明这是一个数组参数。数组参数虽然不是引用参数，但在实际的使用中它表现得很像后者。我们一步步细看示例 5.3，看看**数组实参**是如何工作的（很明显，**数组实参**就是替代数组形参（如 `a[]`）的具体数组）。

数组实参

### 示例 5.3 带有数组参数的函数

#### 函数声明

```
void fillUp(int a[], int size);
// 前提条件：size 是数组 a 的声明大小，用户将输入对应数目的整数。
// 运行结果：数组 a 包含键盘输入的 size 个整数。
```

#### 函数定义

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << "numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
}
```

```
cout << "The last array index used is " << (size - 1) << endl;
}
```

函数 `fillUp` 被调用时，必须包含两个实参：第一个是整数数组，第二个给出该数组的大小。例如，下面就是合法的调用语句：

```
int score[5], numberOfScores = 5;
fillUp(score, numberOfScores);
```

何时使用  
[]

函数 `fillUp` 的这次调用将为数组 `score` 填充五个从键盘输入的整数。注意，函数形参 `a[]`（包含在函数的声明及函数定义的头部）的方括号中是没有索引表达式的（即使在该方括号中填入一个数字，编译器还是会忽略该数字，因此本书中一律不填）。另一方面，函数调用中给出的实参（本例中是 `score`），其后既没有方括号，也没有索引表达式。

函数调用过程中数组实参 `score` 是怎么处理的？简单来讲，就是用数组实参 `score` 替换函数体中出现的所有形参 `a`，然后执行函数体语句。因此函数调用：

```
fillUp(score, numberOfScores);
```

其实等价于如下的代码：

```
{
    size = 5; ← 5 为 numberOfScores 变量的值。
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> score[i];
    cout << "The last array index used is " << (size - 1) << endl;
}
```

形参 `a` 与之前我们遇到的形参都不同，它是一种新的形参类型。形参 `a` 几乎就是实参 `score` 的占位符。当采用 `score` 作为数组实参调用 `fillUp` 函数时，计算机的处理方式就好像是用数组实参 `score` 替换了函数体中出现的形参 `a`。函数调用中，如果实参为一个数组，那么函数体中任何作用在数组参数上的操作都会反映到该实参数组上，因此，函数可以改变数组实参的值。如果函数体中形参被修改了（如被 `cin` 语句修改了），那么数组实参也会随之发生变化。

从上面的介绍来看，数组参数似乎就是一个数组类型的引用参数。这个说法基本成立，数组参数与引用参数还是有细微的差别的。为了透彻理解它们之间的不同，我们先来复习一下数组的有关内容。

内存中的  
数组

数组是内存中一块连续的内存单元，例如考虑以下数组 `score` 的声明：

```
int score[5];
```

声明该数组时，计算机会为该数组分配足够存放五个 `int` 变量的内存空间，这五个变量将逐一地存放在计算机的内存中。计算机只记录第一个索引变量 `score[0]` 的地址，除此之外还记录该数组的大小为 5，数据的基本类型为 `int`。当程序需要使用索引变量 `score[3]` 时，计算机将根据 `score[0]` 的地址计算 `score[3]` 的地址，因为计算机知道 `score[3]` 的地址在 `score[0]` 的地址再往后三个 `int` 变量的地方。因此，计算机在 `score[0]` 地址的基础上再增加三个 `int` 变量所占用的内存数，得到的结果就是 `score[3]` 的地址。

这样看来，一个数组包含三部分信息：第一个索引变量的地址（内存中的地址）、数组的基本类型（决定每个索引变量占多少内存）以及数组的声明大小（即索引变量的个数）。当数组用作函数的数组实参时，只有第一部分信息传递给被调用函数。当一个数组代入其对应的数组形参时，传递给函数的只是数组第一个索引变量的地址。由于数组实参的基本类型必须与数组形参的类型一致，所以函数也知道数组实参的基本类型。但是，数组实参不会告诉函数数组声明的大小。函数运行时，计算机只知道数组的开始地址，每个索引变量占多大内存，但是（除非明确说明），计算机不知道数组一共有多少个索引变量。这也是为什么需要一个 `int` 实参来告诉函数数组的大小（这一点就是数组与引用参数不同的地方，其实可以将数组参数看作一个弱化了了的引用参数，因为它没有告诉函数数组的大小）。<sup>2</sup>

不同大小的  
数组传递给  
同一个数组  
参数

数组参数看起来似乎有点奇怪，但它们有一个非常好的优势。只要数组的基本类型一致，同一个函数可以用来处理不同大小的数组。我们通过示例 5.3 来解释这一优势，示例 5.3 包含如下的数组声明：

```
int score[5], time[10];
```

下面第一个对函数 `fillUp` 的调用给数组 `score` 填充了五个值，第二个调用给数组 `time` 填充了 10 个值。

```
fillUp(score, 5);
fillUp(time, 10);
```

可以用同一个函数来处理不同大小的数组，原因就在于数组的大小是一个独立的实参。

### 数组形参与实参

函数的实参可以是一个数组，但数组实参既不是传值参数也不是引用传递参数，它是一种新的实参类型，称为数组实参。当一个数组实参被传递给一个函数的数组形参时，提供给该函数的只是该数组实参第一个索引变量（即索引值为 0 的索引变量）的地址。数组实参不会告诉函数该数组的大小，因此当函数的参数是一个数组时，往往需要一个 `int` 类型变量来存储数组的大小。

数组实参与引用传递参数比较类似：如果函数体中改变了数组形参，那么函数调用后，该改变同样会反映到数组实参上。因此，函数可以改变数组实参的值（也就是数组索引变量的值）。

声明一个含有数组参数的函数，其语法如下。

**语法**

```
Type_Returned Function_Name(..., Base_Type Array_Name[ ],...);
```

**示例**

```
void sumArray(double& sum, double a[ ], int size);
```

<sup>2</sup> 如果你之前了解过指针的概念，那么可以看出这种用法其实就是一个指针。实际情况也是，数组实参是通过一个指针传递给函数的。关于这部分的详细内容，本书的第10章会有介绍。如果你之前没有接触过指针的概念，那么直接忽略该脚注。

## const修饰符

函数调用中使用数组参数时，函数可以改变该数组的值，通常情况下这没有什么问题。但在一些复杂的场景下，函数的某些代码可能会不经意间改变一些不应该被改变的数组。为了确保程序运行的正确，我们可以显式地告诉编译器该数组不能被修改，计算机就会进行检测以确保没有代码对该数组进行修改。为此，我们只需在函数数组形参的前面加上一个 const 修饰符。加上了 const 修饰符的数组参数被称为**常量数组参数** (constant array parameter)。

const  
常量数组  
参数

例如，下面的函数输出数组的值，但不改变数组的值：

```
void showTheWorld(int a[], int sizeOfa)
// 前提条件：参数 sizeOfa 是数组 a 的声明大小，数组 a 已经被初始化。
// 运行结果：数组 a 的值被输出到屏幕上。
{
    cout << "The array contains the following values:\n";
    for (int i = 0; i < sizeOfa; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

这个函数没有任何问题，但是为了万无一失，可以在函数定义的第一行加上 const 修饰符，如下所示：

```
void showTheWorld(const int a[], int sizeOfa)
```

加上 const 修饰符之后，如果有代码修改数组参数，那么计算机就会报错。例如，下面是函数 showTheWorld 的另一个实现，其中包含一个修改数组参数的语句。由于该函数的数组形参之前有 const 修饰符，因此编译器会报错，这将有助于我们及早发现错误。

```
void showTheWorld(const int a[], int sizeOfa)
// 前提条件：参数 sizeOfa 是数组的声明大小，数组 a 已被初始化。
// 运行结果：数组 a 的值输出到屏幕上。
{
    cout << "The array contains the following values:\n";
    for (int i = 0; i < sizeOfa; a[i]++)
        cout << a[i] << " ";
    cout << endl;
}
```

错误，只有使用了 const 修饰符，编译器才会发现该错误。

如果上述函数没有包含 const 修饰符，程序在编译时将不会有任何错误提示信息。一旦运行该程序，将陷入一个无限循环，不断增加 a[0] 的值并将其显示在屏幕上。

函数 showTheWorld 定义中所犯的错误就在于 for 循环使用了错误的自增项。应该使索引 i 逐步自增，而不是 a[i] 自增。在这个错误的实现中，索引 i 的起始值为 0，然后就没有改变过。但 a[i]，也就是 a[0] 却一直在自增。当索引变量 a[i] 的值改变时，由于有 const 修饰符，因此编译器将产生错误信息。从而可以及早发现程序中存在的问题。

一般来讲，程序中除了函数定义外还会有函数声明，因此在函数定义中使用 const 修饰符之后，还需在函数声明中加上 const 修饰符，从而保证函数声明和定义



的一致。

理论上, `const` 修饰符可以用来修饰任何类型的参数。但实际上, `const` 修饰符一般只用来修饰数组参数和引用类型的类参数。本书的第6章和第7章介绍了类类型相关的知识。



### 陷阱：const 参数的一致使用

`const` 参数在函数中要么全用, 要么就都不用。如果对某一类型的数组参数使用了 `const`, 那么其他所有同一类型、不应该被函数改变的数组参数都应该加上 `const` 修饰符。这种要求与函数的嵌套调用有关。考虑如下 `showDifference` 函数的定义, 该函数的定义中还用到了名为 `computeAverage` 的函数:

```
double computeAverage(int a[], int numberUsed);
// 返回数组 a 中前 numberUsed 个元素的平均值。数组 a 不被修改。

void showDifference(const int a[], int numberUsed)
{
    double average = computeAverage(a, numberUsed);
    cout << "Average of the " << numberUsed
        << " numbers = " << average << endl
        << "The numbers are:\n";
    for (int index = 0; index < numberUsed; index++)
        cout << a[index] << " differs from average by "
            << (a[index] - average) << endl;
}
```

大多数编译器对以上代码都会报错或者报警。函数 `computeAverage` 不会改变数组 `a` 的值。但是, 当编译器处理 `showDifference` 的定义时, 它会认为 `computeAverage` 函数会改变 (至少是有可能改变) 参数 `a` 的值。这是因为当编译器处理 `showDifference` 的定义时, 编译器只知道 `computeAverage` 函数的声明, 而该函数的声明中并没有包含 `const` 修饰符。因此如果在函数 `showDifference` 的参数 `a` 前加了 `const`, 那么在函数 `computeAverage` 的参数 `a` 前也应该使用 `const` 修饰符。函数 `computeAverage` 的正确声明方式如下:

```
double computeAverage(const int a[], int numberUsed); ■
```

### 返回数组的函数

函数返回一个数组的方式与其返回一个 `int` 或者 `double` 变量的方式是不同的。有一种方式可以让一个函数实现与返回一个数组等价的目的。这种方式就是返回一个指向数组的指针。本书的第10章在讲到数组与指针的相互作用时会介绍这一点。除非你知道怎么使用指针, 否则无法写出一个返回数组的函数。

### 示例：生产图表

示例 5.4 的程序使用了一个数组和一些数组参数。该程序可以显示条状图表，用来表示 Apex 塑料匙生产公司四个生产工厂在任意一周内的生产情况。工厂内各车间有着不同的生产指标，比如茶匙车间、汤匙车间、调酒匙车间等。此外，各个工厂下属的车间数也各不相同。

正如示例 5.4 运行结果显示的那样，每 1000 个产量单位用一颗星来表示。既然运行结果以 1000 为单位，那么就应该将实际的产量除以 1000。这就会带来一个问题，计算机只能显示整数颗星，例如产量 1600 就不能用 1.6 颗星来表示，必须对其进行四舍五入，用两颗星来表示。

数组 `production` 用来保存四个工厂的产量。C++ 数组的索引都是从 0 开始的，但工厂的序号却是 1 ~ 4，而不是 0 ~ 3，因此将工厂 `n` 的产量放在索引变量 `production[n-1]` 中，也就是 1 号工厂的产量是放在索引变量 `production[0]` 中的，2 号工厂的产量放在 `production[1]` 中，依次类推。

由于最后的输出以 1000 为单位，因此程序会对数组 `production` 中保存的产量进行转化。如果 3 号工厂的产量是 4040，那么索引变量 `production[2]` 的初始值就为 4040，紧接着 4040 转化为 4.04 并进行四舍五入，最后得到整数 4，所以最后用 4 颗星来表示 3 号工厂的产量。产量的转化过程由函数 `scale` 完成，该函数以数组 `production` 为实参，对数组中的产量分别除以 1000。

函数 `round` 完成四舍五入功能。例如，`round(2.3)` 返回 2，`round(2.6)` 返回 3。`round` 函数已经在第 3 章中介绍过了，在名为“`round` 函数”的示例程序中。

---

### 示例 5.4 生产图表程序

```
1 // 读入相关数据，显示各工厂产量的条形图。
2 #include <iostream>
3 #include <cmath>
4 using namespace std;
5 const int NUMBER_OF_PLANTS = 4;

6 void inputData(int a[], int lastPlantNumber);
7 // 前提条件：lastPlantNumber 是数组 a 的声明大小。
8 // 运行结果：plantNumber 从 1 到 lastPlantNumber，
9 // a[plantNumber - 1] 等于编号为 plantNumber 工厂的产量。

10 void scale(int a[], int size);
11 // 前提条件：a[0] 到 a[size-1] 都为非负值。
12 // 运行结果：a[i] 里原来的值除以 1000，再四舍五入到一个新值。
13 // 所有 i 都满足：0 <= i <= size-1。

14 void graph(const int asteriskCount[], int lastPlantNumber);
15 // 前提条件：a[0] 到 a[lastPlantNumber-1] 都为非负值。
16 // 运行结果：输出条形图显示各工厂的产量。
```

```

17 // 其中, 第N个工厂的产量为 a[N-1], 单位为1000个生产量单位,
18 // 其中N满足: 1<= N <= lastPlantNumber

19 void getTotal(int& sum);
20 // 从键盘读入一系列非负整数,
21 // 将这些数的和存放到变量 sum 中。

22 int round(double number);
23 // 前提条件: number >= 0
24 // 返回四舍五入之后的值。

25 void printAsterisks(int n);
26 // 在屏幕上删除 n 个星号。

27 int main()
28 {
29     int production[NUMBER_OF_PLANTS];

30     cout << "This program displays a graph showing\n"
31           << "production for each plant in the company.\n";

32     inputData(production, NUMBER_OF_PLANTS);
33     scale(production, NUMBER_OF_PLANTS);
34     graph(production, NUMBER_OF_PLANTS);
35     return 0;
36 }

37 void inputData(int a[], int lastPlantNumber)
38 {
39     for(int plantNumber =1;
40         plantNumber<=lastNumber;plantNumber++)
41     {
42         cout << endl
43              << "Enter production data for plant number "
44              << plantNumber << endl;
45         getTotal(a[plantNumber - 1]);
46     }
47 }

48 void getTotal(int& sum)
49 {
50     cout << "Enter number of units produced by each department.\n"
51           << "Append a negative number to the end of the list.\n";

52     sum = 0;
53     int next;
54     cin >> next;
55     while(next >= 0)
56     {
57         sum = sum + next;
58         cin >> next;
59     }

60     cout << " Total = " << sum << endl;

```

```

61 }
62
63 void scale(int a[], int size)
64 {
65     for(int index=0; index<size;index++)
66         a[index] = round(a[index]/1000.0)
67 }
68
69 int round(double number)
70 {
71     return static_cast<int>(floor(number + 0.5));
72 }
73
74 void graph(const int asteriskCount[], int lastPlantNumber)
75 {
76     cout << "\nUnits produced in thousands of units:\n\n";
77     for (int plantNumber = 1;
78         plantNumber <= lastPlantNumber; plantNumber++)
79     {
80         cout << "Plant #" << plantNumber << " ";
81         printAsterisks(asteriskCount[plantNumber - 1]);
82         cout << endl;
83     }
84 }
85
86 void printAsterisks(int n)
87 {
88     for (int count = 1; count <= n; count++)
89         cout << " ";
90 }

```

### 示例运行结果

This program displays a graph showing  
Production for each plant in the company.

Enter production data for plant number 1  
Enter number of units produced by each department.  
Append a negative number to the end of the list.  
**2000 3000 1000 -1**  
Total = 6000

Enter production data for plant number 2  
Enter number of units produced by each department.  
Append a negative number to the end of the list.  
**2050 3002 1300 -1**  
Total = 6352

Enter production data for plant number 3  
Enter number of units produced by each department.  
Append a negative number to the end of the list.  
**5000 4020 500 4348 -1**  
Total = 13868

```

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
2507 6050 1809 -1
Total = 10366

```

Units produced in thousands of units:

```

Plant #1 *****
Plant #2 *****
Plant #3 *****
Plant #4 *****

```

### 自测练习题

- 给出函数 `oneMore` 的定义，该函数包含一个 `int` 类型的数组形参。函数对数组中的每个元素进行加 1 操作。可以根据需要为该函数添加其他必要的形参。
- 考虑如下的函数定义：

```

void too2(int a[], int howMany)
{
    for (int index = 0; index < howMany; index++)
        a[index] = 2;
}

```

下面的函数调用中，哪些是合法的？

```

int myArray[29];
too2(myArray, 29);
too2(myArray, 10);
too2(myArray, 55);
"Hey too2. Please come over here."
int yourArray[100];
too2(yourArray, 100);
too2(myArray[3], 29);

```

- 在所有可以成为常量数组参数的数组参数前面添加 `const` 修饰符。

```

void output(double a[], int size);
// 前提条件：数组 a 中 a[0] 到 a[size-1] 的元素已被初始化。
// 运行结果：输出 a[0] 到 a[size-1] 的值。

void dropOdd(int a[], int size);
// 前提条件：元素 a[0] 到 a[size-1] 已被初始化。
// 运行结果：从 a[0] 到 a[size-1] 的元素中，所有奇数值的元素都改为 0。

```

- 编写一个名为 `outOfOrder` 的函数，该函数包含一个 `double` 类型的数组参数和一个名为 `size` 的 `int` 类型参数，并返回一个 `int` 类型的值。该函数用来检查数组中是否有顺序不当的地方，即是否违反了下面的条件：

```
a[0] <= a[1] <= a[2] <= ...
```

如果数组元素符合以上条件，函数返回 -1；否则返回第一个违反该条件的变量的索引值。例如，假定数组 `a` 的声明如下：

```
double a[10] = {1.2, 2.1, 3.3, 2.5, 4.5,
               7.9, 5.4, 8.7, 9.9, 1.0};
```

数组 `a` 中，第一个违反条件的是 `a[2]` 和 `a[3]`，且 `a[3]` 是第一个出错的索引变量，因此函数返回 3。如果该数组顺序满足之前的条件，则返回 -1。

## 5.3 用数组编程

永远不要相信整体印象，我的孩子，请专注于细节方面。

柯南·道尔爵士，《身份之谜》（福尔摩斯）

本节介绍部分填充的数组，并将简单介绍数组的排序和查询。本节没有介绍 C++ 语言的新知识，但包含许多 C++ 数组参数的实际应用。

### 部分填充的数组

编写程序时，往往不清楚程序所需数组的确切大小，或者说所需数组的大小在程序每次运行时都不同。解决该问题的一个简单办法就是将数组的声明大小设为所有可能大小的最大值。这样程序就可以根据需要自由使用数组了。

部分填充的数组在使用时需多加留意。程序必须记录数组的当前使用情况以避免引用未曾初始化的索引变量。实例 5.5 的程序对这一点作了解释。该程序读入一系列高尔夫球赛的得分，并显示各个得分与平均分之间的差值。程序可以处理最少 1 个得分，最多 10 个得分。所有的得分都存储在数组 `score` 中，`score` 有 10 个索引变量，但程序只使用实际需要的那些索引变量。变量 `numberUsed` 存储当前使用的索引变量数，这样当前使用的索引变量为 `score[0]` 到 `score[numberUsed-1]`。特别指出的是，如果变量 `numberUsed` 为数组的声明大小，那么当前使用的就是整个数组。另外，由于变量 `numberUsed` 记录了数组的使用情况，它往往也是操作部分填充数组函数的一个实参。由于实参 `numberUsed` 可以保证函数不会引用未初始化的索引变量，因此有些时候就没有必要再用一个实参来说明数组的大小。例如，函数 `showDifference` 和 `computeAverage` 均使用 `numberUsed` 来保证函数使用的都是合法的索引变量。但是函数 `fillArray` 需要知道数组的声明大小以使得不会溢出数组。

**提示：**不要吝啬形参的使用

注意实例 5.5 中的函数 `fillArray`，调用它时，数组的声明大小 `MAX_NUMBER_SCORES` 是它的实参，如下所示：

```
fillArray(score, MAX_NUMBER_SCORES, numberUsed);
```

你也许认为 `MAX_NUMBER_SCORES` 是一个全局定义的常量，那么就可以直接在函数 `fillArray` 中使用它，而不用作为实参。这种想法也许是正确的，如果除了实例 5.5 中的程序之外不会再使用该函数，那么这样做是没有问题的。但是 `fillArray` 是一个使用非常广泛的函数，将会应用在不同的程序中。事实上，下一节的实例 5.6 中我们

将再次使用 `fillArray` 函数。实例 5.6 中，表示数组声明大小的是另外一个不同名字的全局常量。因此，如果直接在 `fillArray` 函数体中使用 `MAX_NUMBER_SCORES`，那么示例 5.6 中就不能使用 `fillArray` 函数了。

即使 `fillArray` 函数只在一个程序中使用，最好也将数组的声明大小设为 `fillArray` 的一个实参，这样就可以很明确地表示该函数的调用需要数组声明大小的相关信息。■

### 示例 5.5 部分填充的数组

---

```

1 // 显示各次高尔夫比赛得分与平均分的差距。

2 #include <iostream>
3 using namespace std;
4 const int MAX_NUMBER_SCORES = 10;

5 void fillArray(int a[], int size, int& numberUsed);
6 // 前提条件：size 为数组 a 的声明大小。
7 // 运行结果：numberUsed 表示数组 a 当前存储了多少数。
8 // 使用键盘输入的非负值来填充数组 a[0] 到 a[numberUsed-1]。

9 double computeAverage(const int a[], int numberUsed);
10 // 前提条件：
11 // a[0] 到 a[numberUsed-1] 都被初始化。numberUsed > 0
12 // 返回元素 a[0] 到 a[numberUsed -1] 的平均值。

13 void showDifference(const int a[], int numberUsed);
14 // 前提条件：数组 a 的前 numberUsed 个元素已被初始化。
15 // 运行结果：
16 // 在屏幕上分别输出数组 a 前 numberUsed 个元素与平均值的差值。
17 int main()
18 {
19     int score[MAX_NUMBER_SCORES], numberUsed;
20     cout << "This program reads golf scores and shows\n"
21          << "how much each differs from the average.\n";

22     cout << "Enter golf scores:\n";

23     fillArray(score, MAX_NUMBER_SCORES, numberUsed);
24     showDifference(score, numberUsed);

25     return 0;
26 }

27 void fillArray(int a[], int size, int& numberUsed)
28 {
29     cout << "Enter up to " << size << " nonnegative whole numbers.\n"
30          << "Mark the end of the list with a negative number.\n";
31     int next, index = 0;
32     cin >> next;
33     while ((next >= 0) && (index < size))
34     {
35         a[index] = next;

```

```

36         index++;
37         cin >> next;
38     }
39     numberUsed = index;
40 }

41 double computeAverage(const int a[], int numberUsed)
42 {
43     double total = 0;
44     for (int index = 0; index < numberUsed; index++)
45         total = total + a[index];
46     if (numberUsed > 0)
47     {
48         return (total/numberUsed);
49     }
50     else
51     {
52         cout << "ERROR: number of elements is 0 in computeAverage.\n"
53              << "computeAverage return 0. \n";
54         return 0;
55     }
56 }

57 void showDifference(const int a[], int numberUsed)
58 {
59     double average = computeAverage(a, numberUsed);
60     cout << "Average of the " << numberUsed
61          << " scores = " << average << endl
62          << "The scores are:\n";
63     for (int index = 0; index < numberUsed; index++)
64         cout << a[index] << " differs from average by"
65              << (a[index] - average) << endl;
66 }

```

### 示例运行结果

```

This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1
Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333

```



### 示例：查询数组

查询数组是一项经常要做的工作。比如某个数组记录了选修某门课程的所有学生的学号，如果想知道某学生是否选修了这门课，只需要查询数组中有没有该学生的学号即可。示例 5.6 中的程序首先填充一个数组，然后在该数组中查询用户设定的值。当然，示例 5.6 中的程序是一个演示程序，比较简单，真正比较复杂的程序中将会表达得更加精细，不过该示例足以让我们了解顺序查找算法的核心思想了。顺序查找是最直接的查询算法：程序从头至尾一次查看数组中的元素，看数组中是否有等于被查找值的元素。

示例 5.6 中，函数 `search` 用于在数组中查询。在进行查询时，我们不仅仅是想了解这个数组中是否含有我们要找的目标，如果有的话，我们还想知道具体是哪个索引变量。因此一般来讲，函数 `search` 的返回值为目标值在数组中的索引值。如果数组中没有我们要找的目标，那么 `search` 函数返回 -1。让我们再进一步了解函数 `search` 的细节吧。

函数 `search` 利用一个 `while` 循环来依次查看变量数组的元素，查看其是否为要找的目标值。变量 `found` 用来标记是否找到了目标值，如果找到了，`found` 为 `true`，并终止 `while` 循环。

### 示例 5.6 数组的查询

```

1 // 查询一个非负整数构成的部分填充数组。
2 #include <iostream>
3 using namespace std;
4 const int DECLARED_SIZE = 20;

5 void fillArray(int a[], int size, int& numberUsed);
6 // 前提条件：参数 size 是数组 a 的声明大小。
7 // 运行结果：
8 // 参数 numberUsed 为数组 a 中最终存储的元素个数。
9 // a[0] 到 a[numberUsed-1] 存储了从键盘读入的非负整数。

10 int search(const int a[], int numberUsed, int target);
11 // 前提条件：numberUsed <= 数组 a 的声明大小，
12 // 且 a[0] 到 a[numberUsed-1] 已经被初始化。
13 // 如果存在的话，返回第一个满足 a[index] == target 的索引，
14 // 否则返回 -1。

15 int main()
16 {
17     int arr[DECLARED_SIZE], listSize, target;

18     fillArray(arr, DECLARED_SIZE, listSize);

19     char ans;
20     int result;
21     do

```

```

22     {
23         cout << "Enter a number to search for: ";
24         cin >> target;

25         result = search(arr, listSize, target);
26         if (result == -1)
27             cout << target << " is not on the list.\n";
28         else
29             cout << target << " is stored in array position"
30                 << result << endl
31                 << "(Remember: The first position is 0.)\n";
32         cout << "Search again?(y/n followed by Return): ";
33         cin << ans;
34     } while ((ans != 'n') && (ans != 'N'));
35     cout << "End of program.\n";
36     return 0;
37 }

38 void fillArray(int a[], int size, int& numberUsed)
39 <函数fillArray的定义已经在示例5.5中给出。>
40 int search(const int a[], int numberUsed, int target)
41 {
42     int index = 0;
43     bool found = false;
44     while(!found && (index < numberUsed))
45         if (target == a[index])
46             found = true;
47         else
48             index++;
49     if (found)
50         return index;
51     else
52         return -1
53 }

```

### 示例运行结果

```

Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.

```

---

### 示例：给数组排序

给数组排序恐怕是应用最广泛、研究得最透彻的一项程序任务。比如将一组销售数组从高到低或者从低到高排序，或是将一组单词按照字母顺序进行排序等。下面将要介绍的是一个名为 `sort` 的函数，该函数可以对一个部分填充的数组按从小到大的顺序进行排序。

函数 `sort` 包含一个数组参数 `a`，该数组 `a` 是一个部分填充的数组，因此还需要一个形参 `numberUsed` 来记录当前使用了多少数组单元。函数 `sort` 的声明以及前提条件如下所示：

```
void sort(int a[], int numberUsed);  
// 前提条件: numberUsed <= 数组 a 的声明大小。  
// 元素 a[0] 到 a[numberUsed-1] 已被初始化。
```

函数 `sort` 对数组 `a` 中的元素进行重新排序，函数 `sort` 调用完成之后，数组 `a` 中的元素将满足如下关系：

```
a[0] <= a[1] <= a[2] <= ... <= a[numberUsed - 1]
```

这里我们用到的排序算法为选择排序，该算法是所有排序算法中最容易理解的一种。

可以通过问题的定义来设计算法。本例要解决的问题是对一个数组按照从小到大的顺序排序，也就是排序完成后 `a[0]` 的值最小，`a[1]` 次之，依次类推。因此，相应的选择排序算法应该为：

```
for (int index = 0; index < numberUsed; index++)  
    Place the indexth smallest element in a[index]
```

选择排序有多种实现方式，比如可以再定义一个数组，将要排序数组中的值按照要排序的顺序复制到另一个数组中。实际上，一个数组就可以实现选择排序，并且占用更少的内存。因此，这里函数 `sort` 只用了一个数组，该数组包含的是待排序的值，函数 `sort` 通过数组元素的交换来完成相关的排序工作。接下来，我们通过一个具体的例子来解释这其中的整个过程。

考虑示例 5.7 中的数组，算法首先将最小的值放到 `a[0]` 中，由于当前最小的数在 `a[3]` 中，所以算法将 `a[0]` 和 `a[3]` 进行交换。接下来看次小的值，次小的值在剩下的 `a[1]`，`a[2]`， $\dots$ ，`a[9]` 中，具体来讲是 `a[5]`，因此算法再次交换 `a[1]` 和 `a[5]` 的值，示例 5.7 第四行的图和第五行的图显示了这一过程。然后算法又开始寻找第三个小的值，依次类推，一直到排序完成。在这一过程中，数组前面的元素始终是正确排序的值，然后从数组后面尚未排序的部分挑选最小的值，添加在已排序部分的后面。注意，数组的最后一个元素不需要做任何处理，只要其他所有元素排序完成，`a[9]` 就是整个数组中的最大的值。

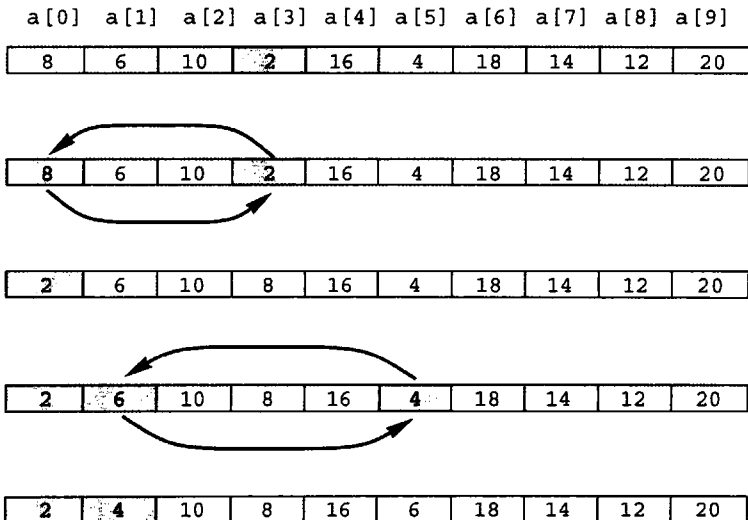
示例 5.8 给出了函数 `sort` 的定义，以及相关的示例程序。函数 `sort` 使用 `indexOfSmallest` 函数去寻找未排序部分中最小的元素索引，然后通过交换将其移动到数组的已排序部分。

示例 5.8 中的 swapValues 函数用于交换索引变量的值，例如如下的代码将交换 a[0] 和 a[3] 的值：

```
swapValues(a[0], a[3]);
```

在第 4 章中曾经介绍过 swapValues。

### 示例 5.7 选择排序



### 示例 5.8 数组排序

```

1 // 测试函数 sort。
2 #include <iostream>
3 using namespace std;

4 void fillArray(int a[], int size, int& numberUsed);
5 // 前提条件：参数 size 是数组 a 的声明大小。
6 // 运行结果；
7 // 参数 numberUsed 为数组 a 中最终存储的元素个数。
8 // a[0] 到 a[numberUsed-1] 存储了从键盘读入的非负整数。

9 void sort(int a[], int numberUsed);
10 // 前提条件：numberUsed <= 数组 a 的声明大小，
11 // 且元素 a[0] 到 a[numberUsed-1] 已被初始化。
12 // 运行结果：a[0] 到 a[numberUsed-1] 已经被重新排序。

```

```

13 // 满足: a[0] <= a[1] <= ... <= a[numberUsed - 1]。

14 void swapValues(int& v1, int& v2);
15 // 交换变量 v1 和 v2 的值。

16 int indexOfSmallest(const int a[], int startIndex, int numberUsed);
17 // 前提条件: 0 <= startIndex < numberUsed。
18 // 引用的数组元素都有值。
19 // 返回 i, 满足 a[i] 是 a[startIndex], a[startIndex + 1], ..., a[numberUsed - 1]
    // 的最小值。

20 int main()
21 {
22     cout << "This program sorts numbers from lowest to highest.\n";

23     int sampleArray[10], numberUsed;
24     fillArray(sampleArray, 10, numberUsed);
25     sort(sampleArray, numberUsed);

26     cout << "In sorted order the numbers are:\n";
27     for (int index = 0; index < numberUsed; index++)
28         cout << sampleArray[index] << " ";
29     cout << endl;

30     return 0;
31 }

32 void fillArray(int a[], int size, int& numberUsed)
33     < 该函数的定义已经在示例 5.5 中给出。 >

34 void sort(int a[], int numberUsed)
35 {
36     int indexOfNextSmallest;
37     for (int index = 0; index < numberUsed - 1; index++)
38         {/// 将正确的值放入 a[index] 中。
39             indexOfNextSmallest =
40                 indexOfSmallest(a, index, numberUsed);
41             swapValues(a[index], a[indexOfNextSmallest]);
42             //a[0] <= a[1] <= ... <= a[index] 是原数组中最小的一些元素。
43             // 其余的元素仍然在原来的位置。

44         }
45 }

46 void swapValues(int& v1, int& v2)
47 {
48     int temp;
49     temp = v1;
50     v1 = v2;
51     v2 = temp;
52 }

54 int indexOfSmallest(const int a[], int startIndex, int numberUsed)

```

```

55 {
56     int min = a[startIndex],
57         indexOfMin = startIndex;
58     for (int index = startIndex + 1; index < numberUsed; index++)
59         if (a[index] < min)
60         {
61             min = a[index];
62             indexOfMin = index;
63             //min 是 a[startIndex] 到 a[index] 最小的元素。
64         }
65     return indexOfMin;
66 }

```

### 示例运行结果

```

This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90

```

### 自测练习

- 编写一个程序，该程序读入 10 个非负整数到一个名为 `numberArray` 的数组中，然后将这些数输出到屏幕上。本题不需要引入任何函数，程序越简单越好。
- 编写一个程序，该程序读入 10 个字母到一个名为 `letterBox` 的数组中，然后以与输入相反的顺序将这些字母输出到屏幕上。例如，如果输入为：

abcd

那么运行结果就为：

dcba

其中采用句点标识输入的结束。该练习无须使用任何函数，程序越简单越好。

- 以下是示例 5.6 中函数 `search` 的另一种定义。要使用该定义，需要对程序做些小小的改动。但是，本练习只需给出 `search` 函数的定义即可。

```

bool search(const int a[], int numberUsed,
            int target, int& where);

// 前提条件：numberUsed <= 数组 a 的声明大小。
// 元素 a[0] 到 a[numberUsed-1] 已被初始化。
// 运行结果：如果 target 是 a[0] 到 a[numberUsed-1] 中的值，函数返回 true，
// 并将 where 设为：a[where] == target。
// 否则函数返回 false，where 的值不变。

```

## 5.4 多维数组

C++ 允许声明带有多个索引的数组。本节将介绍这种多维数组。

### 多维数组基础

#### 数组声明

有时一个数组有多个索引会比较方便，同时 C++ 也支持这种用法。下面的语句声明了一个名为 `page` 的字符数组。数组 `page` 有两组索引，第一组为 0 ~ 29，第 2 组为 0 ~ 99。

```
char page[30][100];
```

#### 索引变量

该数组的索引变量都含有两个索引，例如 `page[0][0]`、`page[15][32]` 以及 `page[29][99]` 等。注意各个索引都应该放在各自的方括号中。和一维数组一样，多维数组中各个索引变量的类型也都是数组的基本类型。

一个数组可以有多个索引，但是最常用的一般只有两组。一个二维数组可以看作是一个二维的显示，其中第一组索引为行，第二组索引为列。例如，二维数组 `page` 可以表示为：

```
page[0][0], page[0][1], ..., page[0][99]
page[1][0], page[1][1], ..., page[1][99]
page[2][0], page[2][1], ..., page[2][99]
.
.
.
page[29][0], page[29][1], ..., page[29][99]
```

可以用数组 `page` 来存储一页书上的所有文字。其中，该页书包含 30 行文字，每行 100 个字符。

C++ 中，像 `page` 这样的二维数组其实可以看作是数组的数组。上面的 `page` 数组可以看作是一个一维数组，它的声明大小为 30，基本类型是大小为 100 的一维数组。在实际的应用中，一般只需将它看成是一个有两组索引的数组（而不是数组的数组）。但是存在一种特殊情况，也就是当函数的数组参数是一个二维数组的时候，将该二维数组看作是数组的数组可能更有助于理解。

### 多维数组的声明

#### 语法

```
Type Array_Name [ Size_Dim_1 ][ Size_Dim_2 ]...[ Size_Dim_Last ];
```

#### 示例

```
char page[30][100];
int matrix[2][3];
double threeDPicture[10][20][30];
```

上面给出的数组声明将为每一个索引的组合定义一个对应的索引变量，例如上面第二个声明将会为数组 `matrix` 定义如下六个索引变量：

```
matrix[0][0], matrix[0][1], matrix[0][2],
matrix[1][0], matrix[1][1], matrix[1][2]
```

## 多维数组参数

下面二维数组的定义实际上是定义了一个声明大小为 30 的一维数组，且其基本类型是大小为 100 的一维数组。

```
char page[30][100];
```

将二维数组理解为数组的数组将有助于理解 C++ 是如何处理多维数组的。

例如，下面的函数带有一个像 page 的多维数组参数，它将数组的内容输出到屏幕上：

```
void displayPage(const char p[][100], int sizeDimension1)
{
    for (int index1 = 0; index1 < sizeDimension1; index1++)
    { // 输出数组的每一行。
        for (int index2 = 0; index2 < 100; index2++)
            cout << p[index1][index2];
        cout << endl;
    }
}
```

注意，函数中的二维数组参数没有给出其第一维的大小，因此专门有一个 int 类型的参数来给出第一维的大小（如普通一维数组一样，也可以在方括号内填入一个数字来指定第一维的大小，但该数字只起注释作用，编译器会将其忽略）。数组第二维的大小（及其他维的大小）在数组参数后给出，如下所示：

```
const char p[][100]
```

如果将多维数组理解为数组的数组，那么就可以这样来理解。既然二维数组参数

```
const char p[][100]
```

是一个数组的数组，那么其第一维便是数组的索引，与一个普通一维数组的索引一样。而第二维部分描述了该一维数组的基本类型，也就是大小为 100 的字符数组。

### 多维数组参数

当函数头或者函数声明中包含多维数组参数时，数组第一维的大小不会给出，但其余维的大小必须写在相应的方括号中。由于第一维没有给出，因此一般还需要一个 int 类型的参数来专门给出第一维的大小。下面是带有一个二维数组参数 p 的函数声明：

```
void getPage(char p[][100], int sizeDimension1);
```

## 示例：使用二维数组的记分程序

示例 5.9 给出了一个使用二维数组 grade 存储并显示班级测验得分情况的程序。班级只包含四个学生，共有三次测验。示例 5.10 说明了数组 grade 是如何保存成绩的。数组的第一维用来表示学生的学号，第二维用来表示一次测验号。由于学号和测验都是从 1 开始的，而不是 0，因此，学号和测验号都必须减去 1 才能得到某学生某



次测验分数的索引。例如, 学号为4的学生在第一次测验中的分数存放在索引变量 `grade[3][0]` 中。

程序还使用了两个普通的一维数组。数组 `stAve` 用来记录每个学生各次测验的平均分。例如, 索引变量 `stAve[0]` 的值为学生1各次测验的平均分, `stAve[1]` 为学生2各次测验的平均分, 依次类推。数组 `quizAve` 用来记录每次测验的平均分, 例如 `quizAve[0]` 保存第一次测验的平均分, `quizAve[1]` 保存第二次测验的平均分, 依次类推。示例 5.10 展示数组 `grade`、`stAve` 以及 `quizAve` 之间的关系, 并给出了一个示例的 `grade` 数组。数组 `grade` 中的值又决定了程序存储在 `stAve` 和 `quizAve` 中的值。示例 5.11 给出了这些示例数据, 程序采用这些示例数据来计算 `stAve` 和 `quizAve` 的值。

示例 5.9 给出了整个过程的完整程序, 该程序首先填充数组 `grade`, 然后分别计算学生的平均分和测验的平均分, 并分别存储在数组 `stAve` 和 `quizAve` 中。程序将数组的维数声明为全局的名字常量。因为该函数仅仅应用在示例 5.9 的程序中, 不会应用到其他地方, 因此我们在函数体中使用了这些全局定义的常量, 而不是将数组维数作为函数的参数。由于填充数组的代码是常规程序, 因此本例没有给出它的定义。

### 示例 5.9 二维数组

```
1 // 将每个学生的各次测验得分读入到一个二维数组 grade 中 (本例没有给出读入代码)。
2 // 计算每个学生的平均分以及各测验的平均分。
3 // 显示测验得分和平均分。
4 #include <iostream>
5 #include <cmath>
6 using namespace std;
7 const int NUMBER_STUDENTS=4, NUMBER_QUIZZES=3;
8 void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[]);
9 // 前提条件: 全局命名常量 NUMBER_STUDENTS 和 NUMBER_QUIZZES 是数组 grade 的维数。
10 // 索引变量 grade[stNum-1, quizNum-1] 存储的是
11 // 编号为 stNum 的学生第 quizNum 次测验的得分。
12 // 运行结果: stAve[stNum-1] 存储的是编号为 stNum 的学生的平均分。
13
14 void computeQuizAve(const int grade[][NUMBER_QUIZZES],
15                     double quizAve[]);
16 // 前提条件: 全局命名常量 NUMBER_STUDENTS
17 // 和 NUMBER_QUIZZES 是数组 grade 的维数。
18 // 索引变量 grade[stNum-1, quizNum-1] 存储的是
19 // 编号为 stNum 的学生第 quizNum 次测验的得分。
20 // 运行结果: quizAve[quizNum-1] 存储的是第 quizNum 次测验的平均分。
21
22 void display(const int grade[][NUMBER_QUIZZES],
23             const double stAve[],
24             const double quizAve[]);
25 // 前提条件: 全局命名常量 NUMBER_STUDENTS
26 // 和 NUMBER_QUIZZES 是数组 grade 的维数。
27 // 索引变量 grade[stNum-1, quizNum-1] 存储的是
28 // 编号为 stNum 的学生第 quizNum 次测验的得分。
29 // stAve[stNum-1] 存储的是编号为 stNum 的学生的平均分。
```

```

27 // quizAve[quizNum-1] 存储的是第 quizNum 次测验的平均分。
   // 运行结果：输出数组 grade、stAve 以及 quizAve 的内容。

28 int main()
29 {
30     int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
31     double stAve[NUMBER_STUDENTS];
32     double quizAve[NUMBER_QUIZZES];
33
   <填充数组的代码在这里，本例没有给出。>
34
35     computeStAve(grade, stAve);
36     computeQuizAve(grade, quizAve);
37     display(grade, stAve, quizAve);
38     return 0;
39 }
40 void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[])
41 {
42     for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
43     { // 每次处理一个学生。
44         double sum = 0;
45         for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
46             sum = sum + grade[stNum-1][quizNum-1];
47         // sum 中记录每个学生的总分。
48         stAve[stNum-1] = sum/NUMBER_QUIZZES;
49         // 学生 stNum 的平均分为 stAve[stNum-1]。
50     }
51 }

52 void computeQuizAve(const int grade[][NUMBER_QUIZZES],
   double quizAve[])
53 {
54     for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
55     { // 每次处理一个测验。
56         double sum = 0;
57         for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
58             sum = sum + grade[stNum-1][quizNum-1];
59         // sum 记录每次测验的总分。
60         quizAve[quizNum-1] = sum/NUMBER_STUDENTS;
61         // 第 quizNum 次测验的平均分为 quizAve[quizNum-1]。
62     }
63 }

64 void display(const int grade[][NUMBER_QUIZZES],
   const double stAve[], const double quizAve[])
65 {
66     cout.setf(ios::fixed);
67     cout.setf(ios::showpoint);
68     cout.precision(1);
69
70     cout << setw(10) << "Student"
71          << setw(5) << "Ave"
72          << setw(15) << "Quizzes\n";

```

```

73     for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
74     { // 每次显示一个学生。
75         cout << setw(10) << stNum
76             << setw(5) << stAve[stNum-1] << " ";
77         for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
78             cout << setw(5) << grade[stNum-1][quizNum-1];
79         cout << endl;
80     }

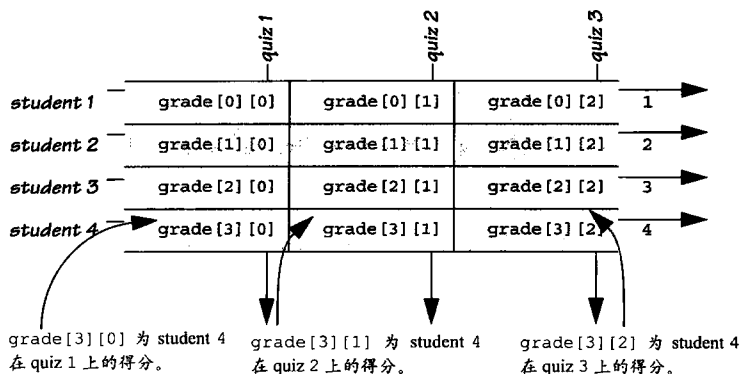
81     cout << "Quiz averages = ";
82     for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
83         cout << setw(5) << quizAve[quizNum-1];
84     cout << endl;
85 }

```

### 示例运行结果

Student	Ave	Quizzes		
1	10.0	10	10	10
2	1.0	2	0	1
3	7.7	8	6	9
4	7.3	8	4	10
Quiz Average =		7.0	5.0	7.5

### 示例 5.10 二维数组 grade



示例 5.11 二维数组 grade

	quiz 1	quiz 2	quiz 3		
student 1	10	10	10	→ 10.0	stAve[0]
student 2	2	0	1	→ 1.0	stAve[1]
student 3	8	6	9	→ 7.7	stAve[2]
student 4	8	4	10	→ 7.3	stAve[3]
	↓	↓	↓		
quizAve	7.0	5.0	7.5		
	quizAve[0]	quizAve[1]	quizAve[2]		

## 自测练习题

20. 给出以下代码的输出。

```
int myArray[4][4], index1, index2;
for (index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 4; index2++)
        myArray[index1][index2] = index2;
for (index1 = 0; index1 < 4; index1++)
{
    for (index2 = 0; index2 < 4; index2++)
        cout << myArray[index1][index2] << " ";
    cout << endl;
}
```

21. 编写一段填充数组 a 的代码，数组 a 的声明如下所示。从键盘输入一些整数，每行输入 5 个数，一共 4 行（不过你编写的代码不应依赖于具体的输入方式）。

```
int a[4][5];
```

22. 为 void 函数 echo 编写相关的函数定义。使得以下的函数调用能输出数组 a（参见自测练习题 21）的内容，并且以输入格式相同的方式输出（每行 5 个数，一共 4 行）。

```
echo(a, 4);
```

## 本章小结

- ◆ 数组可以用来存储和处理一组相同类型的数据。
- ◆ 数组索引变量与同类型普通变量的用法完全相同。
- ◆ for 循环非常适合用来遍历数组元素，并对每个元素做相应的处理。
- ◆ 使用数组编程时，最常碰到的错误就是引用一个不存在的索引变量。要仔细检查遍历数组的循环，确保数组索引没有越界。
- ◆ 数组参数既不是值传递调用参数，也不是引用传递调用参数，它是一种新的参数类型。数组参数类似引用传递调用；如果函数体修改了数组参数的内容，那么该修改在调用时会体现在数组实参上。
- ◆ 数组的索引变量依次存储于计算机的内存中，因此它占用的是内存中的一部分连续空间。当数组被作为函数实参时，传递给该函数的只是第一个索引变量的地址，因此函数通常还包含一个 int 类型的形参来存储数组的大小。
- ◆ 使用部分填充数组时，通常需要一个 int 类型的参数来记录当前数组被使用的元素个数。
- ◆ 在数组前面添加 const 修饰符，就可以指示编译器该数组不能被更改，这样的数组被称为常量数组参数。
- ◆ 如果需要多组索引的数组，可以使用多维数组，我们可以将其理解为数组的数组。

## 自测练习题答案

1. 语句 `int a[5]` 是一个声明，其中 5 是数组的声明大小。`a[4]` 是数组 `a` 中的元素，它是索引为 4 的索引变量，也就是数组的第 5 个元素。
2. a. score  
b. double  
c. 5  
d. 0 ~ 4  
e. `score[0]`、`score[1]`、`score[2]`、`score[3]`、`score[4]` 中的任一个
3. a. 多了一个初始值。  
b. 正确，数组的大小为 4。  
c. 正确，数组的大小为 4。
4. abc
5. 1.1 1.2 2.2 3.3  
1.1 3.3 3.3
6. 0 2 4 6 8 10 12 14 16 18  
0 4 8 12 16

7. 数组 `sampleArray` 的索引变量包括 `sampleArray[0] ~ sampleArray[9]`, 但这段代码试图填充的却是 `sampleArray[1] ~ sampleArray[10]`, 其中 `sampleArray[10]` 导致数组访问越界。

8. 数组访问越界。当 `index` 为 9 时, `index+1` 为 10, 因此 `a[index+1]` 就等于 `a[10]`, 导致数组访问越界。进行修改时, 只需将 `for` 循环的第一行改为:

```
for (int index = 0; index < 9; index++)
```

```
9. int i, a[20];
   cout << "Enter 20 numbers:\n";
   for (i = 0; i < 20; i++)
       cin >> a[i];
```

10. 该数组将占用 14 个字节的内存, 索引变量 `yourArray[3]` 的地址为 1006。

11. 以下函数调用合法:

```
tripler(a[2]);
tripler(a[number]);
tripler(number);
```

以下函数调用非法:

```
tripler(a[3]);
tripler(a);
```

第一个函数调用包含非法索引, 第二个函数调用没有索引表达式, 在第二个函数调用中, 不能将数组名作为函数 `tripler` 的实参。本章的“整个数组作为函数实参”小节中讨论了可以将整个数组用作实参的一种情况, 但使用方式与此完全不同。

12. 循环遍历索引变量 `b[1]~b[5]`, 但对于数组 `b` 来讲, `b[5]` 是一个非法的索引变量。数组 `b` 的索引包含 0、1、2、3、4。正确的代码如下:

```
int b[5] = {1, 2, 3, 4, 5};
for (int i = 0; i < 5; i++)
    tripler(b[i]);
```

```
13. void oneMore (int a[], int size)
    // 前提条件: size 是数组 a 的声明大小, a[0] ~ a[size-1] 都已被初始化。
    // 运行结果: 数组 a 中的所有索引变量都递增 1。
    {
        for (int index = 0; index < size; index++)
            a[index] = a[index] + 1;
    }
```

14. 以下的函数调用合法:

```
too2(myArray, 29);
too2(myArray, 10);
too2(yourArray, 100);
```

函数调用:

```
too2(myArray, 10);
```

虽然合法, 但只能填充数组 `myArray` 的前 10 个索引变量。如果本来就是这样的意图, 那么这么调用是没有问题的。

以下的函数调用非法：

```
too2(myArray, 55);
"Hey too2. Please come over here."
too2(myArray[3], 29);
```

第一个函数调用非法的原因在于第二个参数太大，第二个调用非法的原因在于缺少末位的分号；第三个调用非法的原因在于应该使用整个数组的地方，用的却是数组的索引变量。

15. 由于 output 函数不会改变数组参数的内容，因此可以将函数中的数组参数声明为常量参数。但是 dropOdd 函数有可能会改变数组参数的内容，因此不可以将其数组参数声明为常量参数。

```
void output(const double a[], int size);
// 前提条件：a[0] ~ a[size-1] 都已被初始化。
// 运行结果：a[0] ~ a[size-1] 都已被输出。

void dropOdd(int a[], int size);
// 前提条件：a[0] ~ a[size-1] 都已被初始化。
// 运行结果：a[0] ~ a[size-1] 中的所有奇数都被改为 0。
```

```
16. int outOfOrder(double array[], int size)
{
    for (int i = 0; i < size - 1; i++)
        if (array[i] > array[i+1]) // 对每个 i，获取对应的 a[i+1]。
            return i+1;
    return -1;
}
```

```
17. #include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;
int main()
{
    cout << "Enter up to ten nonnegative integers.\n"
          << "Place a negative number at the end.\n";
    int numberArray[DECLARED_SIZE], next, index = 0;
    cin >> next;
    while ( (next >= 0) && (index < DECLARED_SIZE) )
    {
        numberArray[index] = next;
        index++;
        cin >> next;
    }
    int numberUsed = index;
    cout << "Here they are back at you:";
    for (index = 0; index < numberUsed; index++)
        cout << numberArray[index] << " ";
    cout << endl;
    return 0;
}
```

```
18. #include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;
```

```

int main( )
{
    cout << "Enter up to ten letters"
          << " followed by a period:\n";
    char letterBox[DECLARED_SIZE], next;
    int index = 0;
    cin >> next;
    while ( (next != '.') && (index < DECLARED_SIZE) )
    {
        letterBox[index] = next;
        index++;
        cin >> next;
    }
    int numberUsed = index;
    cout << "Here they are backwards:\n";
    for (index = numberUsed-1; index >= 0; index--)
        cout << letterBox[index];
    cout << endl;
    return 0;
}

19. bool search(const int a[], int numberUsed,
               int target, int& where)
{
    int index = 0;
    bool found = false ;
    while ((!found) && (index < numberUsed))
    {
        if (target == a[index])
            found = true ;
        else
            index++;
        // 如果没有找到目标, 那么
        // found == true && a[index] == target.
        if (found)
            where = index;
    }
    return found;
}

20. 0 1 2 3
    0 1 2 3
    0 1 2 3
    0 1 2 3

21. int a[4][5];
    for (index1, index2;
        for (index1 = 0; index1 < 4; index1++)
            for (index2 = 0; index2 < 5; index2++)
                cin >> a[index1][index2];

22. void echo(const int a[][5], int sizeOfa)
    // 输出数组 a 中的值, sizeOfa 行, 且每行 5 个数字。
{
    for (int index1 = 0; index1 < sizeOfa; index1++)
    {
        for (int index2 = 0; index2 < 5; index2++)
            cout << a[index1][index2] << " ";
    }
}

```



```

        cout << endl;
    }
}

```

## 编程练习

1. 编写程序，读入某城市一年中各月份的平均降雨量以及最近 12 个月每个月实际的降雨量，程序以表格的形式显示最近 12 个月实际的降雨量以及与平均降雨量的差异。月平均降雨量以 1 月、2 月，……的顺序给出。为了得到最近 12 个月的实际降雨量，程序首先询问当前的月份，然后要求输入最近 12 个月的降雨量。程序的运行结果中要求给出月份。

可以采用多种方式来处理月份名称。一种最直接的方法是对月份进行编号，在最后输出时做依次转换即可。输出函数中可以使用大的 switch 语句；而月份的输入可以采取任何形式，但必须方便、易操作。

完成上面的程序后，对程序做进一步改进，要求能够以图形的方式显示最近 12 个月的平均降雨量和实际降雨量。输出的方式类似示例 5.4 中的图，只不过每个月对应两个条形图，分别标记为平均降雨量和实际降雨量。程序应询问用户最终结果的显示方式，然后根据用户的选择来展示最终结果。设计一个循环使程序一直运行，直到用户要求终止程序。

2. 编写一个名为 deleteRepeats 的函数，该函数包含一个部分填充的字符数组作为形参，函数将删除该数组中重复出现的元素。由于部分填充的数组要求两个参数，因此该函数实际上包含两个形参：一个数组参数和一个 int 类型的表示数组使用情况的形参。删除一个元素后，后续的元素会前移，这使得数组最后一个位置空缺。由于函数采用了部分填充数组，第二个参数就用来表示有多少个数组单位正在被使用。第二个参数应该是一个引用参数，删除一个重复元素后它的值会改变，从而反映数组的使用情况。例如，考虑如下的代码：

```

int a[10];
a[0] = 'a';
a[1] = 'b';
a[2] = 'a';
a[3] = 'c';
int size=4;
deleteRepeats(a, size);

```

以上代码运行结束后，a[0] 的值将为 'a'，a[1] 的值为 'b'，a[2] 的值为 'c'，size 的值为 3（这里不再关心 a[3] 的值，因为部分填充数组不再使用该索引变量）。本题目假定数组中只包含小写字母。编写程序对函数进行测试。

3. 标准差是衡量一组数字偏离平均值的指标。标准差越小，说明这组数字越是紧密的分布在平均数的周围；反之，则越是偏离平均数。 $N$  个数  $x_i$  的标准差  $S$  定义如下：

$$S = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N}}$$

其中  $\bar{x}$  是这  $N$  个数的平均值。定义一个函数, 以一个部分填充的数字数组为参数, 返回这组数的标准差。使用部分填充数组需提供两个参数, 因此函数实际上有两个形参: 一个数组参数和一个 `int` 类型的记录部分填充数组使用情况的形参。数组中元素的类型是 `double` 类型。最后, 编写一个完整的程序对函数进行测试。

4. 编写一个程序, 该程序读入一个 `int` 类型的数组, 并对该数组的内容进行统计, 这里我们假定数组的声明大小小于 50。程序的输出为两列: 第一列是数组中出现的不同元素, 第二列是对应的元素在数组中出现的次数。最终的输出应该按照第一列进行排序, 从大到小。

例如, 对于:

```
-12 3 -12 4 1 1 -12 1 -1 1 2 3 4 2 3 -12
```

对应的输出为:

N	Count
4	2
3	3
2	2
1	4
-1	1
-12	4

5. 可以采用数组来存储大整数, 其中每个数组元素存储大整数的一个数字。例如, 整数 1234 可以这样存储在数组 `a` 中: `a[0]` 中存储 1, `a[1]` 中存储 2, `a[2]` 中存储 3, `a[3]` 中存储 4。但通过本练习你会发现, 通过倒序的方式存储大整数其实更方便, 也就是 `a[0]` 中存储 4, `a[1]` 中存储 3, `a[2]` 中存储 2, `a[3]` 中存储 1。编写程序, 读入两个最多 20 位的正整数, 计算并输出这两者的和。程序在读入大整数时, 将各个数字看作是 `char` 类型, 如 1234 可以分 '1'、'2'、'3'、'4' 四个字符读入。程序读入这些字符之后, 会将其转化为对应的 `int` 类型, 这些大整数各位上的数字将存储在一个部分填充的数组中。接下来, 程序执行普通的算数加法操作, 结果存储在一个大小为 20 的数组中, 并最终输出到屏幕上。如果计算结果的位数大于结果数组的大小 (即大于 20 位), 程序将输出一个“整数溢出”的警告。程序应该设置一个全局变量来记录大整数的最高位数, 从而用户可以很方便地改变位数的最大值。设计一个循环, 使得用户可以不断进行加法运算, 直到用户终止程序。
6. 跳水比赛中, 七位裁判给出一个位于 0 ~ 10 之间的分数, 分数有可能是小数。去掉得分中的最高分和最低分, 余下的分数相加, 并将结果乘以该跳水的难度系数, 然后再将计算出来的结果乘以 0.6, 才得出该运动员最后的得分。其中难度系数位于 1.2 ~ 3.8 之间。

编写程序，读入跳水的难度系数和七位裁判的打分，计算并输出该运动员最后的得分。注意，程序应该对输入数据的合法性进行检查。

7. 对某班学生的一次测验结果，生成一个基于文本的直方图。测试的成绩范围：0 ~ 5。编写一个程序，读入用户输入的测试成绩，在用户输入的同时使用数组分别统计得分为 0、1、2、3、4、5 的学生个数。程序应能处理任意数量的学生成绩。

可以通过一个大小为 6 的数组来完成上述任务，数组中的每个元素都初始化为 0。当输入的测试成绩为 0 时，就增加索引变量 `array[0]`；当输入为 1 时就增加索引变量 `array[1]`；依此类推，直到 `array[5]`。

最后输出直方统计图，例如，对于输入 3, 0, 1, 3, 3, 5, 5, 4, 5, 4，程序应该输出：

```
1 grade(s) of 0
1 grade(s) of 1
0 grade(s) of 2
3 grade(s) of 3
2 grade(s) of 4
3 grade(s) of 5
```

8. 生日悖论是说在同一个房间中存在两个生日相同的人的概率惊人的高，这里的生日，仅仅指生日中的天，不包括生日中具体的某一年或者一天中具体的某个时刻。编写程序，估算同一个房间中两人生日相同的概率，这里假设房间中的人数为 2 ~ 50。

程序应该以模拟的方式给出一个估计值，在多次实验中（如 5000 次），随机赋予房间中每个人一个生日，然后统计两人生日相同的次数，之后除以实验次数，从而得出相应的概率。程序的输出类似如下，不过其中的数字会因随机而不同：

```
For 2 people, the probability of two birthdays is about 0.002
For 3 people, the probability of two birthdays is about 0.0082
For 4 people, the probability of two birthdays is about 0.0163
...
For 49 people, the probability of two birthdays is about 0.9654
For 50 people, the probability of two birthdays is about 0.969
```

9. 编写程序实现 tic-tac-toe 游戏。程序轮流询问玩家 X 和玩家 O 如何走，程序显示的游戏界面如下：

```
1  2  3
4  5  6
7  8  9
```

玩家通过输入对应位置的数字来标记他们的走位，每走一步，程序将显示更改过的界面，例如：

```
x  x  o
4  5  6
```

0 8 9

10. 编写程序安排飞机上乘客的座位。假设一架小型飞机上的座位示意图如下：

```

1   A B C D
2   A B C D
3   A B C D
4   A B C D
5   A B C D
6   A B C D
7   A B C D

```

程序用 X 标记已经预订了的座位，例如，座位 1A、2B 以及 4C 被预订之后，飞机座位的示意图如下：

```

1   X B C D
2   A X C D
3   A B C D
4   A B X D
5   A B C D
6   A B C D
7   A B C D

```

程序显示当前可预订的座位之后，提示用户输入想预订的座位，然后可预订的座位再次被更新，一直到所有座位都被预订，或者用户结束程序为止。如果用用户输入的座位已经被预订，程序将提示这个座位已经被预订，请重新选择。

11. 编写程序，接受如示例 5.4 所示的输入，输出类似的直方图，只不过该直方图是纵型的，而不是示例 5.4 那样的横型直方图。程序的实现可以采用二维数组。
12. 数学家约翰·何顿·康威发明了“生命游戏”，虽然从传统角度来看，或许不能称为游戏。该游戏虽然只有为数不多的一些规则，但却能呈现出很有趣的行为。本题目要求你编写一个程序，程序会要求你设定一个初始配置。程序会遵循“生命”的演化规则，从而展示初始配置带来的一连串行为。

“生命”是一个孤立的、生活在二维“世界”中的有机体。虽然理论上讲，该二维世界是无限的；不过，这里我们用一个  $80 \times 22$  的数组来模拟该世界。如果你电脑的屏幕足够大，你尽管使用更大的数组来表示该世界。

这里说的世界用一个二维数组来表示，其中的每一个索引变量都可以存储一个“生命”。“世代”表示时间的流逝，每个“世代”都带给“生命”群体新的诞生和死亡。其中，“生命”的诞生和死亡符合如下的规则：

- (1) 每个“生命”包含八个邻居“生命”，包括上、下、左、右和四个角上的“生命”。
- (2) 如果一个“生命”包含一个或者 0 个邻居，那么它将孤独而死。如果一个“生命”含有三个以上的邻居，那么它将因过度拥挤而死。
- (3) 如果一个没有被“生命”占据的索引变量刚好含有三个“生命”邻居，那么该索引变量将诞生一个新的“生命”。

- (4) 诞生和死亡的发生都是瞬间的, 并且发生在“世代”转换之时。无论是由于何种原因而死亡的“生命”, 均可能会引起“生命”的诞生。但新生成的“生命”却不能让即将死亡的“生命”复活。同样, 一个“生命”的死亡也不会阻止另外一个“生命”的死亡。

\*

实例: \*\*\* 变为 \*, 然后重新变为 \*\*\*, 如此循环。

\*

注意, 一些配置是从一些相对较小的初始配置发展而来的, 而其他的配置则是从另一个区域转移过来的。考虑到文字输出的方便性, 这里采用一个 char 类型、大小为  $80 \times 22$  的数组来存放“生命”群体的每一个“世代”。采用“\*”来表示一个活着的“生命”, 而采用空格来表示一个空的或者死亡的“生命”。如果你电脑的屏幕足够大, 那么尽可能使用更大的数组。

建议: 寻找稳定的配置, 也就是说, 寻找那些不断循环出现的“世代”。循环中出现的配置的数目被称为周期。此外, 还存在一些固定的配置, 也就是说完全不会发生变化的配置。可以编写程序来找出此类的配置。

提示: 定义一个名为 generation 的 void 函数, 该函数唯一的参数就是用 char 类型数组表示的世界, 也就是初始配置。函数然后扫描整个“世界”, 并对“世界”中的每个索引变量进行修改, 根据之前给出的规则来诞生或者终结“生命”。这其中包括, 按顺序检查“世界”中的每个索引变量, 从而决定“生命”的诞生或者死亡。编写一个名为 display 的函数, 该函数包含一个表示“世界”(world) 的数组, 并在屏幕上显示该“世界”的内容。generation 和 display 的调用之间可以存在一定的时间延迟。为此, 只有当你按回车键时, 你的程序才生成并显示下一个“世代”的“生命”。你也可以将该程序改造为一个全自动的程序, 不过本题目并不要求这一点。

13. 孩子们经常玩一个比赛记忆力的游戏, 该游戏使用一般的扑克牌。例如, 给六张牌, 其中的两张标记为“1”, 还有两张标记为“2”, 最后的两张标记为“3”。6 张牌打乱之后正面朝下扣在桌面上, 现在游戏开始。选手选择两张正面朝下的牌, 翻转过来, 如果这两张牌上的标记匹配, 那么继续选择其他的牌; 如果标记不匹配, 则将这两张牌重新正面朝下扣在桌面上。游戏按照这种方式一直进行, 直到所有的牌都正面朝上。

编写一个程序来玩这个游戏。这次采用 16 张牌, 排列成一个  $4 \times 4$  的方阵, 分别采用数字 1 ~ 8, 给八对牌标上记号。程序应该允许用户通过坐标来指定具体的扑克牌。例如, 对于如下的纸牌摆放:

	1	2	3	4
1	8	*	*	*
2	*	*	*	*
3	*	8	*	*
4	*	*	*	*

除了标记为“8”的纸牌对（坐标位置分别为（1，1）和（2，3））外，所有的纸牌都正面朝下摆放。

提示：采用一个二维数组来记录纸牌的布局，采用另外一个二维数组来记录纸牌是正面朝上还是朝下。编写一个函数来模拟纸牌的重新洗牌，这通过随机选择两张纸牌并交换它们的位置来实现。

14. 假设你从四个编号为 0 ~ 3 的影评人员那里收集了电影的评论，每个影评人员对六部编号为 100 ~ 105 的电影进行打分。打分的分值范围为 1（很差）~ 5（很好）。

最终的打分情况如下所示：

	100	101	102	103	104	105
0	3	1	5	2	1	5
1	4	2	1	4	2	4
2	3	1	2	4	4	1
3	5	1	4	2	4	2

编写一个程序，采用一个二维数组来存储该打分数据。程序允许用户对任意三部电影输入一个打分。基于该打分数据，程序之后找到打分情况最接近于用户输入的影评人员。然后通过输出该影评人员对其他电影的评分情况来预测用户的兴趣。程序采用笛卡尔距离来计算用户输入的打分与影评人员给出的打分之间的距离。本题目采用的原理是最近邻算法的一个简化版本。

例如，假设用户对电影 102 的评分为 5，对电影 104 的评分为 2，对电影 105 的评分为 5。那么与该用户评分距离最近的影评人员为 0 号，其距离为： $\sqrt{(5-5)^2 + (2-1)^2 + (5-5)^2} = 1$ 。那么该程序给出的最终预测是：打分为 3 的电影 100、打分为 1 的电影 101 以及打分为 2 的电影 103。

15. 传统的密码设计方案很容易被别人“偷窥”，攻击者通过查看一个没有防备的用户输入自己的密码口令或者 PIN 口令，然后使用该口令来获取账户的信息。解决该问题的一个方法是使用一个随机的应答验证系统。该系统运行时，用户根据自己的口令每次都输入不同的信息来响应系统给出的询问信息。考虑如下的设计方案，其中密码为 5 位的数字（00000 ~ 99999），每一位都有一个对应的随机数字，1、2 或者 3。这一过程中，用户输入的是自己口令对应的随机数，而不是口令本身。

例如，考虑如下的口令 12345，为了验证用户的身份，屏幕上首先会显示如下信息：

```
PIN:  0 1 2 3 4 5 6 7 8 9
NUM:  3 2 3 1 1 3 2 2 1 3
```

根据该对应信息，用户此时应该输入 23113，而不是自己的口令 12345。采用这种方式后，即使攻击者截获到了用户输入的 23113，用户的密码也不会泄露，因为 23113 也可以对应其他的密码，如 69440 或者 70439。下次用户登录系统时，

会显示另外一个不同的随机生成的序列，例如：

```
PIN:  0 1 2 3 4 5 6 7 8 9
NUM:  1 1 2 3 1 2 2 3 3 3
```

你编写的程序应该模拟用户的认证过程，并采用数组来存储数字 0 ~ 9 对应的数字序列。用户的实际口令存储在程序中，程序首先输出生成的随机数字序列，然后读取用户的输入，最后输出用户是否认证成功的信息。

16. 重新完成第 14 题，不过这次可能有多达 1000 位的影评人员，对应的影评存放在一个文本文件中。虽然编写的程序可以支持多达 1000 位的影评人员，但实际的影评人员会从 1 位到 1000 位不等。评论的电影和第 14 题相同，编号分别为 100 ~ 105。你可以设计文本文件的具体格式来存放不同的影评。程序首先从文本文件中加载所有的影评，然后读取用户输入的三部电影的影评，根据之前给出的方法预测用户喜欢的其他电影。
17. 本书第 2 章的编程练习 12 要求用户研究本福特定律。一个简单的编写程序的方法是采用一个数组来存储数字的出现次数。也就是，count[0] 存储数字 0 出现在第一位的次数，count[1] 存储数字 1 出现在首位的次数，依此类推。采用数组重做该章的编程练习 12。
18. 本题目是第 4 章编程练习 16 的扩展。考虑一个名为 scores.txt 的文本文件，该文件用来存储游戏玩家的得分情况。该文件的一个示例如下面所示，其中 Ronaldo 的最高得分为 10400，Didier 的最高得分为 9800，等等。在该文件中保存至少五位游戏选手的得分：

```
Ronaldo
10400
Didier
9800
Pele
12300
Kaka
8400
Cristiano
8000
```

编写一个名为 getHighScores 的函数，该函数含有两个数组参数：一个字符串数组和一个整数数组。函数首先扫描给出的文本文件，然后将字符串数组的第一个索引变量设置为得分最高的玩家名字，将整数数组的第一个索引变量设置为该玩家的最高得分，将字符串数组的第二个索引变量设置为得分次高的玩家名字，将整数数组的第二个索引变量设置为该选手的最高得分，数组的第三个元素类似。最后，这两个数组给出前三名选手的名字和得分情况。编写完整的程序对 getHighScores 函数进行测试。

# 结构体和类

## 6.1 结构体 196

结构体类型 198

陷阱：漏掉结构体定义末尾的分号 200

结构体作为函数参数 201

提示：使用多重结构体 201

结构体的初始化 204

## 6.2 类 206

定义类和成员函数 206

封装 211

公有成员和私有成员 211

取值和赋值函数 214

提示：接口和实现的分离 216

提示：封装的测试 216

结构体与类 217

提示：对象思考 217



## 第6章 结构体和类

“这个时刻来临了。”海象先生说，  
“去讲述众多的事情：  
讲述鞋子、船只，还有封蜡，  
去讲述卷心菜，还有国王们。”  
刘易斯·卡罗尔，《爱丽丝镜中奇遇记》

### 概述

类的概念可能是 C++ 语言与 C 语言最大的差别。类是对象的模板，同一个类的所有对象具有相同的成员函数和成员变量结构。对象包含数据和成员函数，成员函数可以访问该对象的数据。面向对象编程是一种非常流行且十分强大的编程思想，其中类对象是面向对象编程思想最重要的概念。

结构体

我们将分两步来介绍类，首先介绍结构体的定义。结构体可以看作是没有任何成员函数的对象。<sup>1</sup> 结构体最重要的特性是，它可以包含各种不同类型的数据。学会如何定义结构体之后，就可以很容易地掌握类的定义。

本章的学习不需要第 5 章的内容。另外，第 7 章和第 8 章也有对类的介绍。

### 6.1 结构体

我不在乎属于任何一个会接受我作为会员的俱乐部。

格劳乔·马克思，《格劳乔的信件》

有时，将不同数据类型放到一起，看作一个单一的数据会非常有用。例如，考虑银行存款单的例子（英文缩写 CD），存款单在规定的存期内不能提款。一个存款单一般包含如下三部分数据：账户余额、账户利率以及存期（以月计）。前两部分数据可以用 double 类型表示，而存期可以用 int 类型表示。示例 6.1 给出了一个名为 CDAccountV1 的结构体来表示存款单（V1 代表版本 1，因为后续会重新定义这个结构体的新版本）。

#### 示例 6.1 结构体定义

```
1 // 演示 CDAccountV1 结构体的程序。
2 #include <iostream>
3 using namespace std;
```

<sup>1</sup> C++ 中的结构体其实可以包含函数，但一般不会这么使用，本章后续的内容会对此做详尽介绍。

```

4  // 用于表示银行存款单的结构。
5  struct CDAccountV1
6  {
7      double balance;
8      double interestRate;
9      int term; // 存款期限。
10 };

11 void getData(CDAccountV1& theAccount);
12 // 运行结果: theAccount.balance、theAccount.interestRate
13 // 以及 theAccount.term 已经被设置为用户键盘输入的值。

14 int main()
15 {
16     CDAccountV1 account;
17     getData(account);

18     double rateFraction, interest;
19     rateFraction = account.interestRate/100.0;
20     interest = account.balance*(rateFraction*(account.term/12.0));
21     account.balance = account.balance+interest;

22     cout.setf(ios::fixed);
23     cout.setf(ios::showpoint);
24     cout.precision(2);
25     cout << "When your CD matures in "
26           << account.term << " months,\n"
27           << "it will have a balance of $"
28           << account.balance << endl;

29     return 0;
30 }
31 // 使用 iostream.
32 void getData(CDAccountV1& theAccount)
33 {
34     cout << " Enter account balance: $";
35     cin >> theAccount.balance;
36     cout << "Enter account interest rate: ";
37     cin >> theAccount.interestRate;
38     cout << "Enter the number of months until maturity: ";
39     cin >> theAccount.term;
40 }

```

### 示例运行结果

```

Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity: 6
When your CD matures in 6 months,
it will have a balance of $105.00

```

---

## 结构体类型

示例 6.1 中的结构体定义如下：

```
struct CDAccountV1
{
    double balance;
    double interestRate;
    int term;
};
```

struct  
结构体标记

结构体成员

在哪里放置  
结构体定义

其中，关键字 `struct` 声明紧接着的代码是一个结构体定义。标识符 `CDAccountV1` 是该结构体的名字，又称**结构体标记**。结构体标记可以为任何合法的（除了关键字之外）的标识符。尽管 C++ 没有做相关的要求，但一般结构体标记都以大写字母开头。大括号 “{}” 内部的变量为**结构体成员**。如上面的例子所示，结构体定义以一个 `}` 和分号结束。

结构体的定义一般放在所有函数定义之外（和全局常量的定义一样）。这样，全局定义的结构体就可以用在之后所有的代码中。

一旦结构体被定义，那么就可以如同那些预定义类型（如 `int`、`char` 等）一样来使用。注意，示例 6.1 中，结构体类型 `CDAccountV1` 用来在 `main` 函数中声明了一个名为 `account` 的变量，同时也作为函数 `getData` 的参数类型。

成员值

结构体变量的使用与其他普通变量类似，只不过它包含了一批不同类型的**成员值**。结构体定义中的每个成员都有一个值，例如结构体变量 `account` 就包含三个成员值：两个 `double` 类型和一个 `int` 类型。这些成员值作为一个整体，定义了结构体变量的值，后续章节对此会进行相关介绍。

结构体值

成员变量

每个结构体类型都定义了一系列成员名字。示例 6.1 中的结构体 `CDAccountV1` 就包含三个成员名字：`balance`、`interestRate` 以及 `term`。每个成员名字都代表整个结构体变量中的一部分，我们称其为**成员变量**。成员变量可以通过结构体名字加上点运算符再跟上成员名字来访问。例如，示例 6.1 中的结构体变量 `account` 包含如下三个成员变量：

```
account.balance
account.interestRate
account.term
```

前两个成员变量的类型为 `double`，最后一个成员变量的类型为 `int`。如示例 6.1 所示，这些成员变量的用法与其他变量没有什么不同。例如，下面这行代码将成员变量 `account.balance` 和普通变量 `interest` 的值相加，然后再赋值给成员变量 `account.balance`。

```
account.balance = account.balance + interest;
```

重用成员  
名字

不同的结构体可以使用相同的成员名字。例如，在同一个程序中包含下面两个结构体的定义是合法的。

```
struct FertilizerStock
{
    double quantity;
```

```
double nitrogenContent;
};
```

和

```
struct CropYield
{
    int quantity;
    double size;
};
```

### 点运算符

点运算符用来引用结构体变量的成员变量。

#### 语法

`structure_Variable_Name.Member_Variable_Name`

点运算符



#### 示例

```
struct StudentRecord
{
    int studentNumber;
    char grade;
};

int main( )
{
    StudentRecord yourRecord;
    yourRecord.studentNumber = 2001;
    yourRecord.grade = 'A';
}
```

一些书中也将点运算符称为结构体成员访问符，本书不使用该说法。

这种成员名字的重合不会产生任何问题。例如，声明了如下两个结构体变量：

```
FertilizerStock superGrow;
CropYield apples;
```

此时，`superGrow.quantity` 和 `apples.quantity` 其实是两个不同的变量，点运算符之前的结构体名字对其做了区分。

赋值语句  
中的结构  
体变量

一个结构体值可以看作是一系列成员值的集合，同样也可以看作是一个复杂的由多个成员值组成的单个值。因此，结构体值和结构体变量的使用与简单值和简单变量的用法相同。特别指出的是，我们可以用等号为结构体变量赋值。例如，如果 `apples` 和 `oranges` 都是类型为 `CropYield` 的结构体变量，那么如下的语句是没有任何问题的：

```
apples = oranges;
```

该语句等价于：

```
apples.quantity = oranges.quantity;
apples.size = oranges.size;
```

## 结构体类型

我们可以如下定义一个结构体，其中 *Structure\_Tag* 是结构体的名字。

### 语法

```
struct Structure_Tag
{
    Type_1 Member_Variable_Name_1;
    Type_2 Member_Variable_Name_2;
    .
    .
    .
    Type_Last Member_Variable_Name_Last;
};
```

←————— 不要忘记结尾的分号。

### 示例

```
struct Automobile
{
    int year;
    int doors;
    double horsePower;
    char model;
};
```

可以将相同类型的成员变量放在一行，通过逗号隔开。例如，下面的结构体定义与上面的定义完全相同：

```
struct Automobile
{
    int year, doors;
    double horsePower;
    char model;
};
```

结构体变量的声明与一般变量的声明完全相同，例如：

```
Automobile myCar, yourCar;
```

成员变量的引用是通过点运算符进行的。例如，*myCar.year*、*myCar.doors*、*myCar.horsePower* 以及 *myCar.model*。

## 陷阱：漏掉结构体定义末尾的分号

结构体定义中，最后的“}”括号容易让人认为结构体定义已经结束，但并非如此，必须在结束括号“}”后面加上分号。C++ 语言这样设计是有原因的，是为了实现语言的一些特性，尽管到目前为止我们还用不到这样的特性。结构体在定义的同时也可以进行结构体变量的声明。可以通过在结束括号“}”和分号之间添加一系列结构体变量的名字来完成结构体变量的声明。例如，下面的代码定义了一个名为 *WeatherData* 的结构体，同时也声明了该结构体的两个变量，即 *dataPoint1* 和 *dataPoint2*：

```
struct WeatherData
{
    double temperature;
    double windVelocity;
} dataPoint1, dataPoint2; ■
```

### 结构体作为函数参数

#### 结构体参数

一个函数可以包含结构体类型的传值调用参数，或者结构体类型的引用调用参数，或者两者都包含。示例 6.1 中的程序包含一个名为 `getData` 的函数，该函数就含有一个结构体 `CDAccountV1` 类型的引用调用参数。

#### 函数返回 结构体

结构体类型还可以作为函数的返回值类型。例如，以下的函数包含一个 `CDAccountV1` 类型的参数，并最终返回另一个 `CDAccount1` 类型的结构体变量。返回的结构体变量与传入的结构体变量拥有相同的账户余额和存期，但其利率是传入结构体的两倍。

```
CDAccountV1 doubleInterest(CDAccountV1 oldAccount)
{
    CDAccountV1 temp;
    temp = oldAccount;
    temp.interestRate = 2*oldAccount.interestRate;
    return temp;
}
```

注意代码中类型为 `CDAccountV1` 的局部变量 `temp`，它用来返回函数最终的处理结果。如果 `myAccount` 是一个类型为 `CDAccountV1` 的结构体变量，并且其成员变量已经做了相关的初始化，那么下面的代码将为 `yourAccount` 赋值，其利率为 `myAccount` 的两倍。

```
CDAccountV1 yourAccount;
yourAccount = doubleInterest(myAccount);
```

### 提示：使用多重结构体

在某些情况下，一个结构体的成员有可能是另外一个结构体。例如，一个名为 `PersonInfo` 的结构体可以用来保存一个人的身高、体重和出生日期，它的定义如下：

```
struct Date
{
    int month;
    int day;
    int year;
};

struct PersonInfo
{
    double height; // 以英尺为单位。
    int weight; // 以英镑为单位。
    Date birthday;
};
```

声明一个 `PersonInfo` 类型的结构体变量如下：

```
PersonInfo person1;
```

如果结构体变量 `person1` 中的出生日期变量已被初始化，那么输出一个人的出生年份可以采用如下的语句：

```
cout << person1.birthday.year;
```

该语句采用从左到右的结合顺序。我们从最左边开始，`person1` 是一个 `PersonInfo` 类型的结构体变量。为了引用该结构体变量的数据成员 `birthday`，可以采用如下的语句：

```
person1.birthday
```

而该成员变量本身也是一个 `Date` 类型的结构体变量。因此，该成员变量自身也包含成员变量。为了访问成员变量 `person1.birthday` 自身的成员变量。我们采用如下的方式：

```
person1.birthday.year
```

示例 6.2 对示例 6.1 中的存款单结构体做了重写。新版本采用了一个名为 `Date` 的成员变量来记录存款期限，同时采用两个成员变量来表示该账户的余额情况：初始余额与到期余额。■

## 示例 6.2 包含结构成员的结构

```
1 // 演示存款单结构体的程序。
2 #include <iostream>
3 using namespace std;

4 struct Date
5 {
6     int month;
7     int day;
8     int year;
9 };

10 // 存款单结构体的改进版本。
11 struct CDAccount
12 {
13     double initialBalance;
14     double interestRate;
15     int term; // 存款的期限。
16     Date maturity; // 存款到期的日期。
17     double balanceAtMaturity;
18 };

19 void getCDDData(CDAccount& theAccount);
20 // 运行结果：
21 //theAccount.initialBalance、theAccount.interestRate、
22 //theAccount.term 和 theAccount.maturity 的值更新为用户键盘的输入值。
23
24 void getDate(Date& theDate);
```

该存款单结构体是示例 6.1 中的改进版本。

```

25 // 运行结果:
26 //theDate.month, theDate.day 和 theDate.year 更新为用户的输入值。

27 int main()
28 {
29     CDAccount account;
30     cout << "Enter account data on the day account was opened:\n";
31     getCDData(account);
32     double rateFraction, interest;
33     rateFraction = account.interestRate / 100.0;
34     interest = account.initialBalance*(rateFraction*(account.term /12.0));
35     account.balanceAtMaturity = account.initialBalance + interest;

36     cout.setf(ios::fixed);
37     cout.setf(ios::showpoint);
38     cout.precision(2);
39     cout << "When the CD matured on "
40         << account.maturity.month << "-" << account.maturity.day
41         << "-" << account.maturity.year << endl
42         << "it had a balance of $"
43         << account.balanceAtMaturity << endl;
44     return 0;
45 }
46 //使用 iostream.
47 void getCDData(CDAccount& theAccount)
48 {
49     cout << "Enter account initial balance: $";
50     cin >> theAccount.initialBalance;
51     cout << "Enter account interest rate: ";
52     cin >> theAccount.interestRate;
53     cout << "Enter the number of months until maturity: ";
54     cin >> theAccount.term;
55     cout << "Enter the maturity date:\n";
56     getDate(theAccount.maturity);
57 }

58 //使用 iostream.
59 void getDate(Date& theDate)
60 {
61     cout << "Enter month: ";
62     cin >> theDate.month;
63     cout << "Enter day: ";
64     cin >> theDate.day;
65     cout << "Enter year: ";
66     cin >> theDate.year;
67 }

```

### 示例运行结果

```

Enter account data on the day account was opened:
Enter account initial balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity: 6
Enter the maturity date:
Enter month: 2

```



```

Enter day: 14
Enter year: 1899
When the CD matured on 2-14-1899
it had a balance of $105.00

```

---

## 结构体的初始化

可以在结构体声明的时候对其初始化。为初始化结构体变量，可以在结构体变量的后面跟上等号以及用大括号括起来的一系列成员值来实现。例如，下面的语句定义了前面提到的结构体 `Date`。

```

struct Date
{
    int month;
    int day;
    int year;
};

```

一旦定义好了 `Date` 结构体，我们就可以采用如下的语句声明一个名为 `dueDate` 的结构体变量，并对其做初始化。

```
Date dueDate = {12, 31, 2012};
```

初始化值的顺序必须与结构体成员变量定义的顺序一致。在本例中，成员变量 `dueDate.month` 得到的初始值是 12，`dueDate.day` 得到初始化中的第二个值 31，而 `dueDate.year` 得到第三个初始化值 2012。

如果给出的初始化值多于结构体成员的数量，那么就会出错；而初始值少于结构体成员的数量则是允许的。结构体首先按照顺序对成员变量做初始化；没有初始化值的结构体成员则被初始化为对应类型的默认值。

## 自测练习

1. 考虑如下的结构体定义和结构体变量声明。

```

struct CDAccountV2
{
    double balance;
    double interestRate;
    int term;
    char initial1;
    char initial2;
};
CDAccountV2 account;

```

下面各项分别是什么数据类型？如果有错误，请指出。

- a. `account.balance`
- b. `account.interestRate`
- c. `CDAccountV1.term`
- d. `account.initial2`
- e. `account`

2. 给出如下的类型定义。

```
struct ShoeType
{
    char style;
    double price;
};
```

考虑以下代码的输出是什么？

```
ShoeType shoe1, shoe2;
shoe1.style = 'A';
shoe1.price = 9.99;
cout << shoe1.style << " $" << shoe1.price << endl;
shoe2 = shoe1;

shoe2.price = shoe2.price/9;
cout << shoe2.style << " $" << shoe2.price << endl;
```

3. 下面的结构体定义有何问题？

```
struct Stuff
{
    int b;
    int c;
}
int main( )
{
    Stuff x;
    // 其他代码
}
```

4. 给出如下的结构体定义。请声明一个该结构体类型的变量 *x*，并将成员变量 *b* 和 *c* 分别初始化为 1 和 2。

```
struct A
{
    int member b;
    int member c;
};
```

5. 下面是一些结构体的初始化语句。指出其初始化后的结果是什么。如果有错误，请指出。

```
struct Date
{
    int month;
    int day;
    int year;
};
```

- a. Date dueDate = {12, 21};
- b. Date dueDate = {12, 21, 1995};
- c. Date dueDate = {12, 21, 19, 95};

6. 请定义一个名为 `EmployeeRecord` 的结构体，该结构体用来记录员工的工资、年假天数以及员工的状况（小时工或者正式工）。其中员工状况用一个字符表示，'H'（小时工）或者'S'（正式工）。
7. 根据如下的函数声明，给出该函数的定义。其中 `ShoeType` 结构体已经在自测练习题 2 中给出。

```
void readShoeRecord(ShoeType& newShoe);
// 从键盘输入数值，并赋给 newShoe 的各成员。
```

8. 根据如下的函数声明，给出该函数的定义。其中 `ShoeType` 结构体已经在自测练习题 2 中给出。

```
ShoeType discount(ShoeType oldRecord);
// 返回一个与传入参数一致的结构，但其价格比传入的少 10%。
```

## 6.2 类

我们都知道——这个时代知道，只是我们装作不知道。

弗吉尼亚·伍尔夫，《周一或周二》

类和结构体类似，可以把它看作是同时包含了成员变量和成员函数的结构体，它是面向对象编程思想中最重要的概念。

### 定义类和成员函数

**类** 类和结构体非常类似，但类除了包含成员变量外，还包含成员函数。示例 6.3 给出了一个简单的名为 `DayOfYear` 的类的定义。这个类包含一个名为 `output` 的成员函数以及两个成员变量：`month` 和 `day`。关键字 `public` 是访问修饰符。它表明对其后成员的访问没有任何限制。我们将在后面详细介绍 `public` 以及其他访问修饰符。该类用来表示一个日期（如 1 月 1 日或者 7 月 4 日）。

**对象** 类类型变量的值一般被称为对象（或者不严格地来讲，类类型的变量一般被称为对象）。一个对象包含数据成员和函数成员。当采用类进行编程时，一个程序可以看作是许多相互作用的对象的集合。对象之所以能够交互，是因为对象拥有行为。对象的行为是通过调用对象的成员函数来体现的。类变量的声明方式和其他预定义类型的变量的声明方式以及结构体变量的声明方式完全一致。

**成员函数** 如果忽略示例 6.3 中的关键字 `public`，类 `DayOfYear` 的定义和结构体的定义非常类似（除了采用关键字 `class` 来代替结构体定义中的关键字 `struct` 以及类中包含了成员函数 `output` 之外）。需要注意的是，类中的成员函数 `output` 仅仅是一个函数声明。一般来讲，类的定义仅仅包含成员函数的声明，成员函数的定义则放在类定义之外的其他地方。C++ 的类定义中，可以任意安排成员函数和成员变量的顺序，但本书倾向于将成员函数放在成员变量之前。

### 示例 6.3 带有成员函数的类

```

1  // 包含一个非常简单的类的演示程序。
2  // 类 DayOfYear 的改进版本将在示例 6.4 中给出。
3  #include <iostream>
4  using namespace std;

5  class DayOfYear
6  {
7  public:
8      void output(); ← 成员函数声明
9      int month;
10     int day;
11 };

12 int main()
13 {
14     DayOfYear today, birthday;
15     cout << "Enter today's date:\n";
16     cout << "Enter month as a number: ";
17     cin >> today.month;
18     cout << "Enter the day of the month: ";
19     cin >> today.day;
20     cout << "Enter your birthday:\n";
21     cout << "Enter month as a number: ";
22     cin >> birthday.month;
23     cout << "Enter the day of the month: ";
24     cin >> birthday.day;
25     cout << "Today's date is ";
26     today.output(); ← 调用成员函数进行输出
27     cout << endl;
28     cout << "Your birthday is ";
29     birthday.output(); ← 调用成员函数进行输出
30     cout << endl;

31     if (today.month == birthday.month && today.day == birthday.day)
32         cout << "Happy Birthday!\n";
33     else
34         cout << "Happy Unbirthday!\n";
35     return 0;
36 }

37 // 使用 iostream。 ← 成员函数定义
38 void DayOfYear::output()
39 {
40     switch (month)
41     {
42     case 1:
43         cout << "January "; break;
44     case 2:
45         cout << "February "; break;
46     case 3:
47         cout << "March "; break;
48     case 4:
49         cout << "April "; break;

```

通常来讲，成员变量应该是私有的，而不是公有的。本章的后续小节会对此做详细介绍。

```

50         case 5:
51             cout << "May "; break;
52         case 6:
53             cout << "June "; break;
54         case 7:
55             cout << "July "; break;
56         case 8:
57             cout << "August "; break;
58         case 9:
59             cout << "September "; break;
60         case 10:
61             cout << "October "; break;
62         case 11:
63             cout << "November "; break;
64         case 12:
65             cout << "December "; break;
66         default:
67             cout << "Error in DayOfYear::output.";
68     }
69
70     cout << day;
71 }

```

### 示例运行结果

```

Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is October 15
Your birthday is February 21
Happy Unbirthday!

```

对象成员变量的引用与结构体成员变量的引用是相同的，都是采用点运算符来完成的。例如，给定示例 6.3 中类 `DayOfYear` 的对象 `today`，那么该对象的两个成员变量分别为：`today.month` 和 `today.day`。

### 调用成员函数

类的成员函数的调用也是通过点运算符来完成的，这与成员变量的访问类似。例如，示例 6.3 声明了类 `DayOfYear` 的两个对象：

```
DayOfYear today, birthday;
```

那么对象 `today` 调用其成员函数 `output` 的方式如下：

```
today.output();
```

对象 `birthday` 调用 `output` 成员函数的方式如下：

```
birthday.output();
```

### 定义成员函数

定义类的成员函数时，必须包含类名，因为不同类的成员函数名字有可能相同。示例 6.3 仅仅包含一个类的定义，但一般情况下，程序中会包含多个类的定义，而且几个类经常包含相同名字的成员函数。示例 6.3 中类 `DayOfYear` 成员函数 `output` 的定

义与一般函数的定义几乎完全相同，唯一的不同在于成员函数的定义前添加有其所属类的名字。

成员函数 `output` 定义的开头如下：

```
void DayOfYear::output()
```

作用域  
运算符

运算符 `::` 被称为作用域运算符，其作用与点运算符类似，而这都是告诉编译器成员函数属于哪个类。不同的是，作用域运算符用在类后面，而点运算符用在对象后面。作用域运算符由两个冒号组成，中间没有空格。作用域运算符前面的类名一般被称为类型限定语，因为它限定了函数所属的类类型。

类型限定语

### 成员函数定义

成员函数的定义与普通函数的定义基本类似，不同之处在于定义成员函数时，必须在函数名前添加类名和作用域运算符。

#### 语法

```
Returned_Type Class_Name:: Function_Name(Parameter_List)
{
    Function_Body_Statements
}
```

#### 示例

参见示例 6.3。需要注意的是，在成员函数内部使用成员变量时，无须在成员变量前使用对象名和点运算符。

函数定义中  
的成员变量

注意示例 6.3 中成员函数 `DayOfYear::output` 的定义，该函数的定义用到了成员变量 `month` 和 `day`，但引用这些成员变量时并没有使用对象名和点运算符。这并不奇怪，由于这里仅仅是定义成员函数，而此时我们并不知道到底哪个对象将调用该函数，因此不能在变量前面使用对象的名字。当调用成员函数时，如：

```
today.output()
```

成员函数中所有用到的成员变量都被确定为当前调用对象的成员变量。因此，上面的函数调用等价于：

```
{
    switch (today.month)
    {
        case 1:
            .
            .
            .
    }

    cout << today.day;
}
```

在成员函数的定义中，可以直接使用所有的成员（包括成员变量和成员函数），而无须使用点运算符。

### 点运算符与作用域运算符

点运算符与作用域运算符均用在成员名前面来指明该成员属于哪个类或者对象。例如,假定声明了一个名为 DayOfYear 的类和一个该类的对象:

```
DayOfYear today;
```

可以使用点运算符来访问对象 today 的成员。例如, output 是类 DayOfYear (在示例 6.3 中定义) 的成员函数,那么下面的语句将输出存储在对象 today 中的数据:

```
today.output();
```

定义成员函数时,可以使用作用域运算符来指明成员函数所属的类。例如成员函数 output 定义中开头的语句如下:

```
void DayOfYear::output( )
```

需要注意的是,作用域运算符和类名一起使用,而点运算符则和对象名一起使用。

### 类是一种完全的类型

类也是一种类型,它的用法和常见的 int、double 类型用法相似。可以使用类来声明变量,也可以作为函数参数类型,或者作为函数返回值类型。更一般地来讲,可以像使用其他一般类型那样来使用类类型。

### 自测练习题

9. 下面重新定义了示例 6.3 中的类 DayOfYear,我们为该类添加了一个名为 input 的成员函数,这里请给出 input 函数的定义。

```
class DayOfYear
{
public:
    void input( );
    void output( );
    int month;
    int day;
};
```

10. 给出如下的类定义,根据该定义,请写出成员函数 set 的定义。

```
class Temperature
{
public:
    void set(double newDegrees, char newScale);
    // 将成员变量设置为参数给出的值。
    double degrees;
    char scale; // 'F' 代表华氏温度, 'C' 代表摄氏温度。
};
```

11. 请详细描述点运算符和作用域运算符之间的异同。

## 封装

数据类型

抽象数据类型

一种数据类型都会有与之对应的值，例如对于 `int` 类型，可以取 0, 1, -1, 2 等值。这或许会让人认为数据类型就是一些值，但实际上作用在这些值之上的操作更重要。没有这些操作，我们不能对这些值做任何处理。针对 `int` 类型，有 +、-、\*、/、% 等操作以及一些预定义的库函数。**数据类型**不仅仅是一些值的集合，它还包括一系列定义在这些值之上的操作。在使用某种数据类型时，如果该数据类型的值以及相关操作的实现对程序员不可见，那么这种数据类型称为**抽象数据类型**（ADT）。C++ 预定义的数据类型（如 `int`）都是抽象数据类型，因为编程人员在使用 `int` 类型时，不知道 +、- 等操作的实现细节，而且就算了解这些操作的实现细节，在编程中也无法使用。类，作为一种程序员自定义的类型，也是抽象数据类型，也就是类的实现细节应该被隐藏起来，类的使用者无须了解类的实现细节。类的操作是通过类的公有函数来体现的，程序员在使用这些类的时候，无须关心成员函数的具体实现，他只须了解类成员函数的声明以及与成员函数相关的描述信息。

类的使用者同样也不需要了解类成员变量的实现细节，如同成员函数的隐藏一样，成员变量的实现细节同样要隐藏起来。实际上，这两者的隐藏几乎没有什么区别。以示例 6.3 中的 `DayOfYear` 为例，编程人员只需知道 `DayOfYear` 是用来存储日期的，至于 `DayOfYear` 的实现是用数据 3 还是用字符串 “March” 保存日期，这个是编程人员无须关心的。

封装

通过类来隐藏成员函数以及数据的实现细节，只暴露相关的使用接口，这种编程方式有许多种称呼。其中最常见地称为**信息隐藏**、**数据抽象**或者**封装**。它们都表示类的实现对于类的使用者是不可见的，是被隐藏起来的。这是面向对象编程思想中最重要的原则之一。在讨论面向对象编程思想时，这一方法一般被称为封装。实现类封装的一种方法就是将类中所有的成员变量声明为私有的，这部分内容将在下一小节中讨论。

## 公有成员和私有成员

回顾示例 6.3 `DayOfYear` 的定义，为了使用该类，编程人员必须知道该类的两个成员变量的类型为 `int`，名字分别为 `month` 和 `day`。而这恰恰违背了面向对象编程思想中的封装原则。示例 6.4 给出了重写后的代码，该代码更好地体现了封装原则。

private:  
私有成员  
变量

注意示例 6.4 中的关键字 `public` 和 `private`，所有出现在 `private` 关键字之后的成员都是私有的，这表示我们不能在类外来访问这些成员，只能在类成员函数的定义中使用这些成员。改写之后的 `DayOfYear` 类，其用法发生了变化。如下所示的语句，这些语句不可以出现在程序的 `main` 函数中（如语句后边的注释所示），不过它们可以出现在类的成员函数中。

```
DayOfYear today; // 合法
today.month = 12; // 非法
today.day = 25; // 非法
cout << today.month; // 非法
```



```

    cout << today.day; //非法
    if (today.month == 1) //非法
        cout << "January";

```

### 示例 6.4 包含私有成员类

本程序是示例 6.3 中 DayOfYear 的改进版本。

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      void input( );
8      void output( );
9      void set(int newMonth, int newDay);
10     // 前提条件: newMonth 和 newDay 可以构成一个合法的日期。
11     void set(int newMonth);
12     // 前提条件: 1 <= newMonth <= 12
13     // 运行结果: 日期被设定为参数给定月份的第一天。

14     int getMonthNumber( );
15     int getDay();
16 private:
17     int month; ← 私有成员变量
18     int day; ← 私有成员变量
19 }

20 int main()
21 {
22     DayOfYear today, bachBirthday;
23     cout << "Enter today's date:\n";
24     today.input( );
25     cout << "Today's date is ";
26     today.output( );
27     cout << endl;

28     bachBirthday.set(3, 21);
29     cout << "J. S. Bach's birthday is ";
30     bachBirthday.output( );
31     cout << endl;
32     if ( today.getMonthNumber( ) == bachBirthday.getMonthNumber( ) &&
33         today.getDay( ) == bachBirthday.getDay( ) )
34         cout << "Happy Birthday Johann Sebastian!\n";
35     else
36         cout << "Happy Unbirthday Johann Sebastian!\n";
37     return 0;
38 }
39
40 // 使用 iostream 和 cstdlib.
41 void DayOfYear::set(int newMonth, int newDay) ← 赋值函数

```

注意，函数 set 是一个重载函数。成员函数的重载与其他一般函数的重载并无二异。

```

42 {
43     if ((newMonth >= 1) && (newMonth <= 12))
44         month = newMonth;
45     else
46     {
47         cout << "Illegal month value! Program aborted.\n";
48         exit(1)
49     }
50     if ((newDay >= 1) && (newDay <= 31))
51         day = newDay;
52     else
53     {
54         cout << "Illegal day value! Program aborted.\n";
55         exit(1);
56     }
57 }

```

```

58 // 使用 iostream 和 cstdlib.
59 void DayOfYear::set(int newMonth) ← 赋值函数
60 {
61     if ((newMonth >= 1) && (newMonth <= 12))
62         month = newMonth;
63     else
64     {
65         cout << "Illegal month value! Program aborted.\n";
66         exit(1)
67     }
68     day = 1;
69 }
70
71 int DayOfYear::getMonthNumber( ) ← 取值函数
72 {
73     return month;
74 }
75
76 int DayOfYear::getDay( ) ← 取值函数
77 {
78     return day;
79 }

```

```

79 // 使用 iostream 和 cstdlib.
80 void DayOfYear::input( )
81 {
82     cout << "Enter the month as a number: ";
83     cin >> month; ← 私有成员变量只可以用在成员函数的定义中，除此之外，都是非法的。
84     cout << "Enter the day of the month: ";
85     cin >> day;
86     if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
87     {
88         cout << "Illegal date! Program aborted.\n";
89         exit(1);
90     }
91 }
92 void DayOfYear::output( )

```

93 < 剩余的定义与示例 6.3 完全相同。 >

### 示例运行结果

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is March 21
J. S. Bach's birthday is March 21
Happy Birthday Johann Sebastian!
```

一旦将一个成员变量定义为私有的，就只能通过相关的成员函数来改变其值（或者引用该成员变量）。这也就意味着，编译器会对类 `DayOfYear` 的实现进行隐藏。此时编译器会对成员变量的使用权限进行检查，如果访问了这些私有的成员变量，程序在编译的时候就会报错。仔细查看示例 6.4 中的程序会发现，成员变量 `month` 和 `day` 仅仅出现在了成员函数的定义中，不存在 `today.month`、`today.day`、`bachBirthday.month` 以及 `bachBirthday.day` 等语句。

#### 公有的

所有出现在 `public` 关键字之后的成员都是公有的，这表示可以在任何地方使用它们，对于公有成员的使用没有任何限制。

#### 公有成员 变量

任何成员变量都可以设置为公有的或者私有的，同样，任何成员函数也可以设置为公有的或者私有的。但一般来讲，通常将成员变量都设置为私有的，而将大多数成员函数都设置为公有的。

类的定义中，可以使用多个 `public`、`private` 关键字，代码中一旦出现

```
public:
```

其后的所有成员都变为公有的，一旦出现

```
private:
```

则其之后的所有成员都变为私有的。类的定义中可以有多组公有、私有成员。但一般是将所有的公有成员放在一组，而将所有的私有成员放在另一组。

类的定义中，公有成员和私有成员顺序的安排没有统一的标准，一般是将公有成员放在最前面，这样便于类的使用者查看该类。你可以自行安排公有成员和私有成员的顺序，本书的示例中都是将公有成员放在私有成员的前面。

从某种角度看，C++ 似乎偏向将私有成员放在最前面，如果最开始的一组成员既没有 `public` 修饰也没有 `private` 修饰，那么该组成员默认为私有。你可能会在一些代码中看到这样的用法，但本书不采用这种用法。

### 取值和赋值函数

一般我们应该将类的成员变量设为私有的。在某些情况下，可能需要对这些成员变量做些操作。虽然类的成员函数也可以对成员变量进行很多操作，但迟早会碰到一些情况，我们需要对成员变量做一些成员函数没有提供的操作，那此时应该怎么做呢？解决的办法就是为该类添加取值函数和赋值函数。取值函数和赋值函数都是类的成员变量，使用它们可以很方便地访问和修改对象中的数据，通过这种方法我们可

**取值函数** 以对成员变量做各种操作。**取值函数**用来获取对象的数据，在示例 6.4 中，成员函数 `getMonthNumber` 和 `getDay` 都是取值函数。取值函数不必返回成员变量的确切数值，但它的返回值必须能反映成员变量的值。例如，对于类 `DayOfYear` 中获取月份的取值函数，通过字符串返回月份的方式就比按数字方式返回该成员变量的方式好。

**赋值函数** **赋值函数**用来修改对象的数据。在示例 6.4 中，名为 `set` 的两个成员函数都是赋值函数。作为一种约定，取值函数的命名总以 `get` 开头，而赋值函数的命名以 `set` 开头（示例 6.4 中的 `input` 函数和 `output` 函数实际上分别是赋值函数和取值函数，但由于 I/O 是种特例，我们常常称之为输入/输出函数，而不是取值、赋值函数）。

类的定义中应该包含一系列合适的取值、赋值函数。

乍一看，类的取值、赋值函数似乎违背了封装性的原则，但实际上并非如此。注意示例 6.4 中的 `set` 赋值函数，该函数不允许用户将成员变量的值设置为 13 或者任何一个不代表月份的值。同样，`set` 赋值函数也不允许将成员变量 `day` 设置为 1 ~ 31 范围之外的值。但是，如果将成员变量设置为公有的，那么就可以很轻易地给成员变量 `day` 设置任意的值（包括那些不合法的数，如 32）。使用赋值函数，用户就可以对数据成员的改变进行相关的控制。

## 自测练习题

12. 假设程序中包含如下的类定义：

```
class Automobile
{
public:
    void setPrice(double newPrice);
    void setProfit(double newProfit);
    double getPrice();
private:
    double price;
    double profit;
    double getProfit();
};
```

并假设程序的 `main` 函数包含如下的声明，而且程序已经对所有的成员变量进行了赋值。

```
Automobile hyundai, jaguar;
```

那么下面那些语句在程序的 `main` 函数中是合法的？

```
hyundai.price = 4999.99;
jaguar.setPrice(30000.97);
double aPrice, aProfit;
aPrice = jaguar.getPrice();
aProfit = jaguar.getProfit();
aProfit = hyundai.getProfit();
hyundai = jaguar;
```

13. 假设将自测练习题 12 中类 `Automobile` 的所有成员变量从私有的改为公有的，那么练习题 12 的答案又是什么？

14. 解释类的定义中, `public` 关键字和 `private` 关键字的区别。
15. a. 一个实际的类中至少需要多少个 `public` 小节?  
b. 一个类中至少需要多少个 `private` 小节?

### 提示：接口和实现的分离

#### 接口 API

封装原则要求类的使用者不需要了解类实现的具体细节。程序员在使用该类的时候, 只需了解类的使用方法即可。这些如何使用类的方法常常被称为接口或者 API。关于 API 究竟代表什么, 有不少争议。但通常来说, API 代表应用程序编程接口或者抽象编程接口。本书称如何使用类的方法为类的接口。明确区分类的接口与类的实现是非常重要的。一个设计良好的类, 只需暴露给编程人员相关的接口, 而无须编程人员了解其实现的细节。这种将接口和实现分离的类常常被称为抽象数据类型或者良好封装的类。本书第 11 章将详细介绍通过将类的接口和实现分别放在不同的文件中来实现接口和实现分离的目的, 但更重要的是在概念上区分这两者。

#### 实现

对于 C++ 的类来讲, 接口包含两个部分: 注释和公有成员函数。注释通常放在类定义的开头, 用来说明类对象所代表的的数据, 如日期或者银行账户等。公有成员函数也包含注释, 用来解释这些成员函数的用法。一个设计良好的类, 接口应该包含所有使用该类的信息。

类的实现就是具体实现该类接口的 C++ 代码。类的实现包含两部分: 私有数据成员和所有成员函数的定义。为了运行程序, 类的实现是不可或缺的, 但在编写程序的其他部分时, 并不需要了解该类的具体实现。也就是说, 在编写 `main` 函数或者其他普通非成员函数时, 我们无须理会所用到的类的具体实现。

接口和实现相分离的一个巨大好处就是可以根据需要改变类的实现代码, 但却无须修改使用该类的相关代码, 因此类的接口并没有改变。在一个大的项目中, 这种分离可以将编程工作分配给不同的程序员。如果类的接口设计良好, 那么一个程序员可以编写该类的实现代码, 而另一个程序员可以同时编写其他使用该类的代码。即使整个项目由一个程序员来完成, 这种接口和实现分离的方式也可以将一项大的工作分成两个小的工作, 从而使得程序的设计和调试变得更加容易。■

### 提示：封装的测试

如果类的定义良好, 是一个抽象数据类型, 也就是类的接口和实现得到了良好的分离, 那么就可以根据需要改变类的具体实现 (也就是改变成员变量的表现类型或者改变成员函数的具体实现), 而无须改变程序中使用该类的代码。从这一角度出发, 可以很好地测试你所定义的类是一个 ADT 还是一个没有正确封装的类。

例如, 可以将示例 6.4 中类 `DayOfYear` 的实现改为如下代码时, 使用该类的其他代码无须做任何改变。

```
class DayOfYear
{
public:
    void input();
```

```

void output( );

void set(int newMonth, int newDay);
// 前提条件: newMonth 和 newDay 可以构成一个合法的日期。
// 运行结果: 根据参数的值重新对日期进行设置。

void set(int newMonth);
// 前提条件: 1<= newMonth <= 12
// 运行结果: 将日期设置为给定月份的第一天。
int getMonthNumber( );
// 返回代表月份的数字。
int getDay( );

private:
char firstLetter; // 月份单词的第一个字母。
char secondLetter; // 月份单词的第二个字母。
char thirdLetter; // 月份单词的第三个字母。
int day;
};

```

在这一版本的实现中,月份由其英文单词的前三个字母来表示,比如 'J', 'a' 和 'n' 代表一月份 (January)。当然,类的成员函数也需重写,但重写之后的成员函数,其行为和之前完全一样。例如,成员函数 getMonthNumber 的实现:

```

int DayOfYear::getMonthNumber( )
{
    if (firstLetter == 'J' && secondLetter == 'a'
        && thirdLetter == 'n')
        return 1;
    if (firstLetter == 'F' && secondLetter == 'e'
        && thirdLetter == 'b')
        return 2;
    ...
}

```

书写这类函数可能会比较琐碎、乏味,只需花点时间,但并不困难。■

## 结构体与类

结构体通常不包含成员函数,并且所有的成员变量也都为公有的。但 C++ 中,结构体可以包含私有成员变量,并且也可以包含私有以及公有成员函数。除了符号上的区别外,C++ 结构体和类的功能完全相同。但尽管如此,我们还是提倡不要因此将结构体和类一样地使用,否则我们就会发现同一个概念却有着两个不同的名字(以及语法规则)。另一方面,如果按照通常的习惯来使用结构体,就可以很好地区分结构体和类。这也与大多数程序员的使用习惯相同。

结构体和类的一个区别是默认权限的不同。如果定义中第一组成员未指明是公有的还是私有的,那么在结构体中这些成员将为公有的,而在类中这些成员将为私有的。

### 提示:对象思考

如果之前没有使用过类进行编程,那么可能得花一段时间才能体会这其中的特点。采用类进行编程时,关注的焦点在数据而不是以往的算法,但这并不是说算法不重要。

而是算法是为数据服务的，是为数据设计相应的算法而不是相反。在极端的情况下，程序中没有任何全局函数，只有包含成员函数的类。在这种情况下，程序员声明各种不同的对象以及对象间的交互，而不是设计算法来对数据进行操作。对这方面的介绍将贯穿本书。当然，你可以在编程中完全忽略类，或者把它放在一个次要的地位，但这种情况下实际上是 C 语言编程，而不是 C++ 编程。■

## 类和对象

类是一种包含了成员变量和成员函数的数据类型。类定义的语法如下：

### 语法

```
class Class_Name
{
    .
    .
    .
    public:
        Member_Specification_N+1
        Member_Specification_N+2
        .
        .
        .
    private:
        Member_Specification_1
        Member_Specification_2
        .
        .
        .
        Member_Specification_N
};
```

公有成员

私有成员

← 不要忘记这里的分号。

每个 `Member_Specification_i` 可以是一个成员变量，或是一个成员函数。

类的定义中可以使用多个 `public` 和 `private`。如果第一组成员未指明是公有的还是私有的，则默认为私有的。

### 示例

```
class Bicycle
{
public:
    char getColor();
    int numberOfSpeeds();
    void set(int theSpeeds, char theColor);
private:
    int speeds;
    char color;
};
```

类一旦定义，就可以用该类来声明对象（类类型的变量），其用法就和普通数据类型一样。例如，下面的语句声明了两个 `Bicycle` 类的对象：

```
Bicycle myBike, yourBike;
```

### 自测练习题

16. 定义 C++ 的类时，应该将成员变量设为私有的还是公有的？成员函数应该被设置为公有的还是私有的？
17. 定义 C++ 类时，类的接口包含哪些部分？类的实现又包含哪些部分？

### 本章小结

- 结构体可以将不同类型的数据组合成一个单一（复合）的数据。
- 类可以将数据和函数组成一个单一（复合）的对象。
- 类中的成员变量或者成员函数既可以是公有的，也可以是私有的。如果是公有的，可以在类外使用该成员；如果是私有的，则只能在成员函数的定义中使用该成员。
- 函数的形参可以是类类型，也可以是结构体类型。同样，函数的返回值可以是类类型或者结构体类型。
- 类的成员函数可以像普通函数那样被重载。
- 定义 C++ 的类时，应该将接口和实现分开，这样类的使用者只需了解类的接口，而无须关注类的实现。这就是封装原则。

### 自测练习题答案

1. a. double  
b. double  
c. 非法——不能用结构体名字代替结构体变量名  
d. char  
e. CDAccountV2
2. a. \$9.99  
b. \$1.11
3. Stuff 结构体定义的末尾缺少分号。
4. A x = {1, 2};
5. a. 初始值太少；非语法错误。初始化之后，month == 12, day == 21, year == 0。对于没有提供初始化值的成员变量，会自动初始化为相应的 0 值。  
b. 正确的初始化；初始化后，12 == month, 21 == day, 1995 == year。  
c. 错误：初始值的数目太多。



```

6. struct EmployeeRecord
{
    double wageRate;
    int vacation;
    char status;
};

7. void readShoeRecord(ShoeType& newShoe)
{
    cout << "Enter shoe style (one letter): ";
    cin >> newShoe.style;
    cout << "Enter shoe price $";
    cin >> newShoe.price;
}

8. ShoeType discount(ShoeType oldRecord)
{
    ShoeType temp;
    temp.style = oldRecord.style;
    temp.price = 0.90*oldRecord.price;
    return temp;
}

9. void DayOfYear::input( )
{
    cout << "Enter month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}

10. void Temperature::set(double newDegrees, char newScale)
{
    degrees = newDegrees;
    scale = newScale;
}

```

11. 点运算符和作用域运算符都是用来指明该成员属于哪一个类或者结构体的。对于示例 6.3 中的类 DayOfYear, 假设 today 是该类的一个对象, 那么可以采用 today.month 来访问该对象的成员变量 month。定义成员函数时, 作用域运算符告诉编译器该成员函数属于哪一个类。

```

12. hyundai.price = 4999.99; // 非法, price 是私有成员变量。
    jaguar.setPrice(30000.97); // 合法
    double aPrice, aProfit; // 合法
    aPrice = jaguar.getPrice(); // 合法
    aProfit = jaguar.getProfit(); // 非法, getProfit 是私有成员函数。
    aProfit = hyundai.getProfit(); // 非法, getProfit 是私有成员函数。
    hyundai = jaguar; // 合法

```

13. 修改之后, 所有语句都合法。

14. 所有标记为私有的成员, 都只能在该类成员函数的定义中被引用。而标记为 public 的成员则没有这样的限制。

15. a. 只要一个就可以。如果类的定义中完全没有公有成员，编译器会报警。  
b. 可以没有。但一般情况下应该至少给出一个 `private:` 小节。
16. 全部的成员变量都应为私有的。属于类接口的成员应该被设为公有的。可以包含一些仅在该类内部使用的辅助的成员函数，但这些成员函数应被设为私有的。
17. 类中所有私有成员变量的声明都是类实现的一部分。所有公有成员函数的声明以及相关的注释都是类接口的一部分。所有私有成员函数的声明都是类实现的一部分。所有成员的定义（不管是公有成员函数还是私有成员函数）都是类实现的一部分。

## 编程练习

1. 根据下面的规则，编写一个班级的分数统计程序：
  - a. 有两次测验，每次测验的总分都为 10 分。
  - b. 有一次期中考试和期末考试，总分都为 100 分。
  - c. 期末考试成绩占最终成绩的 50%，期中考试成绩占最终成绩的 25%，而两次测验共占最终成绩的 25%。

总分 90 及以上记为 A，80 及以上记为 B，70 及以上记为 C，60 及以上记为 D，低于 60 记为 F。该程序读取学生各次的分数，然后给出最终的成绩记录。最后的成绩记录包括两次测验的分数、两次考试的分数、最终的平均分以及最后的等级。用结构体来完成学生分数的记录。

2. 定义一个名为 `CounterType` 的类，该类用来计数，可以记录所有的非负整数。该类包含一个赋值函数来为计数赋值，还包括对计数进行增 1、减 1 的成员函数。所有的成员函数都必须有安全性检查，不能将计数改变为负值。此外，该类还包含一个返回当前计数的成员函数以及对当前计数进行输出的成员函数。类定义完成后，编写程序对其进行测试。
3. `Point` 是一个相当简单的数据类型，定义在标准模板库中的 `pair` 命名空间下，该练习并不需要了解任何关于标准模板库的知识。编写一个 `Point` 类，用来记录和操作平面上的点。需要声明和定义的成员函数如下：
  - a. 一个名为 `set` 的成员函数，在该类的对象创建完成之后，该函数可以对对象的私有数据成员赋值。
  - b. 一个移动点的成员函数。该函数含有两个参数：第一个参数代表水平方向的位移，第二个参数代表垂直方向的位移。
  - c. 用于将点绕着原点旋转  $90^\circ$  的成员函数。
  - d. 两个常数成员函数，用于获取当前的坐标值。

为这些成员函数添加适当的注释，编写相关的程序对该类进行测试。让用户输入几个点，产生相关的对象，并练习各个成员函数的用法。

4. 编写一个名为 `GasPump` 的类，该类将用于模拟汽车加油站的油泵。在具体编写代码之前，试着从加油站的角度考虑所期望的油泵的行为。

下面给出一些可能的行为，可以根据情况再添加相关的方法，最后在类的定义中实现这些行为：

- 显示总的加油量。
- 显示总的加油量对应的费用。
- 显示单位油量的费用，单位可以是加仑、升、或者其他任何会用到的容量单位。
- 在下次加油之前，油泵应该将总的加油量和总的费用清零。
- 加油一旦开始，随着油量的输出，总的加油量和总费用应同时变化，直到加油结束。
- 在某些情况下，还要让油泵停止加油。

将上面的行为声明为类 `GasPump` 的成员函数，并给出这些成员函数的实现代码。另外，讨论 `GasPump` 需要保存的而用户又无权访问的数据，将这些数据设置为私有成员变量。

5. 定义一个名为 `Fraction` 的类，用以表示两个整数的比值。该类包含相关的赋值函数用于设置分子和分母，并且包含一个成员函数返回该分数的值（`double` 类型），此外还包含一个额外的成员函数用于输出化简后的分数。例如，该函数会把 `20/60` 输出为 `1/3`。这就要求首先找出分子和分母的最大公因子，并分别除分子和分母。编写一个完整的程序对你编写的函数进行测试。
6. 定义一个名为 `Odometer` 的类，用以记录汽车的油耗以及行驶里程（英里）。该类应该包含对应的成员变量来记录汽车行驶的英里数以及燃油效率（英里/加仑）。类要包含一个赋值函数对里程表清零，一个赋值函数用来设置燃油效率，以及一个赋值函数用来接受一次旅程的总里程并将其加到里程表上，还需包含一个取值函数用以返回自上次清零之后汽车消耗的汽油（加仑）。

编写一个完整的程序，通过输入不同的燃油效率对该类进行测试。另外，如果该类包含成员变量，仔细考虑它的访问权限（`public` 或者 `private`）。

7. 编写一个名为 `Pizza` 的类，用以记录披萨的类型（`deep dish`、`hand tossed` 或者 `pan`）、尺寸（小、中或者大）以及配料的种类。可以用常量记录披萨的类型和尺寸，并且包含相关的取值、赋值函数。编写一个名为 `outputDescription` 的 `void` 函数，用于输出描述该披萨的文本信息。此外，编写一个名为 `computePrice` 的函数，根据如下的规则，计算该披萨的价格：

小号披萨 = \$10 + \$2/ 配料

中号披萨 = \$14 + \$2/ 配料

大号披萨 = \$17 + \$2/ 配料

编写相关的程序对 `Pizza` 类进行完整的测试。

8. 编写一个名为 `Money` 的类来记录一定的美元数。该类包含两个私有成员分别记录美元数和美分数。为该类添加相关的取值、赋值函数对这两个私有成员进行操作，此外为该类添加一个函数以 `double` 类型返回最终的美元数。编写相关的程序对 `Money` 类进行完整的测试。
9. 重做上一题，这次采用一个 `double` 类型的变量来记录美元数，而不是两个 `int` 类型的私有成员变量。该类的其他成员函数头保持不变，不过这其中可能会涉及将某些变量从 `int` 类型转变为 `double` 类型。例如，假设存储在 `double` 类型变量中的美元数是 4.55，用户采用实参 13 调用了设置美元部分的赋值成员函数，那么记录美元数的 `double` 变量此时应该变为 13.55。采用上一题的测试程序对修改后的 `Money` 类进行测试，如果保持类接口不变的话，那么上一题的测试程序将无须做任何修改便可正常运行，这也很明显地体现了成员变量封装的好处。
10. 编写一个名为 `Temperature` 的类，记录开氏温度。创建名为 `setTempKelvin`、`setTempFahrenheit` 及 `setTempCelsius` 的成员函数，这些成员函数接受对应的温度值，并将其转化为开氏温度，记录到类的成员变量中。此外，创建对应的取值函数，将类成员变量中记录的温度值分别以开氏温度、华氏温度、摄氏温度返回。编写完整的测试程序对该类进行测试。程序中采用如下的温度转化公式：

$$\text{Kelvin} = \text{Celsius} + 273.15$$

$$\text{Celsius} = (5.0/9) \times (\text{Fahrenheit} - 32)$$

11. 重做第 5 章的编程练习 18，这次使用一个数组参数，而不是两个数组参数。该数组参数的类型为类类型 `Player`。`Player` 类包含一个类型为 `string` 的成员变量来记录选手的名字、一个类型为 `int` 的成员变量记录选手的成绩。对该函数做良好的封装。函数返回时，数组的第一个元素中包含冠军选手的名字和得分情况，数组的第二个元素则包含亚军选手的名字和得分情况，依此类推。
12. 一个名为 `CSA` 的农场每周会向你家邮寄一箱新鲜的蔬菜和水果。编写一个名为 `BoxOfProduce` 的类，该类包含三捆蔬菜或者水果，这里可以采用一个 `string` 类型的数组来表示。为该类编写对应的取值、赋值函数，用以设置和获取数组中存放的蔬菜或水果类型。此外，编写一个 `output` 函数，用以在终端上输出该箱水果或蔬菜的具体内容。

类的定义完成之后，编写一个 `main` 函数，从以下列表中随机选择三个来初始化一个 `BoxOfProduce` 对象：

```
Broccoli
Tomato
Kiwi
Kale
Tomatillo
```

你编写的程序应从一个文本文件中读取以上蔬果列表，这里可以假设该文件只包含五种蔬果列表。

程序随机选择的三种蔬果可以重复，紧接着，`main` 函数应展示该蔬果箱的内容，并允许用户对蔬果箱中的蔬果进行替换。待用户替换完毕之后，输出最终的蔬果箱内容。



## 构造函数及其他工具



### 7.1 构造函数 226

构造函数的定义 226

陷阱：无参构造函数 230

构造函数的显式调用 230

提示：总是为类定义一个默认构造函数 231

示例：BankAccount类 233

类类型成员变量 238

### 7.2 其他工具 241

const修饰符 241

陷阱：const的不一致用法 243

内联函数 246

静态成员 248

嵌套类和局部类定义 251

### 7.3 向量——标准模板库预览 251

向量基础 252

陷阱：方括号的索引超过向量的大小 253

提示：向量的赋值 254

效率问题 255

# 第7章 构造函数及其他工具

只要给我们工具，我们就能完成任务。

英国首相丘吉尔，广播（1941年2月9日）

## 概述

本章将对使用类编程时会用到的一些重要工具进行介绍。其中，最重要的就是用来初始化类对象的构造函数。

7.3 节以向量（vector）类为例介绍类及对象的使用，并以此引入 C++ 标准模板库（STL）的介绍。向量和数组类似，但其大小可以变化。而标准模板库则是由一些预先定义的类所组成的扩展库。7.3 节的内容独立于第 8 章到第 18 章的内容，因此可以根据需要，合理安排学习顺序。

学习 7.1 节和 7.2 节无须第 5 章的知识，而学习 7.3 节则需要前面第 1 ~ 6 章和 7.1 节的知识。

## 7.1 构造函数

好的开始是成功的一半。

一句俗语

声明一个对象时，常常需要初始化对象的成员变量以及其他内容，其中成员变量的初始化是最常用到的。C++ 为对象的初始化提供了专门的方法。定义一个类时，可以定义一种特殊的成员函数，称为**构造函数**。声明一个类的对象时，构造函数自动被调用，从而对该对象部分或全部的成员变量进行初始化，同时根据需要，也可以完成一些其他的初始化工作。

构造函数

### 构造函数的定义

构造函数的定义与一般成员函数的定义类似，但需注意以下两点：

1. 构造函数必须和类同名。例如，如果类的名字为 BankAccount，那么该类的所有构造函数名必须为 BankAccount。
2. 构造函数无返回值。此外，任何数据类型（包括 void 类型）都不可以放在构造函数声明的开头或函数的头文件中。

例如，假设要为示例 6.4 中的类 DayOfYear 添加构造函数，以便初始化其中的月份和日期，那么改写如下（这里为了节省纸张，省略了部分程序注释，在实际的程序中应加上这些注释）：

```
class DayOfYear
{
```

```

public:
    DayOfYear(int monthValue, int dayValue); ←———— 构造函数
    // 初始化月份和日期为相应的参数值。

    void input( );
    void output( );
    void set(int newMonth, int newDay);
    void set(int newMonth);
    int getMonthNumber();
    int getDay( );
private:
    int month;
    int day;
};

```

注意，上面的代码中，构造函数名为 DayOfYear，与类同名。同时在构造函数的声明中，没有任何数据类型的修饰。最后需要注意的是，构造函数是放在类的公有部分声明的。一般来讲，构造函数应该是公有成员函数。因为如果将所有的构造函数都声明为私有的话，那么就不能声明任何该类的任何对象，从而使得该类毫无用处。

基于以上添加了构造函数的类 DayOfYear，可以按照如下方式声明并初始化该类的两个对象：

```
DayOfYear date1(7,4), date2(5,5);
```

假设构造函数按照我们预想的正常执行，那么上面的声明将产生两个对象，对象 date1 的 month 和 day 分别被初始化为 7 和 4，对象 date2 的则分别被初始化为 5 和 5。因此 date1 代表日期 7 月 4 号，date2 则代表日期 5 月 5 号。实际上，上面的语句首先声明对象 date1，继而对应的构造函数被调用，并传递参数 7 和 4。同样，date2 被声明，随之构造函数被调用，并传递参数 5 和 5。从概念上讲，上面的语句与如下的语句等价（虽然这些语句在 C++ 中并不合法）：

```

DayOfYear date1, date2; // 有问题，但可修复
date1.DayOfYear(7,4); // 完全非法
date2.DayOfYear(5,5); // 完全非法

```

正如注释表明的那样，以上语句在 C++ 中是非法的。第一条语句在某些情况下是可以接受的，但后面两条直接调用构造函数的用法是完全非法的。构造函数不能像普通成员函数那样被直接调用。虽然以上三行代码的用意显而易见，但应该按照如下方式来声明并初始化对象：

```
DayOfYear date1(7,4), date2(5,5);
```

构造函数的定义和一般成员函数的定义类似。例如，为类 DayOfYear 添加了以上的构造函数，那么也需添加对应的构造函数定义：

```

DayOfYear::DayOfYear(int monthvalue, int dayvalue)
{
    month = monthValue;
    day = dayvalue;
}

```

由于构造函数与类同名，DayOfYear 在函数头中出现了两次，出现在作用域运算



符::之前的是类名,之后的是构造函数名。同样还需注意的是,构造函数没有返回值类型,哪怕是void类型也不行。除了这些区别外,构造函数的定义和一般成员函数完全相同。

### 构造函数

构造函数是一种特殊的成员函数,它与类同名。声明该类的对象时,构造函数将被自动调用,并对对象进行初始化。注意,构造函数的名字须与对应的类同名。

正如前面介绍的一样,构造函数可以像其他成员函数那样进行定义。然而,构造函数还可以采用另外一种形式。前面构造函数DayOfYear的定义完全等价于下面的定义:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
    : month(monthValue), day(dayValue)
{
    /* 函数体故意留空 */
}
```

#### 初始化列表

以上构造函数定义的第二行被称为初始化列表。如上所示,初始化列表放在函数参数列表结束圆括号与函数体开始大括号之间,以冒号开头,不同初始化变量之间以逗号隔开。每个成员变量的初始化值放在其后的圆括号中。请注意,初始化参数可以放在构造参数列表中。

包含初始化列表的构造函数,函数体不一定要为空。例如,下面的构造函数定义可以检查传入参数的正确性:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
    : month(monthValue), day(dayValue)
{
    if ((month < 1) || (month > 12))
    {
        cout<< "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout<< "Illegal day value!\n";
        exit(1);
    }
}
```

如同一般成员函数一样,也可以对构造函数进行重载。实际上,构造函数经常被重载,从而对象可以有多种初始化方式。例如,7.1节中重新定义了类DayOfYear,从而包含了三个版本的构造函数。重载的三个构造函数DayOfYear分别包含两个参数、一个参数或无参数。

注意,示例7.1中,两个构造函数都调用了—个名为testDate的成员函数来检查初始化值的正确性。成员函数testDate被声明为私有的,因为该函数仅仅是被该类的成员函数使用,因此对外界来说是不可见的。

这里,我们省略了成员函数set。一旦有了用于初始化的构造函数,就不再需要其他成员函数来设定成员变量的值。使用示例7.1中的构造函数可以达到与示例6.4中set函数相同的目的。

### 示例 7.1 带有构造函数的类

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      DayOfYear(int monthValue, int dayValue);
8      // 将月份和日期初始化为实参值。

9      DayOfYear(int monthValue);
10     // 将日期初始化为给定月份的第一天。

11     DayOfYear( );
12     // 将日期初始化为1月1号。

13     void input( );
14     void output( );
15     int getMonthNumber( );
16     // 返回以数字表示的月份，一月份返回数字1，二月份返回数字2。

17     int getDay( );
18 private:
19     int month;
20     int day;
21     void testDate();
22 };

23 int main()
24 {
25     DayOfYear date1(2, 21), date2(5), date3;
26     cout << "Initialized dates:\n";
27     date1.output( ); cout << endl;
28     date2.output( ); cout << endl;
29     date3.output( ); cout << endl;

30     date1 = DayOfYear(10, 31);
31     cout << "date1 reset to the following:\n";
32     date1.output( ); cout << endl;
33     return 0;
34 }

35
36 DayOfYear::DayOfYear(int monthValue, int dayValue)
37     : month(monthValue), day(dayValue)
38 {
39     testDate( );
40 }

41 DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
42 {
43     testDate( );
44 }

```

这里类 DayOfYear 的定义是示例 6.4 中的升级版。

默认构造函数

调用默认构造函数，注意这里没有使用圆括号。

显式调用构造函数 DayOfYear::DayOfYear。

```

45 DayOfYear::DayOfYear( ) : month(1), day(1)
46 { /* 函数体故意留空 */ }

47 // 使用了 iostream 和 cstdlib 库。
48 void DayOfYear::testDate( )
49 {
50     if ( (month < 1) || (month > 12) )
51     {
52         cout << "Illegal month value!\n";
53         exit(1);
54     }
55     if ( (day < 1) || (day > 31) )           < 其他成员函数的定义与示例 6.4 相同。 >
56     {
57         cout << "Illegal day value!\n";
58         exit(1);
59     }
60 }

```

### 示例运行结果

```

Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31

```

## 陷阱：无参构造函数

应该牢记，如果想使用无参构造函数来初始化对象，声明对象时后面不能使用圆括号。例如，示例 7.1 中的如下语句：

```
DayOfYear date1(2, 21), date2(5), date3;
```

对象 date1 将用两个参数的构造函数初始化，对象 date2 将用一个参数的构造函数来初始化；而对象 date3 则使用无参构造函数初始化。

使用无参构造函数初始化对象时，你或许会认为应该在声明的对象后面使用一对空的括号，但实际上不能这么做。例如，考虑下面声明对象 date3 的方式：

```
DayOfYear date3(); // 存在问题
```

这样做的问题在于，尽管你认为是在声明一个对象，并使用无参构造函数来对其进行初始化；但实际上，编译器却认为这是一个函数原型的声明：一个名为 date3 的无参函数，其返回值类型为 DayOfYear 类型。正是由于这个原因，使用无参构造函数来初始化的对象必须采用不同的形式。■

## 构造函数的显式调用

无论何处声明类的对象，构造函数都将被自动调用，但构造函数同样也可以在对象已经声明之后被再次调用。这可以让大家很方便地为对象的所有成员变量赋值。具体如下：显式调用构造函数来构造一个匿名对象，并存储新值。匿名对象是指没有用

任何变量来命名的对象，匿名对象可以赋值给一个有名对象。例如，下面的语句调用 DayOfYear 构造函数创建一个匿名对象，保存日期<sup>1</sup>5月5号。然后该匿名对象赋给变量 holiday（已声明为 DayOfYear 类型），这样变量 holiday 也存储日期5月5号<sup>1</sup>。

```
holiday = DayOfYear(5, 5);
```

（从代码的形式出发，构造函数可以看作是一个返回对应类对象的一般函数。）

需要注意的是，如果显式地调用一个无参构造函数，则必须在函数名后面加上一对圆括号，如下所示：

```
holiday = DayOfYear();
```

只有在声明类的对象，并采用无参构造函数来初始化该对象的情况下，才可以省略无参构造函数的圆括号。

### 构造函数的调用

构造函数在声明对象时被自动调用，但必须给其相应的参数。构造函数也可以被显式调用，但调用方式与一般成员函数不同。

包含构造函数的类对象声明语法：

```
Class_Name Variable_Name (Arguments_for_Constructor);
```

**示例**

```
DayOfYear holiday(7,4);
```

**构造函数显式调用的语法**

```
Variable = Constructor_Name (Arguments_For_Constructor);
```

**示例**

```
holiday = DayOfYear(10,31);
```

由于构造函数必须与类同名，因此上面语法描述中的类名和构造函数名是同一个标识符。

**提示：**总是为类定义一个默认构造函数

### 默认构造函数

无参数的构造函数被称为**默认构造函数**。这个名字容易让人产生误解，因为有时候无参构造函数是自动生成的，而有时候却不是。如果类没有定义构造函数，那么编译器会自动产生一个默认构造函数。这个自动生成的构造函数不会自动做任何事情，仅仅是产生一个没有初始化的该类的对象。如果类中已经定义了构造函数，那么编译器就不会自动产生任何构造函数。例如，假设定义了一个名为 SampleClass 的类。

如果这个类已经定义了一个或多个带有一个或多个参数的构造函数，但没有包含默认构造函数，那么该类将没有默认构造函数，因此下面的语句将是不合法的：

<sup>1</sup> 注意，这一过程不是简简单单地改变成员变量的值，考虑到代码的运行效率，类的定义中还是应该包含一个 set 赋值函数来替代这种显式调用构造函数的用法。

```
SampleClass aVariable;
```

上面语句的问题在于：该语句会让编译器去自动调用默认构造函数，但类的定义里却没有默认构造函数。

为了让这个语句合法，假设 SampleClass 的定义如下：

```
class SampleClass
{
public:
    SampleClass(int parameter1, double parameter2);
    void doStuff( );
private:
    int data1;
    double data2;
};
```

那么下面的声明方式是合法的，在声明对象的同时并调用类的构造函数来初始化对象。

```
SampleClass myVariable(7, 7.77);
```

然而下面的语句是不合法的：

```
SampleClass yourVariable;
```

编译器将上面的语句认为是对默认构造函数的调用，但类的定义里却没有默认构造函数的声明。为了使该语句合法，有两种方法：在对象 yourVariable 的声明后面加上两个参数，或者为类添加一个无参构造函数的声明和定义。

如果按照下面的方式来对 SampleClass 重新定义，那么以上对 yourVariable 对象的声明将没有任何问题。

```
class SampleClass
{
public:
    SampleClass(int parameter1, double parameter2);
    SampleClass( );
    void doStuff( );
private:
    int data1;
    double data2;
};
```

← 默认构造函数

为了避免这种混淆，最好在所有类的定义中都添加默认构造函数的声明和定义。如果不希望用默认构造函数来初始化任何成员变量，那么只需要将其函数体定义为空即可。下面构造函数的定义完全正确，它没有做任何成员变量的初始化工作。

```
SampleClass::SampleClass( )
{ /* 函数体为空 */ } ■
```

### 无参构造函数

没有任何参数的构造函数被称为默认构造函数。声明类对象时，如果想用默认构造函数来初始化对象，则需省略对象名后面的圆括号。例如，使用两个参数的构造函数来声明类对象的语句如下：

```
DayOfYear date1(12,31);
```

使用无参构造函数来声明类对象的语句如下：

```
DayOfYear date2;
```

而不能使用如下的语句来使用无参构造函数。

```
DayOfYear date2();// 错误
```

问题在于：根据 C++ 的语句，编译器会将上面的语句解释为一个名为 date2、返回值类型为 DayOfYear 的无参函数的声明。但显式调用无参构造函数时，则必须使用圆括号。

```
date1 = DayOfYear()
```

### 自测题

1. 假设程序中包含如下的类定义：

```
class YourClass
{
public:
    YourClass(int newInfo, char moreNewInfo);
    YourClass( );
    void doStuff( );
private:
    int information;
    char moreInformation;
};
```

以下语句中，哪些是合法的？

```
YourClass anObject(42, 'A');
YourClass anotherObject;
YourClass yetAnotherObject();
anObject = YourClass(99, 'B');
anObject = YourClass( );
anObject = YourClass;
```

2. 什么是默认构造函数？是否所有的类都有默认构造函数？

### 示例：BankAccount 类

示例 7.2 给出了一个简单的银行账户类的定义，并使用在一个简单的程序中。这里，银行账户包含两部分数据：账户余额和利率。其中余额用两个整型值来表示：一个代表美元，一个代表美分。这说明了一个事实：类的定义中，一个概念上的数据在类的

内部可以用一个或多个成员变量来表示。账户余额看似应该用一个 double 类型来表示，而不是两个整型值来表示。但事实上，一个账户余额可以用美元和美分来准确表达，而 double 类型却是一个近似值。而且，账户余额 (\$323.52) 并不是简单地在一个浮点值前面加上一个美元符号。账户余额 \$323.52 在小数点后不能超过或者少于两位。账户余额不能为 \$323.523，但 double 类型却允许这样的数值出现。从技术角度来讲，美分的数值不是不可以为小数，只是对于银行账户来讲，美分的数量必须是整数。

类 BankAccount 的使用者可以认为账户的余额是用 double 类型表示的，也可以认为是用两个 int 类型表示的。类的取值和赋值函数允许编程人员通过一个 double 类型或者两个 int 类型的数据来读取和设置账户余额，但无须知道表示账户余额的具体细节，这一部分属于被“隐藏”的信息。

注意，示例中的赋值函数 setBalance 和构造函数都被重载了。同时所有的构造函数和赋值函数都会检查数值的正确性。例如，利率不能为负，账户余额可以为负，但不能出现“元”部分为正，而“分”部分为负的情况。

类 BankAccount 还含有四个私有成员函数：dollarsPart、centsPart、round 和 fraction。由于这几个函数只是在类内部使用，因此被定义为私有的。

## 示例 7.2 BankAccount 类

```

1  #include <iostream>
2  #include <cmath>
3  #include <cstdlib>
4  using namespace std;

5  // 银行账户由两部分组成：
6  // 账户余额和利率。
7  class BankAccount
8  {
9  public:
10     BankAccount(double balance, double rate);
11     // 根据实参值初始化账户余额和利率。

12     BankAccount(int dollars, int cents, double rate);
13     // 将账户余额初始化为 $dollars.cents。
14     // 对于负的账户余额，dollars 和 cents 都必须为负。
15     // 初始化账户利率为 rate。
16     BankAccount(int dollars, double rate);
17     // 账户余额初始化为 $dollars.00，账户利率初始化为 rate。

18     BankAccount();
19     // 初始化账户余额为 $0.00，账户利率为 0.0%。
20     void update();
21     // 运行结果：将一年的利息记入账户余额。
22     void input();
23     void output();
24     double getBalance();
25     int getDollars();
26     int getCents();

```

```

27     double getRate( );// 返回账户利率。

28     void setBalance(double balance);
29     void setBalance(int dollars, int cents);
30     // 检查参数是否同时为负或者同时为正。

31     void setRate(double newRate);
32     // 如果 newRate 非负, 将其设置为新的账户利率, 否则终止程序。
33     private:
34         // 负的账户余额用负的 dollars 和负的 cents 表示。
35         // 例如, -$4.50 将 accountDollars 设置为 -4, 将 accountCents 设置为 -50。
36         int accountDollars; // 账户的美元部分
37         int accountCents; // 美分部分
38         double rate; // 两者的百分比
39         int dollarsPart(double amount);
40         int centsPart(double amount);
41         int round(double number);
42
43         double fraction(double percent);
44         // 将百分数转化为小数, 例如 fraction(50.3) 返回 0.503。
45     };

46 int main()
47 {
48     BankAccount account1(1345.52, 2.3), account2;
49     cout << "account1 initialized as follows:\n";
50     account1.output( );
51     cout << "account2 initialized as follows:\n";
52     account2.output( );
53
54     account1 = BankAccount(999, 99, 5.5);
55     cout << "account1 reset to the following:\n";
56     account1.output( );
57
58     cout << "Enter new data for account 2:\n";
59     account2.input( );
60     cout << "account2 reset to the following:\n";
61     account2.output( );
62     account2.update( );
63     cout << "In one year account2 will grow to:\n";
64     account2.output( );
65
66     return 0;
67 }

68 BankAccount::BankAccount(double balance, double rate)
69 : accountDollars(dollarsPart(balance)),
70   accountCents(centsPart(balance))
71 {
72     setRate(rate);
73 }

74 BankAccount::BankAccount(int dollars, int cents, double rate)
75 {

```

私有成员

此声明将会调用默认构造函数, 注意这里没有圆括号。

显式调用构造函数 BankAccount:BankAccount。



```

72     setBalance(dollars, cents); ← 这些函数检查数据是否正确。
73     setRate(rate);
74 }

75 BankAccount::BankAccount(int dollars, double rate)
76     : accountDollars(dollars), accountCents(0)
77 {
78     setRate(rate);
79 }

80 BankAccount::BankAccount( ): accountDollars(0),
81     accountCents(0), rate(0.0)
82 { /* 函数体故意留空 */ }

82 void BankAccount::update( )
83 {
84     double balance = accountDollars + accountCents*0.01;
85     balance = balance + fraction(rate)*balance;
86     accountDollars = dollarsPart(balance);
87     accountCents = centsPart(balance);
88 }

89 // 采用输入/输出流。
90 void BankAccount::input( )           函数 BankAccount::input 的改进版,
91 {                                     参见自测练习题3。
92     double balanceAsDouble;
93     cout << "Enter account balance $";
94     cin >> balanceAsDouble;
95     accountDollars = dollarsPart(balanceAsDouble);
96     accountCents = centsPart(balanceAsDouble);
97     cout << "Enter interest rate (NO percent sign):";
98     cin >> rate;
99     setRate(rate);
100 }

101 // 采用 iostream 和 cstdlib。
102 void BankAccount::output( )
103 {
104     int absDollars = abs(accountDollars);
105     int absCents = abs(accountCents);
106     cout << "Account balance: $";
107     if (accountDollars > 0)
108         cout << "-";
109     cout << absDollars;
110     if (absCents >= 10)
111         cout << "." << absCents << endl;
112     else
113         cout << "." << '0' << absCents << endl;
114     cout << "Rate:" << rate << "%\n";
115 }

116 double BankAccount::getBalance( )
117 {
118     return (accountDollars + accountCents * 0.01);
119 }

```

```

120 int BankAccount::getDollars( )
121 {
122     return accountDollars;
123 }
124
125 int BankAccount::getCents( )
126 {
127     return accountCents;
128 }
129
130 double BankAccount::getRate( )
131 {
132     return rate;
133 }
134
135 void BankAccount::setBalance(double balance)
136 {
137     accountDollars = dollarsPart(balance);
138     accountCents = centsPart(balance);
139 }
140
141 //使用 cstdlib.
142 void BankAccount::setBalance(int dollars, int cents)
143 {
144     if ((dollars < 0 && cents > 0) || (dollars > 0 && cents < 0))
145     {
146         cout << "Inconsistent account data.\n";
147         exit(1);
148     }
149     accountDollars = dollars;
150     accountCents = cents;
151 }
152
153 //使用 cstdlib.
154 void BankAccount::setRate(double newRate)
155 {
156     if (newRate >= 0.0)
157         rate = newRate;
158     else
159     {
160         cout << "Cannot have a negative interest rate.\n";
161         exit(1);
162     }
163 }
164 int BankAccount::dollarsPart(double amount)
165 {
166     return static_cast<int>(amount);
167 }
168 //使用 cmath.
169 int BankAccount::centsPart(double amount)
170 {
171     double doubleCents = amount * 100;
172     int intCents = (round(fabs(doubleCents))) % 100;
173     //在操作数为负值时会出现意想不到的情况。
174     if (amount < 0)
175         intCents = -intCents;

```

该类的使用者无须关心 balance 究竟存储在一个还是两个成员变量中。

此函数更应该作为一个普通函数来定义，但是作为成员函数定义时，可以将其设置为私有的。

这些函数可以是常规的函数，不过以成员函数的方式，我们可以将其设置为私有的。

```

175     return intCents;
176 }
177
178 // 使用 cmath。
179 int BankAccount::round(double number)
180 {
181     return static_cast<int>(floor(number+0.5));
182 }
183
184 double BankAccount::fraction(double percent)
185 {
186     return (percent/100.0);
187 }

```

如果该函数不容易理解，可参见第3章3.2节关于 round 函数的讨论。

### 示例运行结果

```

account1 initialized as follows:
Account balance: $1345.52
Rate: 2.3%
account2 initialized as follows:
Account balance: $0.00
Rate: 0%
account1 reset to the following:
Account balance: $999.99
Rate: 5.5%
Enter new data for account 2:
Enter account balance $100.00
Enter interest rate (NO percent sign): 10
account2 reset to the following:
Account balance: $100
Rate: 10%
In one year account2 will grow to:
Account balance: $110
Rate: 10%

```

### 自测练习

3. 示例 7.2 中的函数 `BankAccount::input` 将一个 `double` 类型的数值读入作为账户余额。当数据以二进制的方式存储在计算机内存中时，这可能会产生微小的误差。作为本书中一个简单的演示程序，这样处理没有问题，但对银行业中的实际程序，该函数是有问题的。这里请读者重写 `BankAccount::input` 函数，令其以字符形式读取小数部分，比如 78.96 读入整数 78 和三个字符 “.”、“9” 和 “6”。这里假设小数部分始终是两位，比如输入 99.00 而不是 99。提示：下面的语句可以将一个字符转换为对应的整数值，比如将 ‘6’ 转换为 6。

```
static_cast<int>(digit) - static_cast<int>('0')
```

### 类类型成员变量

一个类的对象可以作为成员变量出现在另一个类的定义中。这种情况一般不需要

做什么特殊处理，但如果要在外层类的构造函数中初始化该成员变量，则需用到特殊的符号，如示例 7.3 所示。

### 示例 7.3 类类型成员变量

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      DayOfYear(int monthValue, int dayValue);
8      DayOfYear(int monthValue);
9      DayOfYear();
10     void input();
11     void output();
12     int getMonthNumber();
13     int getDay();
14 private:
15     int month;
16     int day;
17     void testDate();
18 };

19 class Holiday
20 {
21 public:
22     Holiday(); // 初始化为 1 月 1 号
23     Holiday(int month, int day, bool theEnforcement);
24     void output();
25 private:
26     DayOfYear date; ←———— 类类型的成员变量。
27     bool parkingEnforcement; // 如果 enforced, 则为 true。
28 };

29 int main()
30 {
31     Holiday h(2, 14, true);
32     cout << "Testing the class Holiday.\n";
33     h.output();
34     return 0;
35 }

36
37 Holiday::Holiday(): date(1, 1), parkingEnforcement(false)
38 /* 函数体特意留空 */

39 Holiday::Holiday(int month, int day, bool theEnforcement):
40     :date(month, day), parkingEnforcement(theEnforcement)
41 /* 函数体特意留空 */
42 void Holiday::output()
43 {
44     date.output();
45     cout << endl;

```

这里 DayOfYear 与示例 7.1 中的定义完全相同。

类构造函数的调用。

```

46     if (parkingEnforcement)
47         cout << "Parking laws will be enforced.\n";
48     else
49         cout << "Parking laws will not be enforced.\n";
50 }

51 DayOfYear::DayOfYear(int monthValue, int dayValue)
52     : month(monthValue), day(dayValue)
53 {
54     testDate();
55 }

56 // 使用 iostream 和 cstdlib。
57 void DayOfYear::testDate()
58 {
59     if ((month < 1) || (month > 12))
60     {
61         cout << "Illegal month value!\n";
62         exit(1);
63     }
64     if ((day < 1) || (day > 31))
65     {
66         cout << "Illegal day value!\n";
67         exit(1);
68     }
69 }
70
71 // 使用 iostream。
72 void DayOfYear::output()
73 {
74     switch(month)
75     {
76         case 1:
77             cout << "January "; break;
78         case 2:
79             cout << "February "; break;
80         case 3:
81             cout << "March "; break;
82         .
83         .
84         .
85         case 11:
86             cout << "November "; break;
87         case 12:
88             cout << "December "; break;
89         default:
90             cout << "Error in DayOfYear::output";
91     }
92
93     cout << day;
94 }

```

参考示例 6.3 查看省略部分的代码。

### 示例运行结果

Testing the class Holiday:

```
February 14
Parking laws will be enforced.
```

---

示例 7.3 中的类 `Holiday` 可以帮助一些城市的警察局了解哪些节假日可以强制泊车（诸如停车计时以及 1 小时停车区等）。这是一个高度简化了的类，实际用到的类会包含更多的成员函数，但这里作为演示目的已经足够。

类 `Holiday` 包含两个成员变量。`parkingEnforcement` 是一个 `bool` 类型的普通成员变量，`date` 则是类 `DayOfYear` 的对象。

下面是示例 7.3 中的一个构造函数：

```
Holiday::Holiday(int month, int day, bool theEnforcement)
    : date(month, day), parkingEnforcement(theEnforcement)
{ /* 函数体故意留空 */ }
```

注意，初始化列表中成员变量 `parkingEnforcement` 的初始化方式和以前一样，即

```
parkingEnforcement(theEnforcement)
```

而成员变量 `date` 作为类 `DayOfYear` 的对象。为了初始化 `date`，需要调用类 `DayOfYear` 的构造函数，在初始化列表中，以如下方式进行：

```
date(month, day)
```

其中 `date(month, day)` 表示调用 `DayOfYear` 类的构造函数，并用参数 `month` 和 `day` 来初始化对象 `date` 的成员变量。这与声明一个 `DayOfYear` 类型的对象 `date` 是完全一致的。另外也需注意，外层类 `Holiday` 的构造函数中包含了成员变量构造函数的调用。

## 7.2 其他工具

智慧……是制造人造物品的设备，尤其是用工具制造工具。

亨利·柏格森，《创造的进化》

本节讨论三个重要但不简单的主体。分别是：类的 `const` 参数、内联函数以及类的静态成员。

### `const` 修饰符

毫无疑问的是，引用调用方式比传值调用方式效率高。传值调用时形参是一个局部变量，只是使用实参的值对其进行了初始化，因此函数调用时，相当于有两个实参的副本存在。但如果是引用调用方式，形参实际上是实参的一个别名，因此程序中只有实参的一个副本存在。对于简单的数据类型，如 `int` 或者 `double`，这两者在性能上的差别基本可以忽略。但对于类类型的形参来说，二者的效率有时会有很大差别。因此对于类类型的参数来讲，使用引用调用方式比传值调用效率会好很多，即使该函

数不会改变参数的值。

采用引用调用方式时，如果函数不改变参数的值，那么可以对该参数进行标记。为此，可以在该参数前面加上 `const` 修饰符。这样的参数称为**常参数**或者**常量引用参数**。例如，示例 7.2 中定义了名为 `BankAccount` 的类用来表示银行账户。在某些程序中，可能需要编写一个对账户余额进行比较的成员函数。该成员函数的定义可能如下所示：

```
bool isLarger(BankAccount account1, BankAccount account2)
// 如果 accout1 的余额大于 accout2，则返回 true，否则返回 false。
{
    return(account1.getBalance( ) > account2.getBalance( ));
}
```

上面的定义没有任何问题，函数的两个参数为传值调用。然而更常见、更高效的做法是使用常量引用调用方式，如下所示：

```
bool isLarger(const BankAccount& account1,
              const BankAccount& account2)
// 如果 accout1 的余额大于 accout2，则返回 true，否则返回 false。
{
    return(account1.getBalance( ) > account2.getBalance( ));
}
```

注意，上面两个定义的区别仅仅是第二个定义采用了“&”运算符将传值调用转变为引用调用，同时使用了常量修饰符。同样，函数声明也应该对形参做相应的修改。

常量参数支持自动错误检查。如果函数因错误代码导致对常量参数的无意修改，那么编译器就会给出错误信息。

`const` 修饰符可以用于任何类型的参数，但一般只针对引用调用参数使用（或者其他实参数据较大的情况，如数组）。

假设调用一个类的成员函数，如示例 7.2 中 `BankAccout` 类的成员函数：

```
BankAccount myAccount;
myAccount.input( );
myAccount.output( );
```

成员函数 `input` 会改变对象 `myAccout` 的成员变量，这有点类似引用调用方式——函数的调用改变调用对象的值。在有些情况下，我们不希望改变对象的成员变量。例如，我们不希望 `output` 成员函数改变对象的任何数据。这种情况下，可以采用 `const` 修饰符指示编译器，该函数的调用不会改变调用对象的任何值。

和限定函数参数不可修改一样，`const` 也可以用来限定成员函数，表示该成员函数不会对该对象做任何修改。如果该函数代码中无意修改了调用对象的值，编译器就会报错。在标记成员变量时，修饰符 `const` 应该放在函数声明的后面，结束分号之前，如下所示：

```
class BankAccount
{
public:
    ...
    void output( ) const;
    ...
}
```

`const` 用于成员函数

除函数声明外，对应的函数定义也需添加 `const` 修饰符，如下所示：

```
void BankAccount::output() const
{
    ...
}
```

函数定义的其余部分与示例 7.2 完全相同。

### 陷阱：const 的不一致用法

修饰符 `const` 要么不使用，要么应该使用的地方都使用。如果对某类型的形参使用了 `const` 修饰符，那么必须对其他类型相同且不会被改变的形参也使用 `const` 修饰符。此外，如果该类型为类类型，那么对其他所有不改变被调用对象值的成员函数都必须使用 `const` 修饰符。这其中的原因可以通过如下 `welcome` 函数的定义来解释：

```
Void welcome(const BankAccount& yourAccount)
{
    cout<< "Welcome to our bank.\n"
        << "The status of your account is:\n";
    yourAccount.output();
}
```

如果没有给成员函数 `output` 添加 `const` 修饰符，那么 `welcome` 函数将产生编译时错误。由函数声明可知，`welcome` 函数不会改变被调用对象的值。但是当编译器处理 `welcome` 函数的定义时，会认为 `welcome` 函数有可能改变对象 `yourAccount` 的值。这是因为，`welcome` 函数定义中出现的 `output` 成员函数没有 `const` 修饰符，因此编译器保守地认为它会改变 `yourAccount` 的值，和之前 `welcome` 函数的声明相冲突，因而报错。因此，如果对 `BankAccount` 类型的参数使用了 `const` 修饰符，那么 `BankAccount` 类中所有不会改变被调用对象值的成员函数都要加上 `const` 修饰符。具体到这个例子，成员函数 `output` 的声明和定义应该加上 `const` 修饰符。

在示例 7.4 中，我们重写了类 `BankAccount` 的定义，这里对所有应该使用 `const` 修饰符的地方都加上了 `const` 修饰符。示例 7.4 中还添加了之前提到的两个成员函数：`isLarger` 和 `welcome`，这两个函数都含有 `const` 修饰符。■

#### 示例 7.4 const 参数修饰符

```
1  #include <iostream>
2  #include <cmath>
3  #include <cstdlib>
4  using namespace std;

5  // 银行账户包含两部分数据：
6  // 账户的余额和利率。
7  class BankAccount
8  {
9  public:
10     BankAccount(double balance, double rate);
11     // 根据给定的参数初始化账户余额和利率。

12     BankAccount(int dollars, int cents, double rate);
```

该类是对示例 7.2 中类的重写，并在必要的地方使用了 `const` 修饰符。



```

13 // 将账户余额初始化为 $dollars.cents 形式。
14 // 对于负的账户余额, dollars 和 cents 都必须为负, 账户利率初始化为 rate。
15 BankAccount(int dollars, double rate);
16 // 账户余额初始化为 $dollars.00,
17 // 账户利率初始化为 rate。

18 BankAccount();
19 // 账户余额初始化为 $0.00, 账户利率初始化为 0.0%。
20 void update();
21 // 运行结果: 一年的利息被添加到账户余额中。
22 void input();
23 void output() const;
24 double getBalance() const;
25 int getDollars() const;
26 int getCents() const;
27 double getRate() const; // 返回账户利率。
28 void setBalance(double balance);
29 void setBalance(int dollars, int cents);
30 // 检查参数是否同时为负或者同时为正。

31 void setRate(double newRate);
32 // 如果 newRate 非负, 账户利率变为 newRate。否则终止程序。
33
34 private:
35 // 负的账户余额通过负的 dollars 和负的 cents 来表示。例如,
36 // -$4.50 将 accountDollars 设为 -4, 将 accountCents 设为 -50。
37 int accountDollars; // 账户的美元部分
38 int accountCents; // 美分部分
39 double rate; // 两者的百分比
40 int dollarsPart(double amount) const;
41 int centsPart(double amount) const;
42 int round(double number) const;
43 double fraction(double percent) const;
44 // 将百分数转变为小数。例如 fraction(50.3) 返回 0.503。
45 };

46 // 如果 account1 的余额大于 account2, 返回 true,
47 // 否则返回 false。
48 bool isLarger(const BankAccount& account1, const BankAccount& account2);

49 void welcome(const BankAccount& yourAccount);

50 int main()
51 {
52     BankAccount account1(6543.21, 4.5), account2;
53     welcome(account1);
54     cout << "Enter data for account 2:\n";
55     account2.input();
56     if(isLarger(account1, account2))
57         cout << "account1 is larger.\n";
58     else
59         cout << "account2 is at least as large as account1.\n";

60     return 0;

```

```

61 }
62
63 bool isLarger(const BankAccount& account1, const BankAccount& account2)
64 {
65     return (account1.getBalance() > account2.getBalance());
66 }
67
68 void welcome(const BankAccount& yourAccount)
69 {
70     cout << "Welcome to our bank.\n"
71         << "The status of your account is:\n";
72     yourAccount.output();
73 }
74
75 // 使用 iostream 和 cstdlib.
76 void BankAccount::output() const
77 {
78     < 此成员函数剩余部分的定义与示例 7.2 相同。 >
79 }
80
81 < 除了函数声明中添加了 const 修饰符的函数在定义中也应该添加 const 修饰符之外。 >

```

### 示例运行结果

```

Welcome to our bank.
The status of your account is:
Account balance: $6543.21
Rate: 4.5%
Enter data for account 2:
Enter account balance $100.00
Enter interest rate (NO percent sign): 10
account1 is larger.

```

### 自测练习

4. 下面的代码为示例 7.2 所示类 BankAccount 的成员函数 input 添加了 const 修饰符，这一做法为什么是不对的？

```

class BankAccount
{
public:
    void input() const;
    ...
}

```

5. 传值调用参数和常量引用调用参数之间有何差异？二者的声明如下：

```

void callByValue(int x);
void callByConstReference(const int& x);

```

6. 考虑如下的定义：

```

const int x = 17;
class A
{
public:
    A();
    A(int n);
}

```

```

int f( ) const;
int g(const A& x);
private:
    int i;
};

```

这里使用了三个 `const` 修饰符，每个 `const` 修饰符都能使编译器进行某种强制检查。对于这三种情况，编译器分别会进行怎样的检查？

### const 参数修饰符

如果将 `const` 修饰符放在引用调用参数的前面，该参数则被称为常量参数。添加了 `const` 修饰符，就告诉了编译器该参数不能被修改。如果在函数定义中，不小心更改了常量参数的值，编译器就会报错。对类类型的参数，如果函数不改变该参数的值，则应该使用常量引用调用参数，而不是使用传值调用参数。

如果成员函数不会改变被调用对象的值，安全起见可以在该成员函数的声明中添加 `const` 修饰符。此时，如果错误地在该成员函数的定义中对被调用对象做了修改，那么编译器会报错。`const` 修饰符应该放在函数声明的后面，结尾分号之前。同时，函数定义中也需添加 `const` 修饰符。

示例：

```

class Sample ,
{
public:
    Sample( );
    void input( );
    void output( ) const;
private:
    int stuff;
    double moreStuff;
};

int compare(const Sample& s1, const Sample& s2);

```

`const` 修饰符要么不使用，要么在所有该使用的地方都应该使用。对于类类型的参数和类的成员函数，应该在需要的地方都添加 `const` 修饰符。如果在类中可以添加 `const` 修饰的地方没有使用 `const` 修饰符，那么该类的所有地方都不应该再使用 `const` 修饰符。

### 内联函数

内联函数  
定义

可以在类的声明中给出成员函数的完整定义，这种定义方式被称为内联函数定义。一般来讲，内联函数只用于函数体很短的成员函数定义。示例 7.5 在示例 7.4 的基础上添加了内联函数的定义。

内联函数不仅仅是在定义上与一般的成员函数不同，而且编译器还会对内联函数做特殊处理。在编译时，内联函数的代码会被直接插入到每个调用内联函数的地方，从而减少函数调用的开销。

内联函数与一般函数功能相同，但却比一般函数更加高效，似乎所有成员函数都应该采用内联函数定义。然而事情并非如此，内联函数也有其自己的缺点。内联函数将类的定义和实现混为一块，违背了面向对象编程中的封装原则。另外，对于大多数编译器，内联函数只能在定义它的源文件中使用。

通常来讲，只有函数体较短的函数应采用内联形式定义。对于函数体较长的函数，内联形式的定义反而会降低效率，因为这会导致大片代码的到处复制。除此之外，是否使用内联函数完全取决于你自己。

任何函数都可以定义为内联函数。要将非成员函数定义为内联，只需在该函数的声明和定义前添加 `inline` 关键字。本书不对内联函数做进一步介绍。

### 示例 7.5 内联函数定义

```

1  #include <iostream>
2  #include <cmath>
3  #include <cstdlib>
4  using namespace std;

5  class BankAccount
6  {
7  public:
8      BankAccount(double balance, double rate);
9      BankAccount(int dollars, int cents, double rate);
10     BankAccount(int dollars, double rate);
11     BankAccount();
12     void update();
13     void input();
14     void output() const;

15     double getBalance() const { return (accountDollars+
16                                     accountCents*0.01); }

17     int getDollars() const { return accountDollars; }

18     int getCents() const { return accountCents; }

19     double getRate() const { return rate; }

20     void setBalance(double balance);
21     void setBalance(int dollars, int cents);
22     void setRate(double newRate);
23 private:
24     int accountDollars; // 账户的美元部分
25     int accountCents; // 美分部分
26     double rate; // 两者的百分比

27     int dollarsPart(double amount) const {return static_cast<int>(amount); }

28     int centPart(double amount) const;

29     int round(double number) const
30     { return static_cast<int>(floor(number+0.5)); }

```

在示例 7.4 的基础上添加了内联函数。

```
30     double fraction(double percent) const { return (percent/100.0);}
31 };
```

<其他函数定义与示例 7.4 相同。>

## 自测练习

7. 重写示例 7.3 中类 DayOfYear 的定义，将函数 getMonthNumber 和 getDay 改为内联形式。

## 静态成员

### 静态变量

在某些情况下，我们希望一个类的所有对象共享一个变量。例如，可能会需要一个变量，用来记录类中某个成员函数被所有对象调用的次数。这种被所有对象共享的变量称为**静态变量**，它可以用在同一个类的不同对象间的通信中，也可以协调这些对象的行为。类的静态变量具有全局变量的优势，但又不会像真的全局变量那样被滥用。特别需要指出的是，类的全局变量可以为私有的，这样就限制了只有类的对象才有权限访问这些变量。

如果一个函数不访问类的任何对象的数据，但却希望此函数作为类的成员，则可以将此函数声明为静态成员函数。静态成员函数可以像一般成员函数那样使用对象来调用，然而更为常见的方式是通过类名和作用域运算符，如下所示：

```
Server::getTurn()
```

由于静态函数的调用不需要类的对象，因此静态函数的实现中也不能用到该类的任何对象。此外，静态函数的定义不能用到任何的非静态变量或者非静态成员函数，除非该变量或函数是由一个局部对象或定义中出现的类对象调用的。如果这种解释很难理解的话，那么记住一个简单的规则：静态函数的定义不能使用到任何与被调用对象有关的内容。

### 初始化静态成员变量

示例 7.6 展示了静态成员变量和静态函数的用法。注意，静态变量是通过在变量的声明前添加关键字 `static` 来实现的，还应该注意，静态成员变量的初始化应该按下面的方式进行：

```
int Server::turn = 0;
int Server::lastServed = 0;
bool Server::nowOpen = true;
```

稍微解释一下上面的初始化语句。每个静态成员变量都必须在类定义之外初始化。另外，静态成员变量只能初始化一次。如示例 7.6 所示，私有静态成员变量也是在类外进行初始化的，这仿佛同私有的限定相悖。然而静态成员变量的初始化是由类的设计者完成的，而且往往就在类的定义文件中。这样，该类的使用者就不能在程序中再次对静态成员变量进行初始化，因为一个静态成员变量只能初始化一次。

## 示例 7.6 静态成员

```
1 #include <iostream>
```

```

2  using namespace std;

3  class Server
4  {
5  public:
6      Server(char letterName);
7      static int getTurn();
8      void serveOne();
9      static bool stillOpen();
10 private:
11     static int turn;
12     static int lastServed;
13     static bool nowOpen;
14     char name;
15 };

16 int Server::turn = 0;
17 int Server::lastServed = 0;
18 bool Server::nowOpen = true;

19 int main()
20 {
21     Server s1('A'), s2('B');
22     int number, count;
23     do
24     {
25         cout << "How many in your group? ";
26         cin >> number;
27         cout << "Your turns are: ";
28         for (count=0; count < number; count++)
29             cout << Server::getTurn() << ' ';
30         cout << endl;
31         s1.serveOne();
32         s2.serveOne();
33     }while (Server::stillOpen());

34     cout << "Now closing service.\n";

35     return 0;
36 }

37
38
39 Server::Server(char letterName): name(letterName)
40 { /* 特意留空 */ }

41 int Server::getTurn() ← 由于 getTurn 函数是静态函数，因此函数的定义
42 {                                中只能引用类的静态成员。
43     turn++;
44     return turn;
45 }
46 bool Server::stillOpen()
47 {
48     return nowOpen;
49 }

```

```

50 void Server::serveOne()
51 {
52     if (nowOpen && lastServed < turn)
53     {
54         lastServed++;
55         cout << "Server " << name
56             << " now serving " << lastServed << endl;
57     }
58     if (lastServed >= turn) // 所有人都已被服务。
59         nowOpen = false;
60 }

```

### 示例运行结果

```

How many in your group? 3
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? 2
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? 0
Your turns are:
Server A now serving 5
Now closing service.

```

需要注意的是，关键字 `static` 只在静态成员函数的声明中出现，在该成员函数的定义中则不再出现。

示例 7.6 是一个简单的客户等待服务模型，其中等待的客户排队在一个队列中，有两个服务器同时为这些客户服务。我们可以提出好多编程方案来充实这个示例，使之成为一个实用的例子。举一个简单的例子，我们可以认为 `getTurn` 函数产生的计数是熟食店或冰淇淋店发放给客户的排号卡，而两个服务器则是服务顾客的两个店员，我们假定顾客都是成批到达而且都排在同一个队列中等候（可能是购买他们喜欢的三明治或者特殊口味的冰淇淋）。

### 自测练习

- 能否将下面的函数定义添加到示例 7.6 的类 `Server` 中，并作为一个静态成员函数？为什么？

```

void Server::showStatus( )
{
    cout<< "Currently serving " << turn << endl;
    cout<< "server name " << name << endl;
}

```

## 嵌套类和局部类定义

本节介绍的内容仅用于参考，除了第 17 章对此有所涉及外，本书其他地方都没有用到。

嵌套类

可以在一个类中定义其他的类，这种在一个类中定义的其他类被称为嵌套类，示例如下：

```
class OuterClass
{
public:
    ...
private:
    class InnerClass
    {
        ...
    };
    ...
};
```

嵌套类可以是公有的，也可以是私有的。如果嵌套类为私有的（如上面的例子所示），那么嵌套类就不能在外层类之外使用。但不管嵌套类是私有的还是公有的，它都可以用在外层类的成员函数中。

由于嵌套类是在外层类的范围内定义的，嵌套类的名字（如上面例子中的 InnerClass）可以在外层类之外作为他用。如果嵌套类为公有的，在外层类之外，嵌套类将是一种新的数据类型，但此时它的类型应该是 `OuterClass::InnerClass`。

除了本书第 17 章稍微地提及外，本书不会用到嵌套类。<sup>2</sup>

局部类

类的定义还可以出现在一个函数的定义中，这时该类被称为局部类。局部类的定义限制在函数定义的范围内，一般不包含静态变量。同样，本书也不会用到局部类。

## 7.3 向量——标准模板库预览

“好吧，我会吃了它，”爱丽丝说，“如果它把我变大了，我就能拿到钥匙了；如果它把我变小了，我就能从门下面钻过去；不管哪种结果都能让我到花园中去。”

刘易斯·卡罗尔，《爱丽丝漫游仙境》

向量

向量可以看作是程序运行中大小可以改变的数组。我们知道，C++ 的数组一旦被定义，它的大小就不能改变。向量与数组有着相同的功能，但它的大小可以在程序运行中改变。

向量的定义来自标准模板库（STL）。本书第 16 章和第 19 章分别介绍模板和 STL 的相关知识。然而在详细了解模板和 STL 的详细知识之前，向量的用法却很容易掌握。使用向量无须太多类的知识。可以在学习完第 6 章后来阅读本节，另外，本节的学习

<sup>2</sup> 该部分内容在第 17 章名为“友元类以及类似的选择”节中。



不依赖本章前几节的内容。

## 向量基础

**声明向量变量** 和数组类似，向量也包含基本类型。同样，和数组一样，向量也是一系列基本类型的集合。但是，向量类型和向量变量的声明语法却与数组不同。

声明一个包含 `int` 类型的向量变量 `v`，如下所示：

```
vector<int> v;
```

**模板类** 其中 `vector<Base_Type>` 是一个模板类，表示可以用任何数据类型代替 `Base_Type` 来产生该数据类型的向量类。可以简单地认为，为向量设定基本类型就如同为数组指定基本类型一样。向量的基本类型可以是任何数据类型，包括类类型。`vector<int>` 是一个向量类，因此前面的声明是将 `v` 声明为 `vector<int>` 类型的变量，这一过程也包含了类 `vector<int>` 默认构造函数的调用，产生一个空的向量对象（没有元素）。

**v[i]** 向量元素序列与数组一样，都是从 0 开始的。同样，可以用方括号加上序列来读取或设置向量的元素。例如，下面的代码改变向量 `v` 的第 `i` 个元素，并输出修改后的值：

```
v[i] = 42;
cout << "The answer is" << v[i];
```

但是，与数组不同的是，方括号的使用有一个限制。可以用 `v[i]` 这样的方式改变向量 `v` 的第 `i` 个元素，但是却不能用这样的方式去初始化第 `i` 个变量。通俗地讲，方括号这种方式只能对已经初始化过的元素进行修改。为了给向量添加新的元素，必须使用成员函数 `push_back`。

**push\_back**

### 向量

向量的使用方式与数组非常类似，唯一的不同在于向量没有固定的大小。一旦需要额外的空间来存储新元素，向量的大小会自动增加。向量定义在标准模板库的 `std` 命名空间下，因此使用向量的程序必须在代码的开头包含如下的语句：

```
#include<vector>
using namespace std;
```

一个给定类型 `Base_Type` 的对象向量类记为 `vector<Base_Type>`。以下是两个向量声明的例子：

```
vector<int> v; // 默认构造函数，产生一个空的向量。
vector<Aclass> record(20); // 向量的构造函数调用。
// 类 Aclass 的默认构造函数初始化 20 个向量元素。
```

使用向量的成员函数 `push_back` 可以为向量添加元素，示例如下：

```
v.push_back(42);
```

一旦向量元素被初始化（无论是使用 `push_back` 函数，还是构造函数），则可以用方括号加上元素索引来访问该元素。

向量元素的增加是按照顺序进行的，第一个添加的是第 0 个元素，其次是第一个元素，依此类推。成员函数 `push_back` 将要添加的元素放在向量的最后。例如下面的代码对向量 `sample` 的前三个元素进行初始化：

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

大小 向量中元素的数量被称为该向量的大小。其中成员函数 `size()` 可以用来获取向量的大小。例如，在执行上面所示的语句之后，`sample.size()` 将返回 3。可以通过下面的语句将向量 `sample` 的元素全部输出：

```
for (int i = 0; i < sample.size(); i++)
    cout<< sample[i] << endl;
```

**size** 其中，函数 `size` 的返回值类型为 `unsigned int`，而不是一般的 `int` 类型。**unsigned** 一般来讲，`unsigned int` 可以在需要的时候自动转化为 `int` 类型，不过一些编译器还会给出警告信息，提示程序中需要的是一个 `int` 类型的值，但给出的却是一个 `unsigned int` 类型的值。如果希望程序非常安全，可以采用显式的类型转化，将返回值由 `unsigned int` 转换为 `int` 类型，或者将循环的循环变量 `i` 声明为 `unsigned int`，如下所示：

```
for (unsigned int i = 0; i < sample.size(); i++)
    cout<< sample[i] << endl;
```

示例 7.7 演示了向量的一些基本用法。

向量包含仅含一个整型参数的构造函数，它可以根据给定的参数 `n` 将向量的前 `n` 个元素进行初始化。例如，下面的语句：

```
vector<int> v(10);
```

向量 `v` 的前 10 个元素被初始化为 0，并且 `v.size()` 将返回 10。这样就可以使用 `v[i]` 来设置第 `i` ( $0 \sim 9$ ) 个元素的值。因此，下面的语句就可以紧跟在上面的声明语句之后，为该向量赋值。

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```

如果要为向量的第 10 个以及其后的元素赋值，就必须使用函数 `push_back`。

#### 陷阱：方括号的索引超过向量的大小

对于向量 `v`，如果索引 `i` 大于或等于 `v.size()`，那么元素 `v[i]` 将不存在，要使用的话，必须首先调用 `push_back` 向该向量添加元素。如果不给向量添加元素而试图直接对 `v[i]` 进行赋值，如下所示：

```
v[i] = n;
```

编译器或许不会给出错误信息，但程序在运行时肯定会出错。■

## 示例 7.7 向量的使用

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;

4  int main()
5  {
6      vector<int> v;
7      cout << "Enter a list of positive numbers.\n"
8           << "Place a negative number at the end.\n";

9      int next;
10     cin >> next;
11     while(next > 0)
12     {
13         v.push_back(next);
14         cout << next << " added. ";
15         cout << "v.size() = " << v.size() << endl;
16         cin >> next;
17     }

18     cout << "You entered:\n";
19     for (unsigned int i=0; i<v.size(); i++)
20         cout << v[i] << " ";
21     cout << endl;

22     return 0;
23 }

```

## 示例运行结果

```

Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8

```

当采用仅含一个 `int` 类型参数的构造函数初始化向量时，向量的元素将被初始化为 0。如果向量的基本类型为类类型，那么该向量初始化时将调用该类的默认构造函数。

## 提示：向量的赋值

向量间的赋值操作将会为赋值运算符左边的向量元素逐一地进行赋值操作（必要时，系统会自动增加被赋值的向量的空间，重置向量的大小）。因此，如果向量基本类型的赋值操作会产生元素的独立副本，那么向量的赋值操作也将产生赋值向量的独立副本，而不是赋值运算符右边赋值向量的引用或者别名。

需要注意的是，如果要使向量间的赋值产生完全独立的副本，就必须保证向量基本类型的赋值操作也能产生独立的副本，向量间赋值操作的结果取决于向量基本类型赋值操作的结果。■

向量定义在标准模板库的 `vector` 中，并放置在 `std` 命名空间中。因此，使用向量的程序必须包含如下的代码：

```
#include<vector>
using namespace std;
```

### 效率问题

#### 容量

每个向量都有自己的容量，即向量元素所占内存空间的大小。成员函数 `capacity()` 用来获取该向量的容量。需要注意的是，不要将向量的容量与向量的大小相混淆。向量的大小是指向量中实际包含的元素的个数，而向量的容量则是系统给该向量分配的元素个数。一般来讲，向量的容量总是大于或等于向量的大小。

一个向量超出原来的容量后，向量的容量会自动增加。容量到底增加多少，取决于具体的实现情况，但一般来讲，增加的容量总是比所需的容量大。通常的实现是，增加的容量是所需容量的两倍。因为容量的增加是一项复杂而开销较大的工作，一次分配较大的内存块比多次分配较小的内存块效率要高。

用户可以完全忽略向量容量的概念，而丝毫不会影响程序的执行。但是，如果特别关注程序的执行效率，我们可能希望自己管理向量的容量分配策略，而不是系统默认的每次分配所需容量的两倍。此时，可以使用成员函数 `reserve` 来显式地增加向量的容量。例如：

```
v.reserve(32);
```

将向量的容量设置为至少存放 32 个元素。

而下面的语句

```
v.reserve(v.size()+10)
```

则设置向量的容量为当前的大小再多 10 个元素。需要注意的是，可以采用 `reserve` 成员函数来增加向量的容量，但如果该成员函数的参数小于当前的容量时，系统却不会减少该向量的容量。

可以使用成员函数 `resize` 来改变向量的大小。例如，下面的语句将向量的大小设置为 24 个元素：

```
v.resize(24);
```

如果该向量之前的元素数少于 24，则新添加的元素会自动初始化。如果向量开始的大小超过 24，那么超出部分的元素都会丢失。这一过程，向量的容量会根据当前的大小自动调整。通过成员函数 `resize` 和 `reserve`，我们就可以根据需要控制向量的大小和容量。

### 向量的大小和容量

向量的大小是向量当前包含元素的个数，而向量的容量则是系统分配给向量的元素的个数。给定一个向量 `v`，可以通过 `v.size()` 和 `v.capacity()` 来获取该向量的大小和容量。

### 自测练习题

9. 如下的程序是否合法？如果合法，给出该程序的输出。

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

int main( )
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size( ); i++)
        v[i] = i;
    vector<int> copy;
    copy = v;
    v[0] = 42;

    for (i = 0; i < copy.size( ); i++)
        cout<< copy[i] << " ";
    cout<< endl;

    return 0;
}
```

10. 向量的大小和容量有什么不同？

### 本章小结

- 构造函数是类中一个特殊的成员函数，当声明一个类的对象时，构造函数将被自动调用；构造函数的名字必须与类名相同。
- 默认构造函数是没有参数的构造函数，应该为每个类定义一个默认构造函数。
- 类的成员变量可以是一个类的对象。如果一个类含有这种对象成员变量，那么该对象成员变量的初始化可以在外层类的初始化列表中进行。
- 对类类型的参数而言，常量引用调用参数比传值调用参数效率更高。
- 将很短小的函数定义为内联函数可以提高代码运行的效率。

- 静态成员变量属于类级变量，被该类的所有对象所共享。
- 向量与数组具有相同的功能，唯一的不同的是，其容量可以根据需要进行动态调整。

## 自测练习题答案

```
1. YourClass anObject(42, 'A'); // 合法
   YourClass anotherObject; // 合法
   YourClass yetAnotherObject(); // 存在问题
   anObject = YourClass(99, 'B'); // 合法
   anObject = YourClass(); // 合法
   anObject = YourClass; // 非法
```

标记为“存在问题”的语句在语法上并没有问题，但实际上它是一个函数声明语句，它声明一个名为 yetAnotherObject 的无参函数，该函数的返回值类型为 YourClass，而不是你认为的对象的声明语句。从这个角度看，我们可以认为该语句不合法。如果要声明一个使用默认构造函数来初始化名为 yetAnotherObject 的对象，正确的做法如下：

```
YourClass yetAnotherObject;
```

- 默认构造函数是没有任何参数的构造函数。并非所有的类都包含默认构造函数。如果类的定义中没有包含任何构造函数的定义，那么系统会给该类产生一个默认构造函数。另一方面，如果定义的类中包含了一个或者多个构造函数的定义，此时需手动为该类定义默认构造函数，否则该类就没有默认构造函数。
- 如果增加一个名为 BankAccount::digitToInt 的私有成员函数，那么成员函数 input 的定义会更容易给出。

```
// 使用 iostream 库。
void BankAccount::input()
{
    int dollars;
    char point, digit1, digit2;
    cout <<
        "Enter account balance (include cents even if .00) $";
    cin >> dollars;
    cin >> point >> digit1 >> digit2;
    accountDollars = dollars;
    accountCents = digitToInt(digit1)*10 + digitToInt(digit2);
    if (accountDollars < 0)
        accountCents = -accountCents;
    cout << "Enter interest rate (NO percent sign): ";
    cin >> rate;
    setRate(rate);
}

int BankAccount::digitToInt(char digit)
{
    return (static_cast<int>(digit) - static_cast<int>('0'));
}
```

4. 成员函数 `input` 会改变调用对象的值，因此如果给该函数添加 `const` 修饰符，编译器会报错。
5. 相同点：两种调用方式都能确保实参的值不被修改；不同点：如果参数的类型是结构体或者类类型，传值调用会产生实参的一个副本，因此其比引用调用消耗更多的内存空间。
6. 在语句 `const int x=7;` 中，关键字 `const` 指示编译器，变量 `x` 的值在程序运行过程中不能被修改。  
 在语句 `int f() const;` 中，关键字 `const` 告诉编译器，函数 `f` 不会对调用的值做任何修改。  
 在语句 `int g(const A& x);` 中，关键字 `const` 告诉编译器，函数 `g` 不会修改引用变量 `x` 的值。

```
7. class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    DayOfYear(int monthValue);
    DayOfYear();
    void input();
    void output();
    int getMonthNumber() { return month; }
    int getDay() { return day; }
private:
    int month;
    int day;
    void testDate();
};
```

8. 不能将其声明为静态成员函数，因为函数中使用到了成员变量 `name`。
9. 该程序合法。运行结果为：

```
0 1 2 3 4 5 6 7 8 9
```

注意，对向量 `v` 的修改不会改变向量 `copy` 的值。赋值语句 `copy = v;` 产生一个完全独立的副本。

10. 向量的大小是指当前向量中包含的元素的数目，而向量的容量是指系统分配给该向量的可以容纳的元素的个数。一般来讲，向量的容量总是大于向量的大小。

## 编程练习

1. 定义一个名为 `Month` 的类作为月份的抽象数据类型。该 `Month` 类包含一个 `int` 类型的成员变量，代表月份，并且包含如下的成员函数：可以用月份英文单词的前三个字母来设置月份的构造函数（含有三个参数）；使用一个 `int` 变量来设置月份的构造函数（含有一个参数）；一个默认构造函数；将月份按整数读入的输入函数；将月份按英文单词的前三个字母读入的输入函数；将月份按整数输出的输出函数；将月份按英文单词的前三个字母输出的输出函数；以及返回下个月份的成员函数，该成员函数返回值类型为 `Month`。定义完成后，编

写程序对其进行测试。

2. 重新定义编程练习 1 中的 Month 类，这次要求 Month 类包含三个成员变量，用于存储月份英文单词的前三个字母。同样，编写程序对 Month 类进行测试。
3. 我母亲经常会带一个红色的计算器去杂货店。计算器可以帮助她计算在店里购买的东西应付的钱数。该计算器只能显示四位数字，每一位对应一个增加按钮和重置按钮。如果钱数超过 99.99 美元，计算器的溢出指示器会变红。

编写一个类 Counter 用来模拟该红色的计算器，给出该类成员函数的定义。类的构造函数可以根据给定的实参构造一个能显示到该参数的 Counter 对象。也就是说，Counter(9999) 将产生一个能记录到 9999 的计算器。新构造的 Counter 对象的显示值为 0。成员函数 void reset() 将计算器的显示值重设为 0；成员函数 void incr1() 将显示值增加一个单位；成员函数 void incr10() 将显示值增加 10 个单位；成员函数 void incr100() 和 void incr1000() 分别将显示值增加 100 个单位和 1000 个单位。所需要执行的计数就是将每次的增量加入到私有成员变量中。成员函数 bool overflow() 用来检测计算器是否发生溢出（溢出是指增量之后，私有成员变量的值大于初始化时设定的最大显示值）。

使用定义的 Counter 类模拟我母亲的红色计算器。尽管该计算机的显示是四位整数，但其中最右的两位数值代表两位数的美分，最左的两位数字代表两位数的美元数。

分别为美分、10 美分、美元及 10 美元设置对应的按键。不幸的是，没有一种按键方案是特别容易记忆的。一个方案是采用“asdfo”，a 代笔美分，紧跟数字 1~9；s 代笔 10 美分，紧跟数字 1~9；d 代表美元，紧跟数字 1~9；f 代表 10 美元，紧跟数字 1~9；每一行数字后边都带有一个回车按钮，用于输入对应的数字。每次操作完毕之后，计算机都会报告当前是否发生溢出。此外，用户也可以按“o”来获取当前的溢出情况。

4. 你经营了几家分布在城区不同地方的热狗店。定义一个名为 HotDogStand 的类，该类包含一个成员变量用来记录热狗店的 ID，此外还包含一个成员变量用来记录该热狗店当前卖出的热狗数量。为该类创建一个构造函数，来初始化这两个成员变量。

再定义一个名为 JustSold 的成员函数，每次卖掉一个热狗的时候都调用该函数一次，增加当前卖掉的热狗的数量。再增加一个成员函数，返回当前卖掉的热狗的数量。

最后，增加一个静态成员变量来跟踪所有热狗店售出的热狗数，并编写一个静态成员函数来返回该静态成员变量的值。编写一个程序，采用至少三个热狗店对 HotDogStand 类进行测试。

5. 在很久以前，美丽的公主 Eve 有许多求婚者。她根据如下的方式来挑选自己中



意的新郎。首先，所有的求婚者按顺序排成一列，第一个求婚者是1号，第二个求婚者是2号，依此类推，直到 $n$ 。Eve从第一个求婚者开始，顺序数三个人（因为她的名字有三个字母），然后排除第三个求婚者的资格。接着Eve再顺序数三个人，将对应的求婚者排除，当数到列末时，她继续从列头开始。例如，有六个求婚者，那么挑选过程如下：

123456 最初的求婚者队伍，从1开始。

12456 3号求婚者被删除，继续从4号求婚者开始。

1245 6号求婚者被删除，继续从1号开始。

125 4号求婚者被删除，继续从5号开始。

15 2号求婚者被删除，继续从5号开始。

1 5号求婚者被删除，1号求婚者是选定的新郎。

如果有 $n$ 个求婚者，编写一个使用向量的程序，计算你应该站第几个位置才能最终与公主结婚。可以使用以下的几个函数：

```
v.erase(iterator); // 删除向量位置 iterator 上的元素。
```

例如，要使用erase函数删除向量theVector从头开始的第四个元素，可以采用如下语句：

```
theVector.erase(theVector.begin() + 3);
```

这里使用数字3而不是4是因为，向量的第一个元素位于索引0上。

6. 本题目要求首先完成第6章的编程练习7，即实现一个名为Pizza的类。再编写一个名为Order的类，该类包含一个类型为Pizza的私有向量，用以跟踪顾客的订单。其中，每个订单可能会包含多个比萨。为Order类编写合适的成员函数，以方便Order的使用者添加比萨（比萨种类包含deep dish、hand tossed或pan，大小包含小、中、大，以及配料的种类），可以采用常量来记录比萨的类型以及大小。编写输出函数对Order类进行输出，并给出订单的总价格。最后，编写完整的程序对Order类进行测试。
7. 重做第6章的编程练习8，为Money类添加默认构造函数，将美元数设置为0美元、0美分。再编写一个带有两个参数的构造函数，分别对美元数和美分数进行初始化。最后，编写完整的程序，对完成的默认构造函数和带有两个参数的构造函数进行测试。
8. 编写程序，以直方图的形式输出一个班级学生作业的得分情况。该程序应以int类型读入每个学生的作业得分，并将其存入一个向量中。程序会一直读取学生的作业得分，直到用户输入-1。此后，程序读取该向量元素的值，并计算出最终的直方图。计算直方图时，设定作业的最低得分为0，而最高得分则应由程序根据用户的计算得出。直方图计算完毕之后，将该直方图输出到终端上。关于直方图的计算方法，可以参考第5章的编程练习7。
9. 2009年以前，美国都会在信封上使用条形码。邮政服务采用名为POSTNET的

格式来表示五位或者多位的邮政编码。条形码包含一些长的或者短的条，如下图所示：



本程序采用一系列的数字来代表信封上的条形码。数字 1 代表一个长条，数字 0 则代表一个短条。因此，以上的条形码对应的数字序列为：

```
110100010100010101000010011
```

条形码对应的数字序列，其首位和末位总是 1，除去首位和末位后，余下 25 位数字。这些数字五位一组：

```
10100 10100 01010 11000 01001
```

接下来，考虑每组的五位数字，每一组总是包含两个数字 1。五位数字序列的每一位都对应一个数字，从左到右，分别为 7、4、2、1、0。数字序列的每一位乘以每一位代表的数字后相加就会得到邮政编码中的一位。下面给出了 10100 编码代表的数：

条形码数字序列	1	0	1	0	0
数值	7	4	2	1	0
二者相乘	7	0	2	0	0

邮政编码的某一位 =  $7 + 0 + 2 + 0 + 0 = 9$

对五组数字序列重复以上计算，从而得到一个完整的邮政编码。注意，如果某组数字序列计算得到的值为 11，那么对应邮政编码中的 0。本例给出的条形码代表的邮政编码为 99504。或许 POSTNET 格式看起来太过复杂，但它可以很容易地监测邮政编码扫描过程中是否发生了错误。

编写一个邮政编码类，用来对美国邮政服务印刷在信封上的五位数字条形码进行编码和解码。该类应包含两个构造函数，第一个构造函数以 `int` 类型读取一个邮政编码；第二个构造函数以之前提到过的 0、1 字符串序列读取邮政编码。尽管这里提供了两种方式来输入邮政编码，但类的实现中只能采用一种方式来记录邮政编码（必须在 0、1 数字序列和 `int` 类型的邮政编码中选择一种）。此外，该类需包含至少两个公有成员函数：一个用来返回 `int` 类型的邮政编码，另一个用来返回 0、1 数字序列的邮政编码。其他的所有辅助成员函数都应设置为私有的。最后编写一个完整的程序对该类进行测试。

- 首先，完成第 6 章的编程练习 12。修改 `main` 函数，采用一个循环来让用户决定要生成多少个 `BoxOfProduce` 的对象。每箱蔬果应包含三捆从以下列表中随机选择的蔬菜或水果：树番茄、西兰花、西红柿、猕猴桃、甘蓝。添加一个

菜单，从而用户可以结束蔬果箱的选择。此外，该菜单也允许用户对蔬果箱中的蔬果种类进行更换。

如果蔬果箱包含树番茄，程序还应自动地附加对应的菜谱。但需注意的是，总共只有五份菜谱。为类 `BoxOfProduce` 添加一个静态成员变量，用以记录剩余的菜谱数。此外，添加一个成员变量用以记录当前的蔬果箱是否包含菜谱。修改 `output` 函数，当蔬果箱包含菜谱的时候，额外输出 “salsa verde recipe”。最后，为该类添加对应的处理逻辑，如果当前的蔬果箱包含树番茄，在有菜谱的情况下为该蔬果箱自动添加菜谱。需要注意的是，每个蔬果箱最多只能包含一个菜谱，即使当前的蔬果箱包含多份树番茄。

编写程序来测试你编写的类，选择程序提供的菜单，不断创建包含树番茄的蔬果箱，直到所有的菜谱都赠送完毕。



# 运算符重载、友元和引用

# 8

## 8.1 基本运算符的重载 264

重载基础 264

提示：构造函数可以返回一个对象 269

返回常量类型 270

一元运算符的重载 272

作为成员函数的运算符重载 272

提示：类可以访问其所有对象 275

重载函数调用符() 275

陷阱：重载 &&、||及逗号运算符 275

## 8.2 友元函数与自动类型转换 276

构造函数的自动类型转换 276

陷阱：成员运算符和自动类型转换 277

友元函数 277

友元类 280

陷阱：不支持友元的编译器 281

## 8.3 引用和其他运算符重载 282

引用 282

提示：返回类的成员变量 283

重载 ">>" 和 "<<" 284

提示：应使用什么样的返回值类型 289

赋值运算符 291

重载自增和自减运算符 291

重载数组运算符[] 294

基于左值和右值的重载 296

# 第8章 运算符重载、友元和引用

永恒的真相并不会正确或永恒，除非每个新的社会环境都赋予它新的意义。

富兰克林·罗斯福，在宾夕法尼亚大学的讲话（1940年9月20日）

## 概述

本章将对类定义中用到的一些工具进行介绍。首先介绍的是运算符重载，可以对诸如“+”、“==”等运算符进行重载，使它们可以应用于类的对象之间；其次介绍的是友元，友元函数虽不是类的成员函数，但却能访问类的私有数据。本章还将介绍如何提供自动类型转换，从而将其他类型的数据自动转换为你所定义的类型。

如果你还没有学习第5章数组的相关知识，那么可以跳过8.3节数组运算符重载的相关内容。因为这部分内容的学习要求必须对数组的相关知识有所了解。

## 8.1 基本运算符的重载

他是一个老练的行家。

萨德歌中的一行（由萨德·阿杜和雷·圣约翰创作）

运算符和  
函数  
操作数  
语法糖

编程时经常用到的诸如“+”、“-”、“%”、“==”等运算符实际上是一种特殊的函数，只不过它们的调用形式与一般函数不同。我们使用  $x + 7$ ，而不是  $+(x, 7)$ ，但实际上，运算符“+”是一个带有两个参数（通常称为操作数）且返回单一值的函数。由此可见，运算符其实不是必需的，它的功能完全可以用函数进行取代。我们完全可以用  $+(x, 7)$  或者  $\text{add}(x, 7)$  的方式来代替“+”运算符。运算符是语法糖的一个典型例子，意思是采用了一种人们比较习惯的语法形式。由于人们习惯于通常的运算符使用语法  $x + 7$ ，而高级语言 C++ 设计的一个出发点就是可以让人们更容易地编写计算机程序。因此，语法糖是一个不错的主意，至少是一个很容易被采纳的主意。C++ 语言中，可以对“+”、“==”等运算符进行重载，这样就可以用它们操作类类型的操作数，运算符的重载与一般函数的重载类似。下面通过一个例子来进行介绍。

### 重载基础

示例 8.1 包含了一个类的定义，该类的值是美元数，如 \$9.99 或者 \$1567.29。Money 类和示例 7.2 中的 BankAccount 类十分类似。二者使用相同的方法来表示美元数，都是用两个 int 来分别表示美元数的整数部分和小数部分。二者的私有帮助函数也相同，其他如构造函数、取值函数和赋值函数也都非常类似。Money 类中不同的部分采用了“+”、“-”运算符的重载，从而可以对 Money 对象进行“+”、“-”操作。除此之外还对“==”运算符进行了重载，可以用来对两个 Money 类的对象进行比较，

看它们是否代表相同的美元数。下面我们来看看这些重载的运算符。

### 示例 8.1 运算符重载

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  // 表示美元数目的 Money 类。
6  class Money
7  {
8  public:
9      Money();
10     Money(double amount);
11     Money(int theDollars, int theCents);
12     Money(int theDollars);
13     double getAmount() const;
14     int getDollars() const;
15     int getCents() const;
16     void input(); // 同时读入美元符号和数字。
17     void output() const;
18 private:
19     int dollars; // 一个负的美元数要求：美元部分为负，
20     int cents; // 且美分部分也为负，例如负的 $4.50 表示为 -4 和 -50。

21     int dollarsPart(double amount) const;
22     int centsPart(double amount) const;
23     int round(double number) const;
24 };

25 const Money operator +(const Money& amount1, const Money& amount2);
26 const Money operator -(const Money& amount1, const Money& amount2);
27 const Money operator ==(const Money& amount1, const Money& amount2);
28 const Money operator -(const Money& amount1);

29 int main()
30 {
31     Money yourAmount, myAmount(10, 9);
32     cout << "Enter an amount of money: ";
33     yourAmount.input();
34     cout << "Your amount is ";
35     yourAmount.output();
36     cout << endl;
37     cout << "My amount is ";
38     myAmount.output();
39     cout << endl;
40     if (yourAmount == myAmount)
41         cout << "We have the same amounts.\n";
42     else
43         cout << "One of us is richer.\n";

```

一元运算符，参见“一元运算符的重载”小节。

关于 const 类型返回值的内容，请参考“返回常量类型”小节

```
44 Money ourAmount = yourAmount + myAmount;
45 yourAmount.output(); cout << " + "; myAmount.output();
46 cout << " equals "; ourAmount.output(); cout << endl;

47 Money diffAmount = yourAmount - myAmount;
48 yourAmount.output(); cout << " - "; myAmount.output();
49 cout << " equals "; diffAmount.output(); cout << endl;

50 return 0;
51 }
```

注意，这里需要使用取值和赋值函数。

```
52 const Money operator +(const Money& amount1, const Money& amount2)
53 {
54     int allCents1 = amount1.getCents() + amount1.getDollars()*100;
55     int allCents2 = amount2.getCents() + amount2.getDollars()*100;
56     int sumAllCents = allCents1 + allCents2;
57     int absAllCents = abs(sumAllCents); // 可以为负数。
58     int finalDollars = absAllCents / 100;
59     int finalCents = absAllCents % 100;

60     if (sumAllCents < 0)
61     {
62         finalDollars = -finalDollars;
63         finalCents = -finalCents;
64     }
65     return Money(finalDollars, finalCents);
66 }
```

如果对该 return 语句有疑问，参见名为“构造函数可以返回一个对象”的提示。

```
67 // 使用了 cstdlib 库。
68 const Money operator -(const Money& amount1, const Money& amount2)
69 {
70     int allCents1 = amount1.getCents() + amount1.getDollars()*100;
71     int allCents2 = amount2.getCents() + amount2.getDollars()*100;
72     int diffAllCents = allCents1 - allCents2;
73     int absAllCents = abs(diffAllCents);
74     int finalDollars = absAllCents / 100;
75     int finalCents = absAllCents % 100;

76     if (diffAllCents < 0)
77     {
78         finalDollars = -finalDollars;
79         finalCents = -finalCents;
80     }

81     return Money(finalDollars, finalCents);
82 }
```

```
83 bool operator ==(const Money& amount1, const Money& amount2)
84 {
85     return ((amount1.getDollars() == amount2.getDollars())
86         && (amount1.getCents() == amount2.getCents()));
87 }
```

```
88 const Money operator -(const Money& amount)
```

```

89 {
90     return Money(-amount.getDollars( ), -amount.getCents( ));
91 }

92 Money::Money( ): dollars(0), cents(0)    如果愿意的话, 可以将这些简短的构造函数定
93 /* 函数体有意留空 */                    义为内联函数, 参见本书的第7章。

94 Money::Money(double amount)
95         : dollars(dollarsPart(amount)), cents(centsPart(amount))
96 /* 函数体有意留空 */

97 Money::Money(int theDollars)
98         : dollars(theDollars), cents(0)
99 /* 函数体有意留空 */
100
101 // 使用了 cstdlib 库。
102 Money::Money(int theDollars, int theCents)
103 {
104     if ((theDollars < 0 && theCents > 0) ||
105         (theDollars > 0 && theCents < 0))
106     {
107         cout << "Inconsistent money data.\n";
108         exit(1);
109     }
110     dollars = theDollars;
111     cents = theCents;
112 }

113 double Money::getAmount( ) const
114 {
115     return (dollars + cents*0.01);
116 }

117 int Money::getDollars( ) const
118 {
119     return dollars;
120 }

121 int Money::getCents( ) const
122 {
123     return cents;
124 }

125 // 使用了 cstdlib 和 iostream 库。
126 void Money::output( ) const
127 {
128     int absDollars = abs(dollars);
129     int absCents = abs(cents);
130     if (dollars < 0 || cents < 0)
131         //accounts for dollars == 0 or cents == 0
132         cout << "$-";
133     else
134         cout << '$';
135     cout << absDollars;
136 }

```



```

135     if (absCents >= 10)
136         cout << '.' << absCents;
137     else
138         cout << '.' << '0' << absCents;
139 }

140 // 使用了 iostream 和 cstdlib 库。
141 void Money::input()
142 {
143     char dollarSign;
144     cin >> dollarSign;
145     if (dollarSign != '$')
146     {
147         cout << "No dollar sign in Money input.\n";
148         exit(1);
149     }

150     double amountAsDouble;
151     cin >> amountAsDouble;
152     dollars = dollarsPart(amountAsDouble);
153     cents = centsPart(amountAsDouble);
154 }

155 int Money::dollarsPart(double amount) const
156     < 其他部分与示例 7.2 中的 BanckAccount::dollarsPart 函数完全相同。 >
157 int Money::centsPart(double amount) const
158     < 其他部分与示例 7.2 中的 BankAccount::centsPart 函数完全相同。 >

159 int Money::round(double number) const
160     < 其他部分与示例 7.2 中的 BankAccount::round 函数完全相同。 >

```

### 示例运行结果

```

Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36

```

### 如何重载运算符

可以对“+”运算符进行重载，从而使之能够对类类型的参数进行相加操作。运算符“+”的重载与普通函数的重载类似，仅在语法上有稍许不同：其中使用“+”作为函数名，并在函数名前使用关键字 operator。运算符“+”的重载声明如下：

```
const Money operator +(const Money& amount1, const Money& amount2);
```

其中操作数（即函数的参数）都为 Money 类型的常量引用参数，实际上操作数可以为任意类型，只是其中之一必须是类类型的参数。一般来讲，参数可以是值传递调用参数，也可以是引用传递调用参数，并且也可以采用 const 修饰符进行限定。但是基于效率角度考虑，对类类型的操作数来讲，一般使用常量引用传递调用参数。在上面的例子中，函数的返回值类型为 Money，但通常返回值可以为任意类型，包括 void 类型。在返回值类型 Money 的前面还有一个 const 修饰符，本章后面会对此做

详细解释，这里可以暂且忽略它。

这里还需注意，重载的“+”、“-”运算符不是 Money 类的成员函数，因此也就不能访问类 Money 的私有成员，所以重载函数的定义中使用了取值和赋值函数。本章的后续内容中，还将介绍另外一种形式的运算符重载：将运算符作为成员函数进行重载。这两种运算符的重载方式都有各自的优缺点。

Money 类的实现中，二元运算符“+”和“-”的重载定义可能比大家想象的要复杂一些，因为其中一部分代码用来处理美元数为负的情况。

一元运算符“-”的重载将在本章的“一元运算符的重载”小节中进行介绍。

Money 类也对运算符“==”进行了重载，这样就可以对两个 Money 对象进行比较。由于该重载函数的返回值为 bool 类型，因此“==”完全可以按照通常的方式使用，例如用在 if-else 语句中。

仔细阅读示例 8.1 的 main 函数后，可以看出使用重载后的二元运算符“+”、“-”和“==”对类 Money 的对象进行操作，同对预定义类型（如 int、double 等）进行操作的方法完全一样。

我们可以对绝大多数运算符进行重载，其中一个重要的限制条件是至少一个操作数是类类型。例如，可以对运算符“%”进行重载，用来对两个 Money 类的对象进行操作，也可以对一个 Money 类的对象和一个 double 类型进行操作，但不能将其应用于两个 double 类型的数据之间。

### 运算符重载

二元运算符，如 +、-、/、% 等，其实是一类特殊的函数，这种函数的调用语法与普通的函数调用不同。对于二元运算符，函数的参数分别放在运算符的前面和后面；而对于普通的函数来讲，函数的参数放在函数名之后的圆括号中。运算符重载的定义与普通函数的定义基本一样，除了重载运算符时，需要在运算符之前使用关键字 operator 以外。预定义运算符，如“+”、“-”等，可以被重载，用于类对象的操作中，示例 8.1 给出了“+”、“-”和“==”运算符重载的例子。

**提示：**构造函数可以返回一个对象

构造函数通常被认为是一个 void 类型的函数，然而构造函数是一种具有特殊性质的函数，在某些情况下让构造函数返回一个值更符合实际的需要。注意示例 8.1 中重载运算符“+”的返回语句如下：

```
return Money(finalDollars, finalCents);
```

该语句的返回表达式是对类 Money 构造函数的调用，尽管构造函数常常被认为是一个 void 类型的函数，但由于构造函数负责类对象的构建，因此可以认为被构建的类对象是构造函数的返回值。如果上面的例子不好理解，那么可以考虑该语句等价的另外一种形式，如下所示：

```
Money temp;
```

```
temp = Money(finalDollars, finalCents);
return temp;
```

### 匿名对象

上例中的表达式 `Money(finalDollars, finalCents)` 通常被称为匿名对象, 因为该对象没有一个明确的名字。但匿名对象是一个完整的对象, 它的用法也与其他对象完全相同, 例如可以调用其中的方法:

```
Money(finalDollars, finalCents).getDollars()
```

该调用返回 `finalDollars` 的整数值。■

### 习题

1. 二元运算符和函数之间有何区别?
2. 假设要重载运算符“<”, 使之能够用于示例 8.1 中定义的类 `Money` 的对象, 那么应该对 `Money` 类的定义做哪些修改?
3. 能通过对运算符加法“+”的重载来改变整数之间的加法行为吗? 为什么?

### 返回常量类型

注意示例 8.1 中运算符重载声明中的返回值类型。例如, 下面是加号运算符的重载声明:

```
const Money operator +(const Money& amount1,
                      const Money& amount2);
```

本小节讨论声明语句最前面的 `const` 关键字。在进入具体的讨论之前, 我们必须了解关于返回值的一些知识。这里假设关键字 `const` 在加号运算符的重载声明和重载定义中都不出现, 即假设其重载声明如下:

```
Money operator +(const Money& amount1, const Money& amount2);
```

我们看看可以对该返回值做一什么操作。

当返回一个对象时, 如 `(m1+m2)` (这里 `m1` 和 `m2` 都是类 `Money` 的对象), 那么返回的对象就可以调用其成员函数, 这些成员函数有可能改变对象的值, 也可能不会。例如:

```
(m1 + m2).output();
```

上面的调用没有任何问题, 而且也不会改变对象 `(m1+m2)` 的值。如果忽略了加号运算符重载声明中第一个 `const` 关键字, 那么下面的语句也是合法的, 但会改变对象的值:

```
(m1 + m2).input();
```

因此, 对象是可以被改变的, 即使它们没有和任何变量进行关联。上面的调用即改变了对象 `(m1+m2)` 的值, 尽管该对象是匿名对象。可以从以下角度来理解对象的改变: 对象内部包含着相关的可以被改变的成员变量。

现在我们假设如示例 8.1 所示, 运算符重载声明和定义中的返回值类型前使用了 `const` 关键字。例如, 下面对加号运算符的重载声明:

```
const Money operator +(const Money& amount1, const Money& amount2);
```

## 常量返回值

其中开头的 `const` 是一种新用法，称为常量返回值或者返回常量类型。这里 `const` 表示，返回的值不能被修改。例如，考虑如下的代码：

```
Money m1(10.99), m2(23.57);
(m1 + m2).output();
```

`output` 函数的调用没有任何问题，因为它不会改变对象 `(m1+m2)` 的值。但对于如下的语句，则会产生编译错误：

```
(m1 + m2).input();
```

那么为什么要使用常量返回值呢？因为它提供了一种自动错误检查。当我们创建对象 `(m1+m2)` 时，不希望修改它的值。本例中，对对象 `(m1+m2)` 的修改可能不会有任何问题，但在特定的情况下，返回的对象其实是一个已经存在的对象，此时对对象的无意修改往往会产生严重的问题。这部分内容在 8.3 节会有相关介绍。

乍眼一看，这种防止对象被改变的方法似乎有点过于严格，因为我们可以编写如下代码：

```
Money m3;
m3 = (m1 + m2);
```

而且你可能希望改变 `m3` 的值。这个完全可以做到，我们可以采用如下的语句进行：

```
m3 = (m1 + m2);
m3.input();
```

其中对象 `m3` 和对象 `(m1+m2)` 是完全不同的两个对象。赋值运算符并不是将 `m3` 和 `(m1+m2)` 作为一个对象进行处理的。相反，它是将对象 `(m1+m2)` 成员变量的值复制到对象 `m3` 中，这样就产生了两个完全独立的对象。对于类类型的对象来讲，默认的赋值运算符不是将两个对象看作一个对象处理，而是将一个对象成员变量的值复制到另一个对象中。

这个区分虽然很微小，但却意义重大，这有助于理解一个类类型的变量和一个类类型的对象并不是一回事。对象是一个类类型的值，并且可以存储在一个类类型的变量中，但二者并不相同。考虑下面的语句：

```
m3 = (m1 + m2);
```

变量 `m3` 和它的值 `(m1+m2)` 是不同的，就如同下面的语句中变量 `n` 和值 5 一样：

```
int n = 5;
```

或者

```
int n = (2 + 3);
```

理解常量返回类型可能需要一段时间。一般来讲，在返回值为类类型的时候，除非有明确的理由需要返回非常量类型，否则，一般比较好的做法是返回常量类型。对于大多数比较简单的程序来说，这么做不会对程序产生什么影响，返回会排除一些细小的错误。

需要注意的是，虽然将基本类型作为常量返回类型没有任何问题，但却毫无意义。当函数或运算符返回基本类型时，使用 `const` 修饰符不会产生任何作用。如果返回类型是基本类型之一，如 `int`、`double` 或者 `char`，此时将返回一个值（如 5、5.5 或

者 'A')，而不会返回一个变量或者类似变量。与变量不同，值不能被改变，比如你不能改变值“5”本身。无论返回值类型前是否有 `const` 修饰符，基本类型的值都是不能被改变的。另一方面，类类型的值（也就是对象）是可以被改变的，因为对象包含成员变量，所以此时修饰符的使用是有意义的。

## 自测练习题

4. 假设省略了 `Money` 类中加号运算符重载定义和声明中开头的 `const` 修饰符，即不再是常量返回类型。那么下面的语句合法吗？

```
Money m1(10.99), m2(23.57), m3(12.34);
(m1 + m2) = m3;
```

如果重载定义和声明如示例 8.1 所示，也即返回常量类型，上面的语句还合法吗？

## 一元运算符的重载

### 一元运算符

除了二元运算符（如  $x+y$  中的  $+$  运算符）之外，C++ 还包含一元运算符，如负号“ $-$ ”。一元运算符是指只有一个操作数的运算符。在下面的语句中，一元运算符“ $-$ ”使变量  $x$  的值等于变量  $y$  的值取负：

```
x = -y;
```

自增运算符“ $++$ ”和自减运算符“ $--$ ”也是一元运算符的例子。

可以像重载二元运算符一样对一元运算符进行重载。例如，示例 8.1 就对负号运算符进行了重载。这样运算符“ $-$ ”就同时拥有一元和二元运算符两个版本。假定程序中包含如下的语句：

```
Money amount1(10), amount2(<), amount3;
```

那么下面的语句就是设置 `amount3` 的值为 `amount1` 减去 `amount2`：

```
amount3 = amount1 - amount2;
```

下面的语句就会在屏幕上输出 \$4.00：

```
amount3.output();
```

另一方面，下面的语句设置 `amount3` 的值为负的 `amount1`：

```
amount3 = -amount1;
amount3.output();
```

### ++ 和 --

同样可以如重载负号运算符一样重载“ $++$ ”和“ $--$ ”运算符。需要注意的是，如果如示例 8.1 中重载负号运算符一样重载“ $++$ ”和“ $--$ ”运算符，那么“ $++$ ”和“ $--$ ”可以用于操作数前面，即  $++x$  和  $--x$ 。本章后续的内容还将介绍如何重载“ $++$ ”和“ $--$ ”运算符，使之可以放在操作数后面。

## 作为成员函数的运算符重载

在示例 8.1 中，所有的运算符重载函数都是作为一般函数进行重载的，在类之外进行定义。同样，我们可以将运算符重载函数作为类的成员函数，具体如示例 8.2 所示。

## 示例 8.2 作为成员函数的运算符重载

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  // 表示美元数的 Money 类。
6  class Money
7  {
8  public:
9      Money();
10     Money(double amount);
11     Money(int dollars, int cents);
12     Money(int dollars);
13     double getAmount() const;
14     int getDollars() const;
15     int getCents() const;
16     void input(); // 同时读入美元符号和美元数。
17     void output() const;
18     const Money operator +(const Money& amount2) const;
19     const Money operator -(const Money& amount2) const;
20     bool operator ==(const Money& amount2) const;
21     const Money operator -() const;
22 private:
23     int dollars; // 一个负的美元数要求：美元部分为负，
24     int cents; // 且美分部分也为负，例如负的 $4.50 表示为 -4 和 -50。

25     int dollarsPart(double amount) const;
26     int centsPart(double amount) const;
27     int round(double number) const;
28 };

29 int main()
30 {
31     <如果 main 函数的定义与示例 8.1 相同，那么示例运行结果也与示例 8.1 相同。>
32 }
33
34 const Money Money::operator +(const Money& secondOperand) const
35 {
36     int allCents1 = cents + dollars*100;
37     int allCents2 = secondOperand.cents + secondOperand.dollars*100;
38     int sumAllCents = allCents1 + allCents2;
39     int absAllCents = abs(sumAllCents); // 美元数可以为负值。
40     int finalDollars = absAllCents / 100;
41     int finalCents = absAllCents % 100;
42     if (sumAllCents < 0)
43     {
44         finalDollars = -finalDollars;
45         finalCents = -finalCents;
46     }

```

这是实例 8.1 的重新实现，其中将重载的运算符作为成员函数。

调用对象自身作为第一个操作数。

```

47     return Money(finalDollars, finalCents);
48 }

49 const Money Money::operator -(const Money& secondOperand) const
50     < 其余部分的定义作为自测练习题 5。 >

51 bool Money::operator ==(const Money& secondOperand) const
52 {
53     return ((dollars == secondOperand.dollars)
54             && (cents == secondOperand.cents));
55 }

56 const Money Money::operator -( ) const
57 {
58     return Money(-dollars, -cents);
59 }

60     < 其他成员函数的定义与示例 8.1 完全相同。 >

```

---

注意，当二元运算符作为成员函数进行重载时，只有一个参数，而不是两个。调用对象相当于以前函数的第一个参数。例如，下面的代码：

```

Money cost(1, 50), tax(0, 15), total;
total = cost + tax;

```

当加号运算符“+”作为成员函数重载时，表达式 `cost+tax` 中的变量 `cost` 是调用对象，而变量 `tax` 则是运算符“+”的唯一参数。

运算符“+”作为成员函数的定义在示例 8.2 中已经给出。注意定义中的如下语句：

```
int allCents1 = cents + dollars * 100;
```

该语句中 `cents` 和 `dollars` 是调用对象的成员变量，也就是操作数 `cost` 的成员变量。如果将重载运用于如下的表达式：

```
cost + tax
```

那么，`cents` 就是 `cost.cents`，而 `dollars` 则是 `cost.dollars`。

注意，由于第一个操作数是调用对象本身，在大多数情况下，应该在运算符重载声明和定义的后面加上 `const` 修饰符。只要运算符重载函数不更改调用对象的值，就应该在该函数的声明和定义后面加上 `const` 修饰符，如实例 8.2 所示。

将运算符重载函数作为成员函数来进行声明和定义看起来似乎有点奇怪，但不难掌握。许多编程语言专家都建议将运算符重载函数作为成员函数进行声明和定义，而不是像实例 8.1 那样作为非成员函数。这样做的好处是更加符合面向对象的编程思想，并且也更高效，因为成员函数可以直接访问类的成员变量，而无须使用取值和赋值函数。但是，在后面的讨论中将会看到，作为成员函数的运算符重载也有明显的缺点。

**提示：**类可以访问其所有对象

一个成员函数或者成员运算符可以直接访问调用对象的所有私有成员变量（或者私有成员函数）。然而，不仅如此，还可以直接访问该类任何对象的任何私有成员变量（或者私有成员函数）。

例如，以下是示例 8.2 Money 类中加号运算符重载定义中的几行语句：

```
const Money Money::operator +(const Money& secondOperand) const
{
    int allCents1 = cents + dollars*100;
    int allCents2 = secondOperand.cents + secondOperand.dollars
                    * 100
}
```

这里，加号运算符是作为成员函数进行重载的，因此函数体第一条语句中，变量 cents 和 dollars 是调用对象的成员变量。然而，通过成员名直接访问对象 secondOperand 的私有成员变量也是合法的，也就是该函数体中的第二条语句：

```
int allCents2 = secondOperand.cents + secondOperand.dollars * 100;
```

因为 secondOperand 是 Money 类的对象，而这条语句出现在类 Money 的成员运算符的定义中。许多编程初学者往往错误地认为只能访问调用对象的私有成员，而没有意识到在类的成员函数定义中，也可以直接访问任何该类对象的所有成员。■

### 自测练习題

5. 完成示例 8.2 中二元运算符“-”的重载定义。

### 重载函数调用符()

函数调用运算符()必须作为成员函数进行重载。函数调用运算符重载之后，就可以以函数调用的方式来使用类的对象。如果类 AClass 对函数调用运算符()进行了重载，并且包含一个 int 类型的参数，那么对于类 AClass 的对象 anObject，就可以用 anObject(42) 的方式对重载的函数进行调用。这里调用对象是 anObject，参数为 42。函数的返回值可以是 void 或者任何其他类型。

与其他运算符重载不同的是，函数调用运算符在重载时可以包含任意个数的参数，因此可以定义多个版本的函数调用运算符重载函数。

### 陷阱：重载 &&、|| 及逗号运算符

预定义运算符 && 和 || 在处理 bool 类型的数据时，使用了短路求值。然而，这两个运算符重载后，将采用完全求值方式。这与大多数编程者的预期相反，因此不可避免地会导致一些错误，最好的办法就是不要重载这两个运算符。

逗号运算符的重载也会产生类似的问题。对于预定义的逗号运算符，操作顺序是从左至右的。但重载之后，逗号运算符不再保证这样的操作顺序。因此，实际编程中也要尽量避免对逗号运算符进行重载。■



## 8.2 友元函数与自动类型转换

相信你的朋友。

禅宗谚语

友元函数是一种非成员函数，但它却具有和成员函数一样的访问权限。在讨论友元函数之前，我们先通过构造函数讨论一下自动类型转换，这有助于理解将运算符（或者其他任何函数）重载为友元函数的优点。

### 构造函数的自动类型转换

如果类的定义包含了合适的构造函数，那么系统会自动进行某些类型转换。例如，如果程序中包含了示例 8.1 或者示例 8.2 中所示的 Money 类，那么就可以在程序中使用如下的代码：

```
Money baseAmount(100, 60), fullAmount;
fullAmount = baseAmount + 25;
fullAmount.output();
```

该代码片段的输出将为：

```
$125.60
```

上面的代码看起来很简单，而且用起来很自然。但有一点需要注意，数值 25（在表达式 `baseAmount+25` 中）并不是一个合适的类型。在示例 8.1 中，运算符“+”的重载仅仅限于两个 Money 对象之间，并没有对一个 Money 对象和一个 int 类型之间的加号进行重载。然而该代码片段可以正常运行，没有任何问题。这是因为 Money 类的定义中包含了一个带有一个 int 类型参数的构造函数，当系统遇到如下的表达式时：

```
baseAmount + 25
```

系统首先检查加号运算符是否被重载为一个 Money 类类型和一个 int 类型值的相加。由于没有这样的重载，紧接着系统会检查该类是否包含带有一个 int 类型参数的构造函数。如果存在这样的构造函数，系统就使用该构造函数将该 int 类型的值转换为一个 Money 类的对象。对于本例，构造函数将 25 转换为一个 Money 类的对象，该对象的成员变量 `dollars` 等于 25 而成员变量 `cents` 等于 0。也就是说，构造函数将 25 转换为一个代表 \$25.00 的 Money 类的对象（该构造函数的定义见示例 8.1）。

注意，除非类的定义包含合适的构造函数，否则这种自动类型转换是不会进行的。如果类 Money 没有包含带有一个 int 类型参数（或者其他类型的参数，如 long 或者 double）的构造函数，那么下面的表达式在编译时会出错：

```
baseAmount + 25
```

这种由构造函数进行的自动类型转换十分常见，常常与重载后的算术运算符，如“+”和“-”一起使用。同时，这种自动类型转换也同样适用于普通函数、成员函数或者其他运算符重载函数。



### 陷阱：成员运算符和自动类型转换

将二元运算符作为成员函数进行重载时，二元运算符的两个参数（即操作数）不再是对等的了。其中一个参数称为调用对象，另外一个则称为调用时真正的参数。这导致的不仅仅是形式上的不美观，而且实际使用中具有一定的缺点。这时，所有自动类型转换仅仅对第二个参数起作用。例如，前面小节提到的，下列语句是合法的：

```
Money baseAmount(100, 60), fullAmount;  
fullAmount = baseAmount + 25;
```

这是因为 Money 类包含带有一个 int 类型参数的构造函数，因此 25 作为一个 int 类型的值可以被构造函数自动转换为 Money 类的对象。

然而，如果将加号运算符作为成员函数进行重载（如示例 8.2 所示），那么加号两边的参数就不能互换了。例如，下面的语句是非法的：

```
fullAmount = 25 + baseAmount;
```

这是因为 25 不能作为一个调用对象。Money 类的构造函数可以对参数进行自动的类型转换，但不对调用对象进行类型转换。

而如果将加号运算符以非成员函数的方式进行重载（如示例 8.1 所示），那么下面的语句将是合法的：

```
fullAmount = 25 + baseAmount;
```

这点是将运算符重载函数定义为非成员函数的最大优点。

将运算符重载函数以非成员函数的方式进行声明和定义，可以使所有的参数都能够进行自动类型转换。而作为成员函数进行声明和定义则可以避免使用取值和赋值函数，从而提高函数的效率。C++ 中存在一种方法，同时具备这二者的优点，这就是下面要介绍的主题：将运算符以友元函数的方式进行重载。■

## 友元函数

如果类的定义包含了一整套的取值和赋值函数，那么就可以使用这些函数来定义非成员函数的运算符重载（如示例 8.1 所示）。然而，尽管这样可以访问类的私有成员变量，但并不是效率最高的方式。回顾一下示例 8.1 中加号运算符重载定义的代码，可以看到，为了访问四个私有成员变量，程序调用了两次 getCents 函数和两次 getDollars 函数。这不仅仅降低了程序的运行效率，而且使程序的代码难于理解。另一种方式是将加号运算符作为成员函数进行重载，这样虽然解决了前面提到的效率问题，但代价是丧失了第一个操作数的自动类型转换。将运算符重载为友元函数，不但能直接访问类的私有成员变量解决效率的问题，而且还能使所有操作数都能够进行自动类型转换。

### 友元函数

友元函数不是类的成员函数，但却可以像类的成员函数一样直接访问类的私有成员（包括私有成员变量和私有成员函数）。为了使一个函数成为类的友元函数，只需在类的定义中用 friend 关键字进行指明即可。例如，示例 8.3 重写了类 Money 的定义，这次将运算符作为类 Money 的友元函数进行了重载。将一个运算符或者函数作为类的

友元函数很简单，只需在类的定义中列出这些运算符或者函数声明，并在每一个声明之前添加关键字 `friend`。

### 示例 8.3 作为友元函数的运算符重载

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  // 表示美元数的 Money 类。
6  class Money
7  {
8  public:
9      Money();
10     Money(double amount);
11     Money(int dollars, int cents);
12     Money(int dollars);
13     double getAmount() const;
14     int getDollars() const;
15     int getCents() const;
16     void input(); // 同时读入美元符号和数字。
17     void output() const;
18     friend const Money operator +(const Money& amount1,
19                                   const Money& amount2);
20     friend const Money operator -(const Money& amount1,
21                                   const Money& amount2);
22     friend bool operator ==(const Money& amount1, const Money& amount2);
23     friend const Money operator -(const Money& amount);
24 private:
25     int dollars; // 一个负的美元数要求：美元部分为负，
26     int cents; // 且美分部分也为负，例如负的 $4.50 表示为 -4 和 -50。
27     int dollarsPart(double amount) const;
28     int centsPart(double amount) const;
29     int round(double number) const;
30 };

31 int main()
32 {
33     < 如果主函数的定义与示例 8.1 相同，那么运行结果也与示例 8.1 完全相同。 >
34 }

35 const Money operator +(const Money& amount1, const Money& amount2)
36 {
37     int allCents1 = amount1.cents + amount1.dollars*100;
38     int allCents2 = amount2.cents + amount2.dollars*100; 注意，友元函数可以直接访问类的成员变量。
39     int sumAllCents = allCents1 + allCents2;
40     int absAllCents = abs(sumAllCents); // 美元数可以为负值。
41     int finalDollars = absAllCents/100;
42     int finalCents = absAllCents%100;

43     if (sumAllCents < 0)
44     {

```

```

44         finalDollars = -finalDollars;
45         finalCents = -finalCents;
46     }

47     return Money(finalDollars, finalCents);
48 }

49 const Money operator -(const Money& amount1, const Money& amount2)
50     <完整的定义作为自测练习题 7。>

51 bool operator ==(const Money& amount1, const Money& amount2)
52 {
53     return ((amount1.dollars == amount2.dollars)
54         && (amount1.cents == amount2.cents));
55 }

56 const Money operator -(const Money& amount)
57 {
58     return Money(-amount.dollars, -amount.cents);
59 }

60 <其他函数的定义与示例 8.1 完全相同。>

```

友元运算符或友元函数必须在类的定义中列出其声明，就像在类中列出成员函数的声明一样，不同的地方在于，友元函数的前面有关键字 `friend`。但是，友元函数不是成员函数，它只是对类的数据成员具有特殊访问权限的普通函数。友元函数的定义和普通函数的定义一样。特别需要指出的是，在示例 8.3 的友元运算符重载函数的定义中，函数定义的第一行并没有 `Money::`。另外，函数的定义中也不需要关键字 `friend`（仅仅出现在函数声明中）。示例 8.3 中友元函数的调用方式和示例 8.1 中的非友元函数、非成员运算符重载函数的调用方式相同，因此所有参数都可以根据构造函数进行自动类型转换。

友元常用在运算符重载函数中，但任意类型的函数都可以声明为友元函数。

此外，一个函数（或者重载的运算符）可以作为一个或者多个类的友元函数。为此，只需相应在每个类的定义中添加友元函数的声明即可。

许多专家认为友元函数（和运算符）没有严格遵循面向对象编程思想。因此根据面向对象编程思想，所有的运算符重载函数以及其他函数都应该作为成员函数。但友元函数的使用为编程提供了便利，它使所有参数都可以进行自动类型转换。同时，由于友元函数声明在类的定义中，因此提供了比非成员、非友元函数更好的封装性。本书介绍了三种类型的运算符重载方式：作为普通函数进行重载、作为成员函数进行重载，以及作为友元函数进行重载。在实际使用中，可以根据具体的情况选择。

## 友元函数

一个类的友元函数是一种特殊的普通函数，它拥有访问该类私有成员的权限。为了使一个函数成为某个类的友元函数，只需在该类的定义中添加该函数的声明，并且声明前添加关键字 `friend`。友元函数可以放在类定义的私有部分，也可以放在类定义的共有部分，但无论放在哪里，友元函数都是公有的。因此为了便于理解，最好是将友元函数放在类定义的公有部分。

### 包含友元函数类的定义

```
class Class_Name
{
public:
    friend Declaration_for_Friend_Function_1
    friend Declaration_for_Friend_Function_2
    .
    .
    .
    Member_Function_Declarations
private:
    Private_Member_Declarations
};
```

不需要将友元函数的声明放在最开始。友元函数可以与其他成员函数混合声明。

### 示例

```
class FuelTank
{
public:
    friend void fillLowest(FuelTank& t1, FuelTank& t2);
    // 用最低等级的燃料来填充油箱，如果相等就使用 t1。

    FuelTank(double theCapacity, double theLevel);
    FuelTank();
    void input();
    void output() const;
private:
    double capacity; // 以升为单位。
    double level;
};
```

友元函数不是类的成员函数。友元函数的定义和调用方式与普通函数完全相同。调用友元函数时不需要使用点运算符，同时定义友元函数时也不需要类型限制符。

## 友元类

### 友元类

函数可以作为一个类的友元函数；同样，一个类也可以作为另外一个类的友元类。如果类 `F` 是类 `C` 的友元，那么类 `F` 的所有成员函数都是类 `C` 的友元函数。要使一个类成为另一个类的友元类，就需要在另一类的定义中声明该类为友元类。

一个类成为另一个类的友元后，这两个类在定义中常常会互相引用，这就要求第一个类要包含第二个类的提前声明。接下来的介绍中给出了一个大体的例子。需要注

意的是，类的提前声明由类定义的第一行和一个分号组成。

如果要将类 F 声明为类 C 的友元，通常的代码框架如下：

```
class F; // 提前声明

class C
{
public:
    ...
    friend class F;
    ...
};

class F {
    ...
}
```

使用友元类的完整例子可参考第 17 章的内容，在这之前我们不会使用到友元类。



#### 陷阱：不支持友元的编译器

有些编译器不支持友元函数。更糟糕的是，一些编译器在某些情况下友元函数可以正常工作，而在其他情况下又不能正常工作。有些编译器并不总是支持对私有成员的访问。一般来讲，这样的编译器在后续版本中会添加对友元函数的支持。在实际项目中，如果碰到此类编译器，要么采用取值函数和赋值函数将函数定义为非成员函数和重载运算符，要么就直接将运算符重载为成员函数。■

#### 运算符重载的规则

- 运算符重载时，该运算符包含的参数必须至少包含一个类类型。
- 大多数运算符都有三种重载方式：以成员函数方式重载、作为类的友元函数重载及以普通函数重载。
- 下面几个运算符只能以成员函数的方式进行重载：`=`、`[]`、`->`、和 `()`。
- 不可以创建新的运算符，只能对已有的运算符进行重载，如 `+`、`-`、`*`、`/`、`%` 等。
- 重载运算符时，不可以改变运算符包含的参数个数。例如，不可以将二元运算符 `%` 重载为一元运算符，当然也不能将一元运算符 `++` 重载为二元运算符。
- 运算符重载不会改变运算符的优先级。重载后的运算符与重载前的运算符具有相同的优先级。例如，对于表达式 `x * y + z` 来讲，不论 `x`、`y`、`z` 是基本数据类型还是类类型，以及运算符是否重载过，其运算顺序始终是 `(x * y) + z`。
- 下面几个运算符是不能被重载的：点运算符 (`.`)、作用域限制符 (`::`)、`sizeof`、三元运算符 (`? :`) 及本书未曾讨论的运算符 (`.*`)。
- 重载的运算符不能包含默认参数。

### 自测练习题

6. 一个类的友元函数与成员函数有什么区别？
7. 完成示例 8.3 中友元减号运算符重载函数的定义。
8. 如果要为示例 8.3 中的类 Money 重载运算符“<”，那么应该在类 Money 的定义中添加什么代码？

## 8.3 引用和其他运算符重载

请不要弄错指向月亮的手指。

禅说

本节介绍一些特殊，但很重要的运算符重载，包括赋值运算符、“<<”、“>>”、“[]”、“++”和“--”等运算符。为了正确地对这些运算符进行重载，有时需返回引用类型，下面首先介绍引用相关的知识。

### 引用

**引用** 引用是一个存储地址的名字。<sup>1</sup> 可以用如下的方式定义一个独立的引用：

```
int robert;
int& bob = robert;
```

上面的语句定义了一个名字为 bob 的引用，且指向变量 robert。这使 bob 成为变量 robert 的别名。对 bob 所做的任何修改都会导致变量 robert 的修改。这样看来，独立的引用只会使程序变得含糊不清，使我们陷入麻烦之中。实际上，在大多数情况下，独立的引用确实是一个麻烦，尽管有的时候还是有用的。本书不会讨论独立引用，而且也不会用到。

如大家所想的，“引用”这一机制用在引用传递调用参数中，因此对本章来说，引用并不是一个全新的概念。

这里讨论“引用”，是因为返回引用会使某些运算符的重载使用起来更加自然。返回一个引用可以看作是返回一个变量，更确切地说，是返回一个变量的别名。引用的语法很简单，只需在返回类型之后添加“&”即可，如下所示：

```
double& sampleFunction(double& variable);
```

由于类型 double& 和类型 double 是不同的，因此必须在函数声明和函数定义中都使用“&”符号。返回的表达式必须是能够被引用的，比如某种类型的变量。但不能是一个如 x+5 这样的表达式；也不能返回一个局部变量的引用，虽然大多数编译器允许这样做，但其结果是不可预料的（因为局部变量在函数返回后会被立即销毁）。一个

<sup>1</sup> 如果之前了解过指针的概念，那么会发现引用和指针有点儿类似。引用本质上就是一个 const 指针，但是两者还是有一定的差别，因此不能完全互换。

简单的返回引用的函数定义如下：

```
double& sampleFunction(double& variable)
{
    return variable;
}
```

虽然上面的函数没有什么实际用途，但演示了引用的一般用法。例如，下面的代码首先输出 99，再输出 42：

```
double m = 99;
cout << sampleFunction(m) << endl;
sampleFunction(m) = 42;
cout << m << endl;
```

在重载某些运算符时，只能返回引用类型。

### 左值和右值

左值是指能够出现在赋值运算符左边的值或者表达式；右值是指能够出现在赋值运算符右边的值或者表达式。

如果希望函数的返回值可以作为左值，那么必须返回一个引用类型。

### 提示：返回类的成员变量

当返回一个类的成员变量时，在大多数情况下，都应该以 `const` 形式返回。假设我们不这么做，看看会有什么问题，查看如下代码：

```
class Employee
{
public:
    Money& getSalary() { return salary; }
    ...
private:
    Money salary;
    ...
};
```

在本例中，`salary` 是一个私有成员变量，对它的修改只能通过类 `Employee` 的赋值函数来进行。`getSalary` 函数返回类型为 `Money` 的成员变量 `salary`。如果不以引用的方式返回 `salary` 对象，那么该函数会首先创建一个 `salary` 对象的临时拷贝，然后返回该临时拷贝。为了提高函数的效率，避免对象的拷贝，一般都返回一个对象的引用，如本例所示。但这么处理存在一个问题：我们可以绕过 `Money` 类对私有成员变量 `salary` 的限制，对它的值进行修改：

```
Employee joe;
(joe.getSalary()).input();
```

通过以上方式，名为 `joe` 的员工可以修改其薪水。

如果 `getSalary` 函数以 `const` 形式返回类的成员变量，如下所示：



```

class Employee
{
public:
    const Money& getSalary( ) { return salary; }
    . . .

private:
    Money salary;
    . . .
};

```

这种情况下，如下的语句将会产生编译时错误：

```
joe.getSalary( ).input( );
```

(函数 getSalary 的声明，理想情况下应该为：

```
const Money& getSalary( ) const { return salary; }
```

但这里，我们不想引入另外一类 const，从而让读者困惑。)

一般来讲，当成员函数返回的是成员变量时，且返回的成员变量为类类型，为了避免外部代码对私有成员变量的访问，此时不建议返回一个引用类型。如果考虑到效率因素必须返回一个引用类型的话，则最好返回一个 const 类型的引用。■

## 重载 “>>” 和 “<<”

运算符 “>>” 和 “<<” 也可以被重载，这样就可以用重载之后的运算符来输入和输出自定义类的对象。重载的方式与其他运算符没有什么大的区别，不过有一些细微的差别。

<< 是一个  
运算符

与 cout 一起使用的插入运算符 << 与 “+” “-” 运算符类似，也是一个二元运算符。例如，考虑如下的语句：

```
cout << "Hello out there.\n";
```

流

上面的语句，运算符是 “<<”，第一个操作数是预定义的对象 cout（包含在 iostream 库中），第二个操作数是字符串值 “Hello out there.\n”。预定义的对象 cout 是 ostream 类的对象，因此重载运算符 “<<” 时，接受 cout 对象的参数也必须是 ostream 类型的。可以改变运算符 “<<” 的两个参数。当学习到第 12 章的文件输入/输出时，我们将会看到如何创建一个 ostream 类的对象，以便将输出结果存储到文件中（这些文件输入/输出对象和 cin 及 cout 对象都被称为流，这也是库的名字为 iostream 的原因）。这里的重载对文件输出也同样有效，而不需要修改运算符 “<<” 的重载定义。

重载 <<

示例 8.1 到 8.3 中类 Money 的定义，我们使用了一个名为 output 的成员函数来输出 Money 类的值。这么做是没有问题的，但如果能使用一个简单的插入运算符 “<<” 来输出 Money 类的值将会更好。如下代码所示：

```

Money amount(100);
cout << "I have " << amount << " in my purse.\n";

```

而不是采用如下的输出方式：

```
Money amount(100);
cout << "I have ";
amount.output( );
cout << " in my purse.\n";
```

重载运算符“<<”所面临的一个问题就是要返回什么值。比如下面的表达式：

```
cout << amount
```

上面表达式中的两个操作数分别为 `cout` 和 `amount`，表达式的执行结果是将 `amount` 的值输出到屏幕上。但由于运算符“<<”是和“+”、“-”类似的运算符，因此上面的表达式也有返回值。诸如 `n1+n2` 这样的表达式，其返回值很明确，但 `cout << amount` 将返回什么值呢？为回答这个问题，首先看一个更复杂的带有“<<”运算符的表达式。

#### << 运算符链

考虑如下使用了连续“<<”运算符的表达式：

```
cout << "I have " << amount << " in my purse.\n";
```

由于运算符“<<”与加号运算符“+”类似，因此上面的表达式其实和下面的表达式等价：

```
((cout << "I have ") << amount) << " in my purse.\n";
```

为了使以上表达式有意义，运算符“<<”应该返回什么值呢？首先看看子表达式：

```
(cout << "I have ")
```

要使整个表达式有意义，那么上面的子表达式就应该返回 `cout`，以便随后的运算可以按照下面继续进行：

```
((cout << amount) << " in my purse.\n");
```

类似地，表达式 `(cout << amount)` 也应该返回 `cout`，以便表达式继续求值：

```
cout << " in my purse.\n";
```

#### << 运算符 返回一个流

整个过程如示例 8.4 所示。运算符“<<”应该返回它的第一个参数，类型为 `ostream` 类。

因此，运算符“<<”的重载声明如下：

```
class Money
{
public:
    . . .
    friend ostream& operator <<(ostream& outs,
                               const Money& amount);
```

#### 示例 8.4 “<<”作为运算符

```
cout << "I have " << amount << " in my purse.\n";
```

上面的表达式等价于：

```
((cout << "I have ") << amount) << " in my purse.\n";
```

该表达式的求值过程如下：

首先对表达式 `(cout << "I have")` 求值, 返回 `cout` :

```
((cout << "I have ") << amount) << "in my purse.\n";
```

字符串 "I have" 为运行的输出。

```
(cout << amount) << " in my purse.\n";
```

紧接着对 `(cout << amount)` 求值, 返回 `cout` :

```
cout << amount) << " in my purse.\n";
```

变量 `amount` 的值为运行的输出。

```
cout << " in my purse.\n";
```

最后对表达式 `cout << " in my purse.\n"` 求值, 返回 `cout` :

```
cout << " in my purse.\n";
```

字符串 "in my purse.\n" 为运行的输出。

```
cout;
```

由于表达式中没有 "<<" 运算符, 因此整个表达式求值结束。

一旦对插入运算符 "<<" 重载后, 就不再需要成员函数 `output` 了, 可以将其从类的定义中删除。插入运算符 "<<" 的重载定义与成员函数 `output` 的定义非常类似。下面给出该运算符重载定义的大概形式 :

```
ostream& operator <<(ostream& outputStream, const Money& amount)
{
    /* 这部分代码与示例 8.1 中成员函数 output 的函数体非常类似。不同之处在于美元
    用 amount.dollars 表示, 而美分用 amount.cents 表示。*/
    return outputStream;
}
```

<< 和 >> 返回一个引用

注意, 函数的返回值类型为引用类型。

提取运算符 ">>" 的重载方式和插入运算符 "<<" 非常类似, 然而重载提取运算符时, 第二个参数 (操作数) 是用来接受输入值的对象, 因此第二个参数通常是一个普通的引用传递调用参数, 而不应该是一个常量参数。提取运算符重载定义的大概形式如下 :

```
istream& operator >>(istream& inputStream, Money& amount)
{
    /* 这部分代码与示例 8.1 中成员函数 input 的函数体非常类似。不同之处在于美元用
    amount.dollars 表示, 而美分用 amount.cents 表示。*/
    return inputStream;
}
```

提取运算符 ">>" 和插入运算符 "<<" 重载的完整定义如示例 8.5 所示, 同时也对类 `Money` 进行了改写, 改写后的类可以使用运算符 ">>" 和 "<<" 来输入和输出 `Money` 类的值。

注意, 不能将运算符 "<<" 和 ">>" 重载为成员函数。因为这两个运算符要正常工作, 那么它的第一个参数 (操作数) 必须是 `cout` 或者 `cin` (或者其他文件 I/O 流对象)。如果将这两个运算符重载成员函数, 那么第一个参数 (操作数) 只能是调用对象, 这会导致重载后的运算符不能按照期望的方式工作。

### 示例 8.5 重载 “<<” 和 “>>”

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  // 表示美元数的 Money 类。
6  class Money
7  {
8  public:
9      Money();
10     Money(double amount);
11     Money(int theDollars, int theCents);
12     Money(int theDollars);
13     double getAmount() const;
14     int getDollars() const;
15     int getCents() const;
16     friend const Money operator + (const Money& amount1,
17                                   const Money& amount2);
18     friend const Money operator - (const Money& amount1,
19                                   const Money& amount2);
20     friend bool operator == (const Money& amount1,
21                              const Money& amount2);
22     friend const Money operator - (const Money& amount);
23     friend ostream& operator << (ostream& outputStream,
24                                   const Money& amount);
25     friend istream& operator >> (istream& inputStream, Money& amount);
26 private:
27     int dollars, cents;
28     // 一个负的美元数要求：美元部分为负，
29     // 且美分部分也为负，例如负的 $4.50 表示为 -4 和 -50。
30
31     int dollarsPart(double amount) const;
32     int centsPart(double amount) const;
33     int round(double number) const;
34 };

35 int main()
36 {
37     Money yourAmount, myAmount(10, 9);
38     cout << "Enter an amount of money: ";
39     cin >> yourAmount;
40     cout << "Your amount is " << yourAmount << endl;
41     cout << "My amount is " << myAmount << endl;
42
43     if (yourAmount == myAmount)
44         cout << "We have the same amounts.\n";
45     else
46         cout << "One of us is richer.\n";

47     Money ourAmount = yourAmount + myAmount;
48     cout << yourAmount << " + " << myAmount
49           << " equals " << ourAmount << endl;

```

由于运算符 “<<” 返回一个引用，因此可以连续地使用 “<<” 运算符。运算符 “>>” 的用法类似。

```

44     Money diffAmount = yourAmount - myAmount;
45     cout << yourAmount << " - " << myAmount
46         << " equals " << diffAmount << endl;

47     return 0;
48 }

    <其他成员函数的定义见示例8.1, 其他运算符重载定义见示例8.3.>

49 ostream& operator <<(ostream& outputStream, const Money& amount)
50 {
51     int absDollars = abs(amount.dollars);
52     int absCents = abs(amount.cents);
53     if (amount.dollars < 0 || amount.cents < 0)
54         // 解决dollars==0或者cents == 0的情况。
55         outputStream << "$-";
56     else
57         outputStream << '$';
58     outputStream << absDollars;

59     if (absCents >= 10)
60         outputStream << '.' << absCents;
61     else
62         outputStream << '.' << '0' << absCents;

63     return outputStream;
64 }
65
66 // 使用了 cstdlib 和 iostream 库。
67 istream& operator >>(istream& inputStream, Money& amount)
68 {
69     char dollarSign;
70     inputStream >> dollarSign;
71     if (dollarSign != '$')
72     {
73         cout << "No dollar sign in Money input.\n";
74         exit(1);
75     }

76     double amountAsDouble;
77     inputStream >> amountAsDouble;
78     amount.dollars = amount.dollarsPart(amountAsDouble);
79     amount.cents = amount.centsPart(amountAsDouble);

80     return inputStream;
81 }

```

main 函数中, 用 cout 替换 outputStream。

第7章自测练习题3给出了另外一种输出的算法。

返回一个引用。

main 函数中, 用 cin 替换 inputStream。

由于不是 Money 类的成员运算符, 因此需要一个调用对象。

返回一个引用。

### 示例运行结果

```

Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36

```

### 自测练习题

9. 在示例 8.5 中, 运算符 “<<” 的重载包含如下的代码:

```
ostream & operator << ("<-");
```

这是否是一个循环? 我们是否在用 “<<” 来定义 “<<”?

10. 为什么不能将运算符 “<<” 和 “>>” 重载为成员函数?

11. 以下是一个名为 Percent 类的定义。该类的对象用来表示百分数, 如 10% 或 99% 等。请给出运算符 “<<” 和 “>>” 的重载定义, 以便输入/输出类 Percent 的对象。这里假设输入始终为整数加字符 '%', 例如 25%。所有的百分数都为合法的数字, 并且存储在名为 value 的整型成员变量中。这里不需要给出其他运算符的重载定义以及构造函数, 只需给出运算符 “<<” 和 “>>” 的重载定义。

```
#include <iostream>
using namespace std;
class Percent
{
public:
    friend bool operator ==(const Percent& first,
                           const Percent& second);
    friend bool operator <(const Percent& first,
                          const Percent& second);

    Percent();
    friend istream& operator >>(istream& inputStream,
                               Percent& aPercent);
    friend ostream& operator <<(ostream& outputStream,
                               const Percent& aPercent);

    // 通常还会有其他成员和友元。
private:
    int value;
};
```

### 提示: 应使用什么样的返回值类型

一个函数有四种不同的方式返回一个类型为 T 的返回值:

- 返回一个一般的值, 对应的函数声明为 T f( )。
- 返回一个常量值, 对应的函数声明为 const T f( )。
- 返回一个引用, 对应的函数声明为 T& f( )。
- 返回一个常量引用, 对应的函数声明为 const T& f( )。

何时采用某种返回值类型并没有一个明确的规定, 因此实际中的使用方式因人而异。即使作者或者编程人员有一个明确的规则, 但也有例外的情况出现。当然, 这还是一定的规律可循的。

如果返回值是简单类型, 如 int 或者 char, 那么返回值时使用 const 修饰符是没有任何实际意义的。因此, 在返回简单类型时, 编程人员往往不使用 const 修饰符。如果希望返回的值可以作为左值, 即可以放在赋值运算符的左边, 那么就必須返回引用类型; 反之, 就返回一般的值。类类型不是简单数据类型。接下来, 我们讨论返回

值为类类型的情况。

是否返回引用类型取决于返回的对象是否要作为左值。如果希望返回的对象作为左值，也就是出现在赋值运算符的左边，那么就on该返回引用类型，即在返回值类型后面使用引用符号“&”。

返回一个局部变量（或者其他生命期较短的变量）的引用，不论是否使用了 `const` 修饰词，都会产生问题，因此应该避免返回一个局部变量的引用。

对于类类型来讲，返回值类型 `const T` 和 `const T&` 非常类似。二者都意味着不能通过调用某些函数来修改返回的对象，如

```
f().mutator();
```

但其返回值可以通过赋值运算符复制到其他变量中，然后通过其他变量来调用相关函数以修改对象。如果不能决定到底使用 `const T&` 还是 `const T`，那么就使用 `const T`（没有“&”符号）。尽管使用 `const T&` 可能会比 `const T` 要高效一点<sup>2</sup>，但二者的差别是很小的，因此许多编程人员常常使用 `const T` 作为返回值类型，而不是 `const T&`。就像前面提到的，某些时候使用 `const T&` 可能会出现问題。

下面的总结对于采用哪种返回值类型可能会有帮助。这里假设 `T` 为一类类型，拷贝构造函数的介绍将在第10章中讨论，这里仅仅作为一个参考。如果还没有学习第10章，可以简单地忽略所有涉及拷贝构造函数的描述。

如果一个类的公有成员函数返回一个类的私有成员变量，那么返回值类型一定要使用 `const` 修饰符。这在本章名为“返回类的成员变量”的提示中已经解释过了（一个例外是在返回 `string` 类型的时候，编程人员通常以普通值的方式返回 `string`，而不使用 `const` 修饰符。这是因为尽管 `string` 类型是一个类类型，但常常将 `string` 类型看作是一个简单的数据类型，如 `int` 或者 `char` 等）。

下面是关于使用何种返回值类型的总结，其中 `T` 为一个类类型：

- 返回一个一般的值，对应的函数声明为 `T f()`；

这种返回值不能作为左值，此外返回值可以直接调用相关的成员函数来修改返回值，如 `f().mutator()`。返回值会调用拷贝构造函数。

- 返回一个常量值，对应的函数声明为 `const T f()`；

这种返回值类型与前一种情况几乎完全相同，唯一不同的地方在于返回值不能通过调用成员函数来修改返回值，即不能调用 `f().mutator()`。

- 返回一个引用，对应的函数声明为 `T& f()`；

这种返回值类型可以作为左值，而且可以通过成员函数的调用来修改返回值，即可以使用 `f().mutator()`。返回值不会调用拷贝构造函数。

- 返回常量引用，对应的函数声明为 `const T& f()`；

这种返回值类型不能作为左值，且不能通过成员函数的调用来修改返回的对象，即不能使用 `f().mutator()`。返回值不会调用拷贝构造函数。■

<sup>2</sup> 这是因为返回值为 `const T&` 时不会调用拷贝构造函数，而返回值为 `const T` 则会调用拷贝构造函数（参见第10章处的相关介绍）。

### 重载 “<<” 和 “>>”

输入和输出运算符 “<<” 和 “>>” 和其他运算符一样，也可以被重载。如果希望重载之后的运算符能够与预定义的 cin、cout 正常工作，那么重载后的返回值类型应该分别为 istream 和 ostream 类型，同时必须返回引用类型。

#### 声明

```
class Class_Name
{
    . . .

public:
    . . .

    friend istream& operator >>(istream& Parameter_1,
                               Class_Name& Parameter_2);
    friend ostream& operator <<(ostream& Parameter_3,
                               const Class_Name& Parameter_4);
    . . .
}
```

输入和输出运算符不一定要重载为类的友元，但一定不能重载为类的成员函数。

#### 定义

```
istream& operator >>(istream& Parameter_1,
                    Class_Name& Parameter_2)
{
    . . .
}
ostream& operator <<(ostream& Parameter_3,
                    const Class_Name& Parameter_4)
{
    . . .
}
```

如果类中包含足够多的取值和赋值函数，那么可以将输入和输出运算符 “<<” 和 “>>” 重载为非友元形式。然而，更常见且高效的方法是将其重载为友元。

### 赋值运算符

如果要重载赋值运算符 “=”，那么必须将其作为成员函数进行重载。如果类的定义中没有提供赋值运算符重载函数，那么类会提供一个默认的赋值运算符。这个默认的赋值运算符会简单地将该类一个对象的成员变量拷贝到该类另一个对象的成员变量中。对于简单的类来说，这样的操作正好满足要求。但涉及指针时，默认的赋值运算符就不能满足要求了，我们将在讨论指针的时候来介绍赋值运算符的重载方法。

### 重载自增和自减运算符

前置形式和  
后置形式

自增运算符 “++” 和自减运算符 “--” 分别包含两个版本。即运算符前置形式（如 ++x）和运算符后置形式（如 x++），这两者进行的操作是不一样的。因此，当我们在对这两个运算符进行重载时，就必须区分前置和后置形式。C++ 根据参数的个数来区



分前置形式和后置形式。如果按照通常的方法来重载 ++ 运算符（即作为成员函数重载时不带参数，或者作为非成员函数重载时带有一个参数），那么重载的是前置版本。要对后置形式进行重载，即  $x++$  或者  $x--$ ，就必须为重载函数再增加一个 `int` 类型的参数。该参数仅仅用来告诉编译器这是一个运算符后置形式，在实际调用时不需要给出实际的参数值。

例如，示例 8.6 包含一个类的定义，该类包含一对整数。自增运算符 “++” 同时重载了前置版本和后置版本。重载后的自增运算符所执行的操作和预定义的针对 `int` 数据类型的操作类似，都很直观。让重载之后的运算符与本来的功能保持一致，这是最符合常理的。当然，我们也可以使重载之后的 “++” 运算符返回任意类型的值，也可以执行任意类型的操作。

运算符后置版本的重载定义中忽略了制作为标记用的整型参数，如示例 8.6 所示。当编译器碰到 `a++` 时，会将其看作 `IntPair::operator++(int)` 的调用，且 `a` 作为调用对象。

当自增运算符或自减运算符用于简单数据类型时，如 `int` 类型或者 `char` 类型，其前置形式返回引用，而后置形式返回一般的值。如果希望重载之后的自增和自减运算符完全模拟其对简单数据类型的操作，那么应该在重载前置版本时返回引用类型，而在重载后置版本时返回一般值。但实际使用中发现，返回引用类型的自增和自减运算符往往会带来一系列的问题，因此不论是哪个版本，我们通常都返回一般的值。

## 自测练习

12. 下面的语句是否合法？并解释原因（类 `IntPair` 的定义如示例 8.6 所示）。

```
IntPair a(1,2);
(a++)++;
```

### 示例 8.6 重载 ++

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;

4 class IntPair
5 {
6 public:
7     IntPair(int firstValue, int secondValue);
8     IntPair operator++(); // 前置形式
9     IntPair operator++(int); // 后置形式
10    void setFirst(int newValue);
11    void setSecond(int newValue);
12    int getFirst() const;
13    int getSecond() const;
14 private:
15     int first;
16     int second;
17 };
```

函数声明或者运算符重载声明中无须给出参数的名字，因为 C++ 不会用到这些声明中的参数。

```

18 int main()
19 {
20     IntPair a(1,2);
21     cout << "Postfix a++: Start value of object a: ";
22     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;
23     IntPair b = a++;
24     cout << "Value returned: ";
25     cout << b.getFirst( ) << " " << b.getSecond( ) << endl;
26     cout << "Changed object: ";
27     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;

28     a = IntPair(1, 2);
29     cout << "Prefix ++a: Start value of object a: ";
30     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;
31     IntPair c = ++a;
32     cout << "Value returned: ";
33     cout << c.getFirst( ) << " " << c.getSecond( ) << endl;
34     cout << "Changed object: ";
35     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;
36     return 0;
37 }
38

39 IntPair::IntPair(int firstValue, int secondValue)
40     : first(firstValue), second(secondValue)
41 { /* 函数体有意留空 */ }
42 IntPair IntPair::operator++(int ignoreMe) // 后置版本
43 {
44     int temp1 = first;
45     int temp2 = second;
46     first++;
47     second++;
48     return IntPair(temp1, temp2);
49 }

50 IntPair IntPair::operator++( ) // 前置版本
51 {
52     first++;
53     second++;
54     return IntPair(first, second);
55 }

56 void IntPair::setFirst(int newValue)
57 {
58     first = newValue;
59 }

60 void IntPair::setSecond(int newValue)
61 {
62     second = newValue;
63 }

64 int IntPair::getFirst( ) const
65 {

```

```

66     return first;
67 }

68 int IntPair::getSecond() const
69 {
70     return second;
71 }

```

#### 示例运行结果

```

Postfix ++: Start value of object a: 1 2
Value returned: 1 2
Changed object: 2 3
Prefix ++a: Start value of object a: 1 2
Value returned: 2 3
Changed object: 2 3

```

---

### 重载数组运算符[]

可以对方括号（数组运算符）进行重载，这样就可以和类的对象一起使用。如果希望重载之后的数组运算符用在赋值运算符的左侧，那么重载函数的定义必须返回引用类型。数组运算符只能作为成员函数进行重载。

回顾一下数组运算符有助于对其进行重载，因为它和以前见到的其他运算符不同。由于它只能作为成员函数进行重载，因此使用“[]”运算符的表达式中有一个必然为调用对象。例如，对于表达式 `a[2]`，`a` 是调用对象，而 `2` 则是运算符的参数。重载数组运算符“[]”时，索引参数必须为整数类型，即应该为 `enum`、`char`、`short`、`int`、`long` 或者 `unsigned` 之一。

例如，示例 8.7 定义了一个名为 `CharPair` 的类，它的对象与一个包含两个索引值 1 和 2 的字符数组类似，注意表达式 `a[1]` 和 `a[2]` 与数组的索引变量一样。仔细观察数组运算符“[]”的定义，你就可以发现运算符重载函数返回了一个引用类型，并且是一个成员变量的引用，而不是对整个对象的引用。这是因为对象的成员变量类似数组的索引变量。改变 `a[1]` 时，我们希望对应的成员变量 `first` 也随之改变。这样就提供了一种改变类私有成员变量的途径，例如示例 8.7 中的代码就是通过改变 `a[1]` 和 `a[2]` 来改变类的私有成员变量的。尽管成员变量 `first` 和 `second` 是私有成员变量，但这样的代码是合法的，这是因为没有直接使用成员变量 `first` 和 `second`，而是通过 `a[1]` 和 `a[2]` 来间接实现的。

#### 示例 8.7 重载 []

---

```

1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;

4 class CharPair
5 {
6 public:

```

```

7      CharPair( ){/* 函数体故意留空 */}
8      CharPair(char firstValue, char secondValue)
9          : first(firstValue), second(secondValue)
10         {/* 函数体故意留空 */}
11
12     char& operator[](int index);
13 private:
14     char first;
15     char second;
16 };

17 int main()
18 {
19     CharPair a;
20     a[1] = 'A';
21     a[2] = 'B';
22     cout << "a[1] and a[2] are:\n";
23     cout << a[1] << a[2] << endl;

24     cout << "Enter two letters (no spaces):\n";
25     cin >> a[1] >> a[2];
26     cout << "You entered:\n";
27     cout << a[1] << a[2] << endl;

28     return 0;
29 }
30
31 // 使用了 iostream 和 cstdlib 库。
32 char& CharPair::operator[](int index)
33 {
34     if (index == 1)
35         return first;
36     else if (index == 2)
37         return second;
38     else
39     {
40         cout << "Illegal index value.\n";
41         exit(1);
42     }
43 }

```

注意，这里返回的是对象的成员变量，而不是整个 CharPair 对象。因为成员变量与数组的索引变量类似。

### 示例运行结果

```

a[1] and a[2] are:
AB
Enter two letters (no spaces):
CD
You entered:
CD

```

### 基于左值和右值的重载

虽然本书不会这样使用，但其实也可以重载一个函数名（或者运算符），使之在作为左值和作为右值时有不同的动作（回顾一下，用作左值意味着可以出现在赋值运算符的左边）。例如，如果希望函数 `f`，在作为左值和右值时有不同的行为，可以采用如下的代码：

```
class SomeClass
{
public:
    int& f(); // 用作左值调用。
    const int& f() const; // 用作右值调用。
    ...
};
```

两个函数的参数列表不一定为空，但必须完全相同，否则就成为不同的重载形式。需要注意的是第二个函数 `f` 的声明包含两个 `const` 关键字。这两个 `const` 关键字都不能省略，同时，引用符号“&”也不能省略。

## 本章小结

- 运算符（如“+”和“=”等）可以被重载，重载之后的运算符可以用于自定义类的对象之间。
- 运算符可以看作是一种特殊的函数，拥有特殊的调用语法。
- 类的友元函数实际上是一个普通函数，只不过它可以像类的成员函数那样访问类的私有成员。
- 当运算符被重载为类的成员函数时，它的第一个操作数为调用对象。
- 如果类的定义已经包含一整套的取值和赋值函数，那么将函数定义为友元的唯一原因就是友元函数更加有效和简单。
- 引用是一种命名变量的方式，它实际上是变量的一个别名。
- 重载运算符“<<”和“>>”时，返回值的类型应该是流类型的引用，即在返回类型名字的后面添加“&”符号。

## 自测练习题答案

1. 二元运算符（如“+”、“\*”或者“/”等）和函数的区别在于二者的调用语法不同。在函数调用中，参数放在函数名后面的圆括号中。而对于二元运算符，参数则放在运算符的前后。而且运算符重载函数的声明和定义中必须使用关键字 `operator`。

## 2. 需添加如下的函数声明和定义：

```
bool operator< (const Money& amount1, const Money& amount2);

bool operator< (const Money& amount1, const Money& amount2)
{
    int dollars1 = amount1.getDollars();
    int dollars2 = amount2.getDollars();
    int cents1 = amount1.getCents();
    int cents2 = amount2.getCents();
    return ((dollars1 < dollars2) ||
            ((dollars1 == dollars2) && (cents1 < cents2)));
}
```

## 3. 重载运算符时，至少应有一个参数是类类型。这就防止了更改整数之间的加法操作。

## 4. 如果省略了 Money 类中加号运算符重载定义和声明中的第一个关键字 const，那么下面的语句是合法的：

```
(m1 + m2) = m3;
```

如果类的定义如示例 8.1 所示，即加号运算符返回 const 值，那么上面的语句是非合法的。

## 5. const Money

```
Money::operator- (const Money& secondOperand) const
{
    int allCents1 = cents1 + dollars*100;
    int allCents2 = secondOperand.cents
                    + secondOperand.dollars*100;
    int diffAllCents = allCents1 - allCents2;
    int absAllCents = abs(diffAllCents);
    int finalDollars = absAllCents/100;
    int finalCents = absAllCents%100;

    if (diffAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```

## 6. 友元函数和成员函数的相同点在于二者的函数定义中都可以使用类中的任何成员（公有成员或者私有成员）。但是，友元函数的定义和使用与普通函数类似，调用友元函数时无须使用点运算符，同时编写函数定义时也无须使用类型修饰符。而对于成员函数，调用时需要使用对象名称和点运算符，同时进行函数定义时必须使用由类名和作用域运算符“::”组成的类型修饰符。

## 7. // 使用 cstdlib 库。

```
const Money operator- (const Money& amount1,
                      const Money& amount2)
```

```

    {
        allCents1 = amount1.cents + amount1.dollars*100;
        allCents2 = amount2.cents + amount2.dollars*100;
        diffAllCents = allCents1 - allCents2;
        absAllCents = abs(diffAllCents);

        finalDollars = absAllCents/100;
        finalCents = absAllCents%100;
        if (diffAllCents < 0)
        {
            finalDollars = -finalDollars;
            finalCents = -finalCents;
        }

        return Money(finalDollars, finalCents);
    }
}

```

#### 8. 应添加如下的函数声明和定义：

```

// 重载友元操作符 <
bool operator < (const Money& amount1,
                const Money& amount2);

// 重载友元操作符 <=
bool operator <= (const Money& amount1,
                  const Money& amount2)
{
    return ((amount1.dollars < amount2.dollars) ||
            ((amount1.dollars == amount2.dollars) &&
             (amount1.cents < amount2.cents)));
}

```

9. 要理解为什么不会出现循环，首先想一想重载的基本信息：一个函数或运算符可以有二个或者多个定义。也就是二个或者多个不同的运算符可以共享同一个名字。在下面的语句中：

```
outputStream << "$-";
```

运算符“<<”是定义在 iostream 库中的运算符名字，它的第二个参数是一个字符串常量。而示例 8.5 中定义的运算符“<<”则是另外一个不同的运算符，它的第二个参数类型是 Money 类类型。

10. 如果“<<”和“>>”运算符按照我们期望的方式工作，那么第一个操作数（参数）必须是 cout 或者 cin（或者其他文件 I/O 流）。但如果我们希望将其重载为 Money 类的成员运算符，那么第一个操作数必须是调用对象，即它的类型必须是 Money 类类型，但这不是我们所期望的。

#### 11. // 使用 iostream 库。

```

istream& operator >>(istream& inputStream,
                    Percent& aPercent)
{
    char percentSign;
    inputStream >> aPercent.value;
    inputStream >> percentSign; // 丢弃%符号。
    return inputStream;
}

```

```
// 使用 iostream 库。
ostream& operator<< (ostream& outputStream,
                    const Percent& aPercent)
{
    outputStream << aPercent.value << '%';
    return outputStream;
}
```

12. 语句是合法的，但其含义却不是我们所期望的。(a++) 使 a 中成员变量的值增加 1，但是 (a++)++ 却是将对象 a++ 中的成员变量的值增加 1，这里 a++ 和 a 是不同的对象（可以重新定义运算符 ++，使 (a++)++ 将 a 中的成员变量的值增加 2 是可以做到的，但这需要用到 this 指针，这部分内容会在第 10 章中讨论）。

## 编程练习

- 修改示例 8.5 中 Money 类的定义，使类的定义中包含下面的内容：
  - 重载 “<”、“<=”、“>” 和 “>=”，使得它们可以对 Money 类的对象进行操作。（提示：参见自测练习题 8。）
  - 在类的定义中添加如下的成员函数（这里仅仅提供了函数声明）。

```
Money Money::percent(int percentFigure) const
// 返回调用对象货币数目的百分数。例如，如果参数 percentFigure 的值为 10，
// 那么函数调用的返回值为调用对象中货币数目的 10%。
```

例如，如果 purse 是类 Money 的对象，代表的值为 \$100.10，那么下面的函数调用：

```
purse.percent(10);
```

返回 \$100.10 的 10%，也就是说该调用会返回一个 Money 对象，且代表的货币值是 \$10.01。

- 定义一个代表有理数的类。有理数是指那些可以表示为两个整数之商的数，例如，1/2、3/4、64/2 等都是有理数（这里的 1/2 以及其他的除法都是日常数学中用到的除法，而不是 C++ 编程中的整数除法）。有理数可以表示为两个 int 类型的值，一个表示分子，一个表示分母，类名为 Rational。该类包含一个带有两个参数的构造函数，此构造函数可以将对象的成员变量设置为任何合理的值。另外，还包括一个带有一个 int 类型参数的构造函数，该形参名为 wholeNumber，该构造函数可以将对象初始化为有理数 wholeNumber/1。此外，还包含一个默认的构造函数，将对象初始化为 0（即 0/1）。重载输入和输出运算符 “<<” 和 “>>”。有理数的输入和输出按照如下的形式：1/2、15/32、300/401 等。需要注意，分子和分母都有可能包含符号，即 -1/2、15/-32 以及 -300/-401 等。重载下面的运算符，使它们可以对 Rational 类型的对象进行操作：==、<、<=、>、>=、+、-、\* 和 /。编写一个程序对该类进行测试。



提示：两个有理数  $a/b$ 、 $c/d$  相等，当且仅当  $a*d == c*b$ 。 $b$  和  $d$  都是正有理数的情况下， $a/b$  要小于  $c/d$  必须满足： $a*d < c*b$ 。首先应该对有理数进行化简，从而保证分母总为正数，并且分子和分母应该尽可能的小。例如，有理数  $4/-8$  化简之后变为  $-1/2$ 。

3. 定义一个表示复数的类，复数的形式表示如下：

$$a + b * i$$

其中  $a$  和  $b$  是 `double` 类型的数值，而  $i$  则代表 -1 的平方根。将复数表示为两个 `double` 类型的值。复数类的两个成员变量分别为 `real` 和 `imaginary`，其中与  $i$  相乘的变量为 `imaginary`，复数类名为 `Complex`。该类包含一个带有两个 `double` 类型参数的构造函数，可以将对象的成员变量设置为任何合理的值。此外还包含一个带有一个 `double` 类型参数的构造函数，该形参名为 `realPart`，该构造函数将复数初始化为 `realpart+0*i`。另外还应包含一个默认构造函数，将复数初始化为 0。重载下面的运算符，使它们可以对 `Complex` 类的对象进行操作：`==`、`+`、`-`、`*`、`<<` 和 `>>`。同样编写一个程序对完成的类进行测试。提示：两个复数的加法和减法操作就是对两个 `double` 类型的成员变量分别进行加法和减法。而两个复数的乘积则按照如下的公式进行：

$$(a+bi)*(c+di) == (a*c-b*d)+(a*d+b*c)*i$$

在接口文件中，应该定义一个常量  $i$ ，如下所示：

```
const Complex i(0, 1);
```

4. 按照下面的步骤，修改示例 8.7 中的程序。

- a. 示例 8.7 中，将私有的 `char` 类型的成员变量 `first` 和 `second` 替换为一个大小为 100 的 `char` 类型的数组和一个私有数据成员 `size`。

给出一个默认构造函数，将 `size` 初始化为 10 并将字符数组的前 10 个元素设置为 “#”（这里只是用了 100 个位置中的前 10 个）。

定义一个取值函数，用来返回私有成员 `size` 的值。

- b. 对数组运算符 “[ ]” 进行重载，使得可以用一个小于 `size` 的非负整数作为索引来对数组进行访问并设置其中的值。
- c. 添加一个带有一个 `int` 类型参数 `sz` 的构造函数，该构造函数将字符数组的前 `sz` 个元素初始化为 “#”。
- d. 添加一个构造函数，该构造函数有两个参数：一个 `int` 类型的参数 `sz`，一个是大小为 `sz` 的字符数组。该构造函数将私有字符数组的前 `sz` 个元素初始化为参数字符数组中对应的元素。

注意：在对程序进行测试时，应设计良好的测试用例，包括边界值以及故意使用不正确的值进行测试。程序中可以不包含检查代码越界的代码，但是应该包含相应的错误处理。方式可以是给出错误信息然后退出，也可以是让用户重新输入一个正确的值。

5. 定义一个名为 `Vector2D` 的类，用于存储二维向量的信息。类中应该包含用来获取和设置  $x$  分量和  $y$  分量的成员函数（其中  $x$  和  $y$  的类型都为 `int` 类型）。然后，对 “ $*$ ” 运算符进行重载，使它返回两个向量的点积。二维向量  $A$  和  $B$  的点积等于：

$$(A_x \times B_x) + (A_y \times B_y)$$

编写一个程序对该类进行测试。

6. 定义一个名为 `MyInteger` 的类，用来存储整数。该类包含用来获取和设置整数值成员函数。重载 “[ ] ” 运算符，使之返回位置  $i$  处的数位，其中  $i=0$  是整数的最低位。如果不存在这样的数位，则返回  $-1$ 。例如，如果  $x$  是 `MyInteger` 类的对象并被设置为 418，则  $x[0]$  应该返回 8， $x[1]$  应该返回 1， $x[2]$  应该返回 4，而  $x[3]$  应该返回  $-1$ 。
7. 定义一个用来存储素数、名为 `PrimeNumber` 的类，其默认的构造函数将素数设置为 1。添加另外一个构造函数，允许使用者设置一个具体的素数。添加一个函数，用以获取该素数。最后，针对运算符  $++$  和  $--$  重载前缀和后缀操作，从而可以返回下一个素数（对  $++$  运算符）和之前的素数（对  $--$ ）。例如，如果该类对象的素数被设置为 13，那么调用  $++$  将返回一个代表素数 17 的 `PrimeNumber` 对象。为该函数编写相关的测试程序。
8. 重做第 6 章的编程练习 10，同样是编写 `Temperature` 类，只不过这次将  $=$ 、 $<<$  和  $>>$  运算符重载为成员函数。如果两个对象代表的温度值相同，那么运算符  $==$  返回 `True`。运算符  $<<$  用来输出 `Temperature` 对象对应的华氏温度，而运算符  $>>$  则用来输入一个华氏温度。为该类编写相应的测试程序。
9. 第 6 章的编程练习 12 要求你编写一个名为 `BoxOfProduce` 的类，该类用来存储三捆要寄送给客户的蔬果（存放在一个声明大小为 3 的字符串数组中）。重写该类的定义，这次采用向量而不是字符串数组，此外，添加合适的构造函数以及取值、赋值函数。该类应该包含一个添加额外蔬果的成员函数，从而其包含的蔬果可能多于三捆。名为 `output` 的成员函数可以输出该类包含的蔬果内容。对加法运算符 “ $+$ ” 进行重载，从而可以对两个 `BoxOfProduce` 对象中的蔬果进行合并，并返回一个新的合并后的对象。编写程序测试你完成的类以及重载的加法运算符。本题目只要求完成对 `BoxOfProduce` 类的修改，不必完成编程练习 12 的其他程序实现。





# 字符串

## 9.1 字符串类型数组 304

C字符串值和C字符串变量 304

陷阱：对C字符串使用“=”和“==” 308

<cstring>中的其他函数 309

示例：命令行参数 311

C字符串的输入/输出 313

## 9.2 字符操作工具 315

字符的输入/输出 315

成员函数get和put 315

示例：使用换行函数检查输入 318

陷阱：输入时没有处理‘\n’ 319

成员函数putback、peek和ignore 320

字符操作函数 322

陷阱：函数toupper和tolower返回int型数值 324

## 9.3 标准string类 325

标准string类简介 325

string类的输入和输出 327

提示：getline函数的其他版本 330

陷阱：对cin同时使用>>和getline 331

使用string类处理字符串 331

示例：回文检测 335

string类对象和C字符串的转换 338

# 第9章 字符串

波洛尼厄斯：您在读些什么，殿下？哈姆雷特：词，词，词。

莎士比亚，《哈姆雷特》

## 概述

本章我们要介绍两种数据类型，它们可以表示像“Hello”这样的一串字符。第一种数据类型是由基本类型 `char` 组成的字符数组，这种数组需要在末尾位置由一个空字符——‘\0’——表示字符串的结束。这是 C++ 从 C 语言继承过来的表示字符串的古老方法，所以我们可以将这类字符串叫作 C 字符串。尽管比较古老，但 C++ 中对字符串的处理无法离开 C 字符串。例如，像引用字符串“Hello”，在 C++ 中就是以 C 字符串的形式实现的。

ANSI/ISO C++ 标准包括一种现代的处理字符串机制，也就是我们接下来要讨论的另一种数据类型——`string` 类。如果可以在 C 字符串和 `string` 类之间选择的话，大家通常都会选择 `string` 类，因为它提供了更强大的功能，而且对安全性要求较高的应用提供了错误检查的支持。`string` 类的实现机制使用了模板，所以它很像标准模板库 (STL) 中的模板类。模板和 STL 的知识将分别在 16 章和 19 章中介绍，本章我们只讨论 `string` 类的基础知识，不需要事先了解模板的相关知识。

接下来的内容并不要求你有关于数组的深入知识，但是你必须熟知基本的数组符号，比如 `a[i]`。5.1 节的知识已经足够支持你学习本章的内容。同样你也不需要深入了解关于类的知识，9.1 节中关于 C 字符串的内容和 9.2 节中关于字符操作的内容与第 6、7、8 章中关于类的知识并没有多大联系。不过，在阅读 9.3 节之前，你需要阅读第 6 章和第 7 章的如下部分：7.1 节和 7.2 节中的“`const` 修饰符”小节。

## 9.1 字符串类型数组

从一开始便应考虑到结局。

让·德·拉封丹，《寓言集》卷三 (1668)

本节我们将介绍 C++ 从 C 语言继承而来的表示字符串的方法，在 9.3 节我们再学习更现代的 `string` 类。尽管传统的 C 语言字符串显得有些过时，但是它至今仍然被广泛地使用，并且它是完整的 C++ 语言的一部分。

### C字符串值和C字符串变量

在 C++ 中，我们可以使用一个 `char` 类型的数组来表示一个字符串。以“Hello”为例子，我们可以方便地用六个索引过的变量表示它：五个字符表示“Hello”，还有一个字符‘\0’表示字符串的结尾。字符‘\0’被叫作空字符，它可以被用来表示一

## C 字符串

个字符串的结束符，因为它可以与所有实际的字符区分开来。结束符使程序可以逐个地读取数组中的字符，在读到‘\0’的时候知道这个字符串已经结束了。保存在这种以‘\0’作为结尾的数组中的字符串被称为**C 字符串**。

虽然我们在写程序的时候将结束符‘\0’写为两个字符，但是就像换行符‘\n’一样，结束符‘\0’实际上只代表一个字符值。如同其他的字符值一样，‘\0’可以被保存在一个 char 类型的变量中，或者是字符串数组的一个元素中。

## 空字符 ‘\0’

空字符‘\0’用来标识存储在一个数组中的 C 字符串的结尾，这个数组常常被叫作 C 字符串变量。尽管‘\0’书写的时候占用了两个字符，但实际上它和一个字符型变量或字符串数组中的一个元素一样，只占用一个字符的空间。

现在你已经使用过 C 字符串了。在 C++ 中，虽然没有必要了解所有具体细节，但像“Hello”这样的字符串常量是以 C 字符串形式存储的这种细节知识我们还是应该掌握的。

C 字符串  
变量

一个 C 字符串变量实际上是一个字符型数组。所以，像下面这样声明一个字符数组就生成了一个 C 字符串变量，可以存储长度不超过 9 的 C 字符串变量。

```
char s[10]
```

其中的 10 表示可以存储九个字符和一个字符串结束符‘\0’。

C 字符串变量是没有被填充满的字符数组。和其他未被填充满的数组一样，C 字符串变量使用的存储空间从索引位置 0 开始。不过 C 字符串并不像其他数组一样用一个整数类型的变量标记元素的索引位置，而只是在末尾字符后面添加一个结束标识符‘\0’。举个例子，对于字符串常量“Hi Mom!”，数组元素将这样填充存储空间：

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\0	?	?

字符‘\0’用来标记 C 字符串的结束。当从数组元素 s[0] 开始，依次读取字符串中的字符 s[1]，s[2] 等，遇到结束符‘\0’的时候，程序会知道此时自己已经读到了 C 字符串的末尾。由于结束符‘\0’也要占用数组存储空间的一个存储单元，所以以一个数组所能存储的字符串的长度将始终比数组的容量少 1。

C 字符串  
变量和字符  
数组

C 字符串变量和一般的字符数组的区别就是 C 字符串的末尾必须是空字符‘\0’。这种区别并不是将字符数组分为了两类，这只是对字符数组的两种不同使用方式而已。一个 C 字符串实质上是一个字符数组，是对字符数组的一种特殊使用方式。

## 初始化

在声明 C 字符串的同时可以初始化它，具体的声明方法如下：

```
char myMessage[20] = "Hi there.";
```

C 字符串  
变量

值得注意的是，C 字符串变量的值并不需要将其所在的字符数组填满。

### C 字符串变量的声明

C 字符串变量本质是一个字符数组，但它的使用方式与普通的字符数组不同。C 字符串的声明应该和普通的字符数组相同。

#### 语法

```
char Array_Name[Maximum_C-string_Size + 1];
```

#### 示例

```
char myCString[11];
```

存储 ‘\0’ 需要一个额外的数组单元，它用来标记数组中 C 字符串的结束。比如，上面的 C 字符串变量 myCString 只能用来保存长度为 10 的字符串。

如果 C 字符串声明的同时就对它初始化，我们可以忽略数组声明时必要的容量声明，C++ 将会自动分配字符串常量长度 +1 的存储空间。（超出的一个存储单元用来存储结束符 ‘\0’。）比如：

```
char shortString[] = "abc";
```

是和下面的声明等价的：

```
char shortString[4] = "abc";
```

不过，一定不要将下面的两个初始化语句混为一谈：

```
char shortString[] = "abc";
```

和

```
char shortString[] = {'a', 'b', 'c'};
```

它们是不同的。第一种初始化方式会在字符 ‘a’、‘b’ 和 ‘c’ 后面添加空字符 ‘\0’。而第二种初始化不会添加。

### 初始化一个 C 字符串变量

C 字符串变量可以在声明的同时进行初始化，比如下面这个例子：

```
char yourString[11] = "Do Be Do";
```

这样的初始化方式会自动在数组中 C 字符串结束的位置添加一个空字符 ‘\0’。

如果不给出方括号中的数字，那么这个 C 字符串变量的长度将是字符串值长度 + 1。比如，下面声明的字符串将包含九个数组元素（前八个存储 C 字符串 “Do Be Do”，最后一个存储结束字符 ‘\0’）。

```
char myString[] = "Do Be Do";
```

### C 字符串 变量的索引 变量

由于 C 字符串变量是一种数组，所以其中的元素也可以像其他数组的元素一样通过索引来访问。假如程序中含有如下的 C 字符串变量声明：

```
char ourString[5] = "Hi";
```

根据上面的声明，在程序中我们可以使用这样索引的方式访问五个数组元素：

ourString[0]、ourString[1]、ourString[2]、ourString[3] 和 ourString[4]。例如,下面的代码将 ourString 的内容修改为一个全部字符都为‘x’的字符串:

```
int index = 0;
while (ourString[index] != "\0")
{
    ourString[index] = "X";
    index++;
}
```

不要漏掉  
‘\0’

像这样使用索引访问字符串中的元素的时候,千万不要将结尾的空字符 ‘\0’ 用别的字符替换掉了。如果数组中没有了空字符 ‘\0’, 那么它将不再表示一个字符串。例如,下面的代码修改了数组 happyString 的内容,使它不再代表一个字符串:

```
char happyString[7] = "DoBeDo";
happyString[6] = "Z";
```

上面的代码执行结束后,六个字符 “DoBeDo” 仍然存在于数组 happyString 中,但是由于不包含用来标示 C 字符串结束的空字符 ‘\0’, happyString 已经不是一个字符串了。很多处理字符串的函数都需要用空字符来判断字符串是否已经到了结尾。

我们来看另外一个例子。在之前的代码中有一个更改 C 字符串变量 ourString 的 while 循环,遇到空字符 ‘\0’ 的时候 while 循环结束。如果这个循环中一直没有检测到 ‘\0’, 那么大量的内存单元都会被填充上非预期的值,这可能导致程序出现无法预测的行为。出于安全的考虑,最好能改进上面的 while 循环,即使不小心没有将空字符 ‘\0’ 写在字符串的结尾,循环也会在遍历完数组后自动结束。改写后的 while 循环如下:

```
int index = 0;
while ( (ourString[index] != "\0") && (index < SIZE) )
{
    ourString[index] = "X";
    index++;
}
```

我们假设 SIZE 是一个已经定义过的变量,值就是数组 ourString 的大小。

### <cstring> 库

在声明和初始化 C 字符串的时候,我们并不需要用到任何 include 指令或者 using 语句。但是在处理 C 字符串的时候,很自然地需要用到一些 <cstring> 库里定义的字符串处理函数。所以当代码需要处理 C 字符串的时候,经常会在源代码文件的开头包含如下的 include 语句:

```
#include <cstring>
```

<cstring> 库中所有的定义均定义在全局命名空间中,而不是 std 中,所以并不需要 using 语句指明命名空间。





陷阱：对 C 字符串使用 “=” 和 “==”

C 字符串值和 C 字符串变量不同于一般数据类型的值和变量，许多常用的运算符并不适用于 C 字符串。我们不能使用 “=” 来为 C 字符串变量赋值；使用 “==” 来比较两个 C 字符串是否相同也是不正确的。造成这些的原因是 C 字符串和 C 字符串变量实质都是数组。

#### 为 C 字符串 变量赋值

为一个 C 字符串变量赋值并不像为其他数据类型变量赋值那么方便，例如下面这个语句就是非法的：

```
char aString[10];
aString = "Hello";
```

← 非法

尽管声明字符串变量的同时可以用等号为它赋值，但这种赋值并不是在程序的任何地方都是合法的。事实上，像下面这样在声明的同时使用等号被称为初始化而不是赋值：

```
char happyString[7] = "DoBeDo";
```

如果想要为一个字符串变量赋值，必须使用其他方法。

对 C 字符串变量赋值有很多种方法，其中最简单的方式是使用定义好的函数 `strcpy()`，使用方法如下所示：

```
strcpy(aString, "Hello");
```

上面的语句将 `aString` 赋值为 “Hello”，不幸的是，这个版本的 `strcpy` 函数没有正确性检查的功能。也就是说，`strcpy` 函数不能保证拷贝的字符串值不超过第一个参数表示的字符串变量的容量。更多的 C++ 版本里有三个参数的 `strcpy` 函数，第三个函数表示了要拷贝的最大字符数，只要这个参数比第一个参数的容量小 1，那么 `strcpy` 的字符串拷贝操作就是安全的，当然这是在你的 C++ 版本中包含三个参数的 `strcpy` 函数的前提下。例如：

```
char anotherString[10];
strcpy(anotherString, aStringVariable, 9);
```

上面的语句中，不论 `aStringVariable` 的实际长度是多长，`aStringVariable` 中的字符最多只能被拷贝九个。

#### 比较 C 字符 串是否相符

同样，我们也不能使用 “==” 来比较两个 C 字符串是否相等。（实际情况更加糟糕，C++ 允许我们使用 “==” 运算符来比较两个 C 字符串变量，但是这种比较并不能检验出两个 C 字符串变量中的值是否相等。所以，当我们使用 “==” 运算符比较两个 C 字符串的时候，编译器并不会给出错误信息，但是我们得到的结果是错误的！）

要比较两个 C 字符串，可以使用预定义的函数 `strcmp`。比如：

```
if (strcmp(cString1, cString2))
    cout << "The strings are NOT the same.";
else
    cout << "The strings are the same.";
```

值得注意的是，函数 `strcmp` 的使用也许和大家猜测的不一致。如果两个字符串不相同，那么函数返回结果 `true`。函数 `strcmp` 会依次比较两个字符串同一位置上

## 字典顺序

的单个字符，如果发现 `cString1` 中的字符数字编码值小于 `cString2` 中字符的数字编码值，那么函数返回一个负值。相反，如果发现 `cString1` 中的字符数字编码值比 `cString2` 中字符的数字编码值大，那么函数返回一个正值（某些情况下 `strcmp` 函数会返回字符编码的差异，不过对你来说这并不重要）。如果所用字符都相同，则返回 0。比较字符时的大小次序关系叫作字典顺序。如果字符串只包含小写字母或者只包含大写字母，那么字典顺序恰好就是字母顺序。

`strcmp` 函数返回的负值、正值和 0 对应于两个字符串的比较结果是小于、大于和等于。如果你将 `strcmp` 作为一个布尔表达式用在 `if` 判断或者循环语句中，来比较两个字符串时，如果字符串相等，要把 `strcmp` 返回的非零值转换成 `true`；如果字符串不相等，要把 `strcmp` 返回的零值转换成 `false`。在比较字符串的时候要记得这个逻辑上的转换。

一些支持标准的 C++ 编译器包含 `strcmp` 函数的安全版本，使用第三个参数指明要比较的最大的字符数。

函数 `strcpy` 和 `strcmp` 都存在于头文件为 `<cstring>` 的库中，所以如果要使用这两个函数，必须在代码文件的开头部分写上下面这条 `include` 语句：

```
#include <cstring>
```

`strcpy` 和 `strcmp` 函数都定义在全局命名空间中，而不是 `std` 命名空间，所以不需要使用 `using` 语句。■

**<cstring>中的其他函数**

示例 9.1 中出现了几个 `<cstring>` 中常见的函数，在使用这些函数的时候务必在代码文件的开头部分写上如下的 `include` 语句：

```
#include <cstring>
```

由于 `<cstring>` 库中所有的函数都定义在全局命名空间而不是 `std` 命名空间中，所以不需要使用 `using` 语句。

我们已经介绍过 `strcpy` 和 `strcmp` 函数了，还有一个 `strlen` 函数也很常用。用一个例子说明它的功能，`strlen("dobedo")` 返回 6，因为字符串 "dobedo" 包含六个字符。

还有函数 `strcat`，它用于两个 C 字符串的首尾连接，也就是将两个较短的字符串连接成一个较长的 C 字符串。`strcat` 函数的第一个参数只能是 C 字符串变量，第二个参数则可以是任何能表示 C 字符串的表达式，比如字符串常量。连接后的较长的字符串保存在函数的第一个参数的存储空间中。例如下面的语句：

```
char stringVar[20] = "The rain";
strcat(stringVar, "in Spain");
```

这两句代码将 `stringVar` 的值修改成了 "The rainin Spain"。从这个例子里可以看出，在连接两个 C 字符串的时候要考虑空格。在示例 9.1 中包含了一个更加安全的、带三个参数的 `strcat` 函数，不过并不是在所有版本的 C++ 中都能看到这个函数。

## C 字符串实参和形参

一个 C 字符串实质是一个数组，所以一个函数的 C 字符串形参实质是一个简单的数组形参。

如果函数有数组形参，而且函数要更改这个字符串形参的值，那么最安全的做法是增加一个 `int` 型的参数来指明 C 字符串的长度。

另外，如果函数只是将字符串作为实参而且不需要修改它的值，那么用来指定数组大小或者数组内实际内容长度的 `int` 型参数就无须给出了。这是因为 C 字符串变量中的空字符 `'\0'` 可以表示 C 字符串已经结束。

### 示例 9.1 库 `<cstring>` 中的预定义字符串函数

函数	描述	注意
<code>strcpy</code> ( <code>Target_String_Var</code> , <code>Src_String</code> )	将 C 字符串 <code>Src_String</code> (源字符串) 中的内容拷贝到 C 字符串变量 <code>Target_String_Var</code> (目标字符串) 中	无法检测 <code>Target_String_Var</code> 的大小是否能容纳 C 字符串 <code>Src_String</code> 的内容
<code>strncpy</code> ( <code>Target_String_Var</code> , <code>Src_String</code> , <code>Limit</code> )	与上面两个参数的 <code>strcpy</code> 函数类似，但是最大拷贝字符数限制为 <code>Limit</code> 指定的数值	选择适当的 <code>Limit</code> 的值可以得到一个安全的 <code>strcpy</code> 函数，但不是所有版本的 C++ 都支持这个函数
<code>strcat</code> ( <code>Target_String_Var</code> , <code>Src_String</code> )	将字符串 <code>Src_String</code> 的内容合并到字符串变量 <code>Target_String_Var</code> 的结尾处	无法检测字符串变量 <code>Target_String_Var</code> 的大小能否容纳合并后的字符串
<code>strncat</code> ( <code>Target_String_Var</code> , <code>Src_String</code> , <code>Limit</code> )	和上面两个参数的 <code>strcat</code> 函数类似，但是最大合并的字符数限定为 <code>Limit</code> 指定的数值	选择适当的 <code>Limit</code> 的值可以得到一个安全的 <code>strcat</code> 函数，但不是所有版本的 C++ 都支持这个函数
<code>strlen</code> ( <code>Src_String</code> )	返回字符串 <code>Src_String</code> 的长度 (空字符 <code>'\0'</code> 不算在内)	
<code>strcmp</code> ( <code>String_1</code> , <code>String_2</code> )	依据字典顺序，如果 <code>String1</code> 和 <code>String2</code> 相同则返回 0，如果 <code>String1</code> 比 <code>String2</code> 小则返回负值，如果 <code>String1</code> 比 <code>String2</code> 大则返回正值	如果 <code>String1</code> 与 <code>String2</code> 相同则返回 0 值，这对应于布尔值中的 <code>false</code> ，注意这与惯有的思路相反
<code>strncmp</code> ( <code>String_1</code> , <code>String_2</code> , <code>Limit</code> )	与上面两个参数的 <code>strcmp</code> 函数类似，但最大可比较的字符数被限制为 <code>Limit</code> 所指定的数值	选择适当的 <code>Limit</code> 的值可以得到一个安全的 <code>strcmp</code> 函数，但不是所有版本的 C++ 都支持这个函数

### 示例：命令行参数

迄今为止我们还没有为 main 函数设置过参数，实际上是可以向 main 函数传递参数的。输入的参数将在通过命令行调用我们的程序的时候被匹配，比如一台 UNIX 机器的命令行：

```
ls /home
```

调用了 ls 程序，程序的参数是 /home，所以命令行将会列出 /home 目录下的内容。为了设置一个 C++ 命令行程序的参数，我们需要使用这样的 main 函数头：

```
int main (int argc, char * argv[])
```

参数 argc 是一个指定程序将接收多少个参数的整型数。程序的名字默认被当作一个参数，所以 argc 的值最小是 1。

参数 argv 是一个 C 字符串类型的数组。argv[0] 是程序的名称。argv[1] 是第一个参数的名字，argv[2] 是第二个参数的名字，依此类推，直到 argv[argc-1]。

举个例子，如果程序的名字是 getPalindromes，调用它的命令行是：

```
getPalindromes string1 string2
```

那么在 main 函数里，argc = 3，argv[0] = "getPalindromes"，argv[1] = "string1"，argv[2] = "string2"。如果程序中要使用 argv，那么一定要确保 argc 的值是正确的，否则程序在接收参数时将会发生错误。

### 自测练习题

1. 下面的声明语句哪些是等价的？

```
char stringVar[10] = "Hello";  
char stringVar[10] = {'H', 'e', 'l', 'l', 'o', '\0'};  
char stringVar[10] = {'H', 'e', 'l', 'l', 'o'};  
char stringVar[6] = "Hello";  
char stringVar[] = "Hello";
```

2. 下面的代码运行后变量 singingString 中保存的 C 字符串是什么？

```
char singingString[20] = "DoBeDo";  
strcat(singingString, " to you");
```

假设这段代码是在一段完整可运行的程序中且程序中已经使用了 include <cstring> 语句。

3. 下面的代码是否存在错误？请指出。

```
char stringVar[] = "Hello";  
strcat(stringVar, " and Good-bye.");  
cout << stringVar;
```

假设这段代码是在一段完整可运行的程序中且程序中已经使用了 include <cstring> 语句。

4. 如果用来返回字符串长度的函数 `strlen` 并不是一个预定义好的函数, 请写出 `strlen` 函数的定义。`strlen` 函数只能有一个参数, 即一个 C 字符串。
5. 下面声明的字符串变量最多能保存多长的字符串? 请给出答案和解释。

```
char s[6];
```

6. 下面的常量分别包含多少个字符?

- a. `'\n'`
- b. `"n"`
- c. `"Mary"`
- d. `"M"`
- e. `"Mary\n"`

7. 字符串的本质是一个 `char` 类型的数组, 那下面两种声明和初始化方式到底有什么区别?

```
char shortString[] = "abc";
char shortString[] = { 'a', 'b', 'c' };
```

8. 有一个如下的字符串变量, 请写一个循环程序为这个字符串变量赋值, 使它所有的字符都是 X, 字符串长度不能变化。

```
char ourString[15] = "Hi there!";
```

9. 有如下的字符串声明, 其中的 `SIZE` 是已定义好的常量:

```
char ourString[SIZE];
```

字符串变量 `ourString` 已经在某个地方赋值过了, 后面的循环程序在保证字符串长度不变的前提下将字符串的内容都修改为了字符 X, 请回答代码后面的问题:

```
int index = 0;
while (ourString[index] != '\0')
{
    ourString[index] = 'X';
    index++;
}
```

- a. 请解释这段代码出现的问题, 它会修改数组之外的内存值吗? 为什么?
  - b. 请修改这段代码中出现的問題。
10. 利用库函数编写一段代码, 将字符串常量 "Hello" 拷贝到下面的字符串变量中。注意在使用库函数前一定要使用 `#include` 包含所需的头文件。

```
char aString[10];
```

11. 执行下面的代码后, 会有什么样的输出?

```
char song[10] = "I did it";
char franksSong[20];
strcpy ( franksSong, song );
strcat ( franksSong, "my way!");
cout << franksSong << endl;
```

12. 请指出下面代码存在的问题:

```
char aString[20] = "How are you?";
```

```
strcat(aString, "Good, I hope.");
```

## C字符串的输入/输出

可以使用插入运算符“<<”来输出 C 字符串。实际上之前我们已经使用“<<”输出过字符串常量了。对于字符串变量，使用方式也是一样的，比如：

```
cout << news << "Wow.\n";
```

其中的 news 是一个 C 字符串变量。

使用“>>”运算符来输入字符串变量也是可以的，但必须记住的是，和输入其他数据类型一样，当 C 字符串按照这种方式输入时，所有的空白符（空格、制表符以及换行符）将被忽略掉。当碰到任意空格和换行符时一个输入就结束了。例如，考虑下面的代码：

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << b << "END OF OUTPUT\n";
```

当把上面的代码放在一个完整的程序中时，就会出现类似下面的运行结果：

```
Enter some input:
Do be do to you!
DobeENO OF OUTPUT
```

C 字符串从变量 a 和 b 都读入了一个单词：变量 a 接收到的字符串值为“Do”，因为紧靠着 Do 后面的字符为空格；变量 b 接收到的字符串为“be”，因为在 be 后面的字符为空格。

如果程序要读入一整行输入，我们可以通过输入运算符“>>”来一次读入一个单词。不过这样做不但麻烦，而且可能读到一行中的空格符。还有一种更加简单的方法，可以一次读入一整行的输入，并将输入作为字符串存储在一个变量中。这种方法使用预定义的成员函数 getline 来读入，getline 是所有输入流（例如 cin 或者文件输入流）的成员函数，该函数有两个参数。第一个参数是接收输入内容的 C 字符串变量；第二个参数是一个整型数值，一般是声明的字符串变量的大小。函数的第二个参数指明了 C 字符串变量所能接受的最大字符数。例如，下面的代码：

getline

```
char a[80];
cout << "Enter some input:\n";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

当把这些代码放到一个完整的程序中时，就会产生类似下面的运行结果：

```
Enter some input:
Do be do to you!
Do be do to you! END OF OUTPUT
```

使用函数 cin.getline，就可以将整行输入读入。读入过程会到输入行末尾才结束，即使已经读入的字符串长度比函数第二个参数指定的字符数少。

另外，当执行 getline 函数时，如果已经读入了第二个参数所允许的字符数，即

使输入的内容还没有到达末尾,输入操作也将结束。例如下面的代码:

```
char shortString[5];
cout << "Enter some Input:\n";
cin.getline(shortString, 5);
cout << shortString << "END OF OUTPUT \n";
```

当把上面的代码放到一个完整的程序中时,就会产生类似下面的运行结果:

```
Enter some input:
dobe d o w a p
dobeEND OF OUTPUT
```

值得注意的是,尽管 getline 函数的第二个参数是 5,但是字符串变量 shortString 一共读入了四个而非五个字符。这是因为空字符 '\0' 占据了一个数组的位置。存储在 C 字符串变量中的每一个 C 字符串都是以空字符作为结尾的,而且空字符永远会占据一个存储单元。

文件输入/  
输出

我们将展示 cout 和 cin 的用法,它们用于 C 字符串的输入和输出。此外,它们也可以用于文件的输入和输出。输入流 cin 可以由一个文件输入流代替;同样,输出流 cout 也可以由一个文件输出流代替。(文件输入/输出流将在第 12 章中讨论。)

### getline

成员函数 getline 可以用于整行内容的读入,读入的内容被保存在一个 C 字符串变量中。

#### 语法

```
cin.getline (String_Var, Max_characters + 1);
```

整行的内容是从流 Input\_Stream 中读取的,然后被保存在变量 String\_Var 中。如果这一整行的字符数比 Max\_characters 多,那么只读入最前面的 Max\_characters 个字符。(+1 的原因是每个 C 字符串都要以 '\0' 结尾,所以保存在变量 String\_Var 中的字符串长度不会超过读入的字符数。)

#### 示例

```
char oneLine[80];
cin.getline(online, 80);
```

在第 12 章中我们将会看到如何使用与文件关联的输入流来替代 cin。

### 自我练习

13. 假设有下面的代码:

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << '-' << b << "END OF OUTPUT\n";
```

在代码运行时输入下面的内容,那么接下来的输出是什么?

```
Enter some input:
The
    time is now.
```

14. 假设有下面的代码：

```
char myString[80];
cout << "Enter a line of input:\n";
cin.getline(myString, 6);
cout << myString << "<END OF OUTPUT">";
```

在代码运行时输入下面的内容，那么接下来的输出是什么？

```
Enter a line of input:
May the hair on your toes grow long and curly.
```

## 9.2 字符操作工具

他们把那个词拼写为“Vinci”，读作“Vinchy”；外国人的拼写总是强于他们的发音。

马克·吐温，《流浪汉在海外》

无论什么样的字符串都是由单个字符构成的。所以当处理字符串的时候，使用一些可以处理和操作 `char` 型数据的工具是非常有必要的。本节将介绍这些工具。

### 字符的输入/输出

所有的数据都是以字符数据的形式输入/输出的。当程序输出数字 10 时，实际上输出的是 1 和 0 两个字符。同样，如果用户希望输入数字 10，实际上用户键入的是两个字符 1 和 0。计算机最终如何解释“10”这两个字符取决于程序是如何编写的。但是不论程序是如何编写的，计算机设备读入的永远都是 1 和 0 两个字符而不是数字 10。字符和数字之间的转换一般都是自动进行的，我们不必关注其细节，但是有时这种自动类型转换会变成一种障碍，所以 C++ 提供了一些可以在底层使用的工具，使我们可以跳过自动类型转换，让字符的输入/输出完全按照我们希望的进行。如果需要，我们甚至可以编写一个函数将 `int` 型的数字按照罗马数字的形式输入或输出。

### 成员函数 `get` 和 `put`

程序使用函数 `get` 读入一个字符，并将字符保存在 `char` 型变量里。所有输入流，不论是文件输入流还是 `cin`，都包含一个 `get` 成员函数。我们在这里讨论的时候将 `get` 作为对象 `cin` 的成员函数来讨论（当我们学习第 12 章的文件输入/输出时，可以看到文件输入流的 `get` 函数和 `cin` 函数十分相似）。

在此之前，要从输入中读取一个字符（或者其他类型的输入），需要在 `cin` 后面使用输入运算符 `>>`。当使用输入运算符时，程序会自动完成一些工作，比如跳过空白符。不过有时候我们并不需要跳过空白符，成员函数 `cin.get` 可以直接读取下一个字符，

`cin.get`



而不管它是不是一个空白符。

成员函数 `get` 有一个 `char` 类型的参数，这个参数用来接收从输入流中读取的输入内容，比如下面这个语句将读入键盘输入的下一个字符，将其保存在变量 `nextSymbol` 中：

```
char nextSymbol;
cin.get(nextSymbol);
```

读取空格符  
和 '\n'

需要注意的是，使用这种方式程序可以读入任何字符。如果下一个输入字符是空格，那么程序就将读入一个空格符。如果下一个输入是一个换行符（就是说程序正好读取到一行的末尾），那么上面的 `nextSymbol` 变量的值将被设置为 '\n'。比如程序中包含如下的代码段：

```
char c1, c2, c3;
cin.get(c1);
cin.get(c2);
cin.get(c3);
```

下面的两行作为这个程序的输入：

```
AB
CD
```

那么变量 `c1` 的值将为 'A'，变量 `c2` 的值将为 'B'，变量 `c3` 的值将为 '\n'。注意变量 `c3` 的值并不是 'C'。

检查输入是  
否已经到了  
一行的末尾

成员函数 `get` 的一个用途就是可以检查输入是否已经到了一行的末尾。下面的循环读入一行输入，并在读入到换行符 '\n' 时停止。所有后面的输入将从下一行的开始处读入。作为第一个例子，下面的代码仅仅是将所有的输入简单地反馈回来，但使用这种方法，可以对输入的内容做任何我们需要的操作。

```
cout << "Enter a line of input and I will echo it:\n";
char symbol;
do
{
    cin.get(symbol);
    cout << symbol;
} while (symbol != '\n');
cout << "That's all for this demonstration.\n";
```

上面的循环会读入任意一行输入，并立即将其反馈到屏幕上，包括空格在内。下面是运行结果示例：

```
Enter a line of input and I will echo it:
Do Be Do 1 2 34
Do Be Do 1 2 34
That's all for this demonstration.
```

值得注意的是，换行符 'n' 是被读入后马上被输出的。由于输出了换行符，所以以 "That's" 开头的内容出现在新的一行。

**‘\n’ 和 “\n”**

‘\n’ 和 “\n” 看起来似乎完全一样，在 cout 语句中它们会输出相同的内容，但是并不是所有场合它们都可以相互替换使用。‘\n’ 是 char 类型的值，可以保存在 char 类型变量中。另一方面，“\n” 是一个字符串，只是只包含了一个字符。所以 “\n” 不是 char 类型的值，不能保存在 char 类型的变量中。

put

成员函数 put 与成员函数 get 十分相似，但它不是用于输入而是用于输出的。函数 put 输出一个字符。成员函数 cout.put 带一个参数，这个参数必须是 char 类型的表达式，比如 char 类型的常量或者是变量。当调用 put 函数时，参数的值将会被输出到屏幕上。比如，下面的语句将字母 ‘a’ 输出到屏幕上：

```
cout.put("a");
```

与输出运算符 “<<” 相比，成员函数 cout.put 并不能完成更多的工作，但它与 get 函数相对应，是应该存在的。（后面我们在第 12 章介绍文件的输入/输出时，会看到 put 函数不仅可以和 cout 一起使用，还可以用于文件流的输出。）

**成员函数 get**

成员函数 get 用来读取一个输入的字符。与输入运算符 >> 不同的是，get 函数读取下一行输入的字符，它并不关心输入的字符是什么。实际上，即使接下来的输入字符是空行或者换行符，get 函数也可以读入它们。get 函数有一个 char 类型的参数，当调用 get 函数时，程序将会读入下一个输入的字符，并将其保存在参数指明的变量中。

**示例**

```
char nextSymbol;
cin.get(nextSymbol)
```

在第 12 章我们将会看到，只需使用输入文件流来代替 cin，就可以让 get 函数从文件中读取字符。

和其他使用 cin 和 cout 的时候一样，如果要在程序中使用 cin.get 或者 cout.put，就需要在程序中包含下面的语句：

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

或者

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

### 示例：使用换行函数检查输入

在示例 9.2 的程序中，由函数 `getInt` 来检查用户的输入是否合法，如果用户的输入不恰当，就要求用户重新输入。写这个程序的目的是为了测试 `getInt` 函数，但其实这段代码可以用于任何需要从键盘录入数据的程序。

值得注意的是这里使用了 `newLine` 函数，此函数将原封不动地读取当前行剩余的所有内容，但是并没有处理这些内容，这相当于抛弃了该行剩余的输入。所以如果输入是 “No”，那么程序只会读入第一个字母 N，接着调用 `newLine` 函数将剩余的部分丢弃。这意味着如果用户在新的一行键入 75，那么程序将直接读入 75 而不会试图去读取 No 中剩余的字符 o，其运行结果将在示例 9.2 中展示。如果程序没有使用 `newLine` 函数，那么读入的将会是该行剩余的字符 o，而不是新的一行中输入的数字 75。

#### 示例 9.2 输入检查

```
1 // 程序将测试函数 getInt 和 newLine。
2 #include <iostream>
3 using namespace std;

4 void newLine( );
5 // 丢弃当前行剩余的内容。
6 // 本行末尾的 '\n' 也将被丢弃。

7 void getInt(int & number);
8 // 设置变量 number 的值为用户期望的值。

9 int main( )
10 {
11     int n;

12     getInt(n);
13     cout << "Final value read in = " << n << endl
14          << "End of demonstration.\n";

15     return 0;
16 }

17 // 使用 iostream。
18 void newLine( )
19 {
20     char symbol;
21     do
22     {
23         cin.get(symbol);
24     } while (symbol != '\n');

25 }
```

```

26 // 使用 iostream。
27 void getInt(int & number)
28 {
29     char ans;
30     do
31     {
32         cout << "Enter input number: ";
33         cin >> number;
34         cout << "You entered " << number
35             << " Is that correct? (yes/no): ";
36         cin >> ans;
37         newLine();
38     } while ((ans == 'N') || (ans == 'n'));
39 }

```

### 示例运行结果

```

Enter input number: 57
You entered 57 Is that correct? (yes/no): No No No!
Enter input number: 75
You entered 75 Is that correct? (yes/no): yes
Final value read in = 75
End of demonstration.

```

### 陷阱：输入时没有处理

在使用成员函数 `get` 输入时，我们要考虑到所有输入的字符，甚至包括那些本来无意义的字符，如空格符和换行符 ‘\n’。使用 `get` 函数遇到的最常见的错误就是经常忘记处理每一行末尾的换行符。如果输入流中的换行符没有被处理（通常被丢弃），当程序中的 `get` 函数准备读入下一个字符的时候，实际上读取的却是换行符 ‘\n’。我们可以使用示例 9.2 中定义过的 `newLine` 函数（或者使用下面马上就要出现的 `ignore`）来处理输入流中的换行符。接下来我们看一个具体的例子。

将 `cin` 的不同用法混合使用是合法的。比如，下面的代码是合法的：

```

cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);

```

但是上面的代码其实是有问题的，假如你期望程序按照下面这样运行：

```

Enter a number:
21
Now enter a letter:
A

```

如果程序运行时按照上面那样输入，那么变量 `number` 的值正好是预期的 21。但是让我们失望的是，我们会发现变量 `symbol` 中的值并不是我们预期的 ‘A’ 而是换行符 ‘\n’。在读入数字 21 之后，从输入流中读入的下一个字符并不是 ‘A’ 而是换行符

'\n'。需要一直警惕的是，get 函数并不会自动忽略空格或者换行符。(实际上，如果上面的程序后半部分不需要任何输入，我们甚至连输入字符 'A' 的机会都没有，一旦变量 symbol 读入了换行符，程序就会继续往下执行。如果接下来是输出语句，那么屏幕上将会显示输出的内容，不会再有输入 'A' 机会了。)

将上面的代码进行一些修改，再次按照上面那样输入，这次变量 number 中的值会是 21 而且变量 symbol 中的值也会是我们期望的 'A'。

```
cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin >> symbol;
```

同样，我们也可以使用示例 9.2 中定义的 newLine 改写这个程序：

```
cout << "Enter a number:\n";
int number;
cin >> number;
newLine( );
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);
```

改写后，程序中使用了两种 cin 的使用方式而且可以正常工作，但使用时一定要多加小心。

在接下来的小节中，我们将介绍函数 ignore，也可以利用它来改写上面的程序。■

### 成员函数 putback、peek 和 ignore

在有些情况下，程序需要提前知道输入流中的下一个字符是什么字符。在从流中读取一个字符之后，我们可能会发现这个字符并不是我们需要的，所以我们会想把这个字符“放回去”。比如一个程序需要读入一个空格前面所有的内容，那么当程序读入了第一个空格的时候就应该停止。但是要注意的是，此时流中的空格字符已经被读入了，它已经不再存在于流中了。不过，后续的程序可能还会需要流中的这个空格字符，解决这个问题的方法就是使用 cin.putback 函数。此函数有一个 char 类型的参数，作用就是将参数表示的字符放回到流中，使其成为即将读取到的下一个字符。它的参数可以是任意 char 型表达式，而且使用 putback 将并非最近读入的字符，而是任何其他字符放回流中也是允许的。

putback

成员函数 peek 的功能和它的名字是相配的，它将返回 cin 中马上要被读取的下一个字符，但是并不会真的读入这个字符，该字符会被留在 cin 流中。peek 函数可以被用来预知下一个将要读入的字符是什么。

peek

如果需要忽略某个字符之前的所有内容，比如换行符 '\n'，应该使用成员函数 ignore。比如下面的代码，将忽略所有的输入内容，直到遇到换行符 '\n' 为止：

ignore

```
cin.ignore(1000, '\n');
```

上面的参数 1000 指明了可以被忽略的最大字符数。如果在超过了 1000 个字符后

仍然没有遇到换行符，那么将不会继续忽略剩余的字符。当然，上面语句中的 `int` 类型参数和指定的字符参数的值是可以任意设定的，并不一定是 1000 和 `'\n'`。

在第 12 章中我们将会学习文本文件输入流的成员函数 `putback`、`peek` 和 `ignore`，它们并不是 `cin` 所特有的。

15. 假设有下面可正常运行的代码：

```
char c1, c2, c3, c4;
cout << "Enter a line of input:\n";
cin.get(c1);
cin.get(c2);
cin.get(c3);
cin.get(c4);
cout << c1 << c2 << c3 << c4 << "END OF OUTPUT";
```

如果运行时输入如下所示，接下来的输出是什么？

```
Enter a line of input:
a b c d e f g
```

16. 假设有下面可正常运行的代码：

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin.get(next);
while (next != '\n')
{
    if ((count % 2) == 0)
        cout << next;
    count++;
    cin.get(next);
}
```

如果 count 是偶数，则为 True。

如果运行时输入如下所示，接下来的输出是什么？

```
Enter a line of input:
abcdef gh
```

17. 如果自测练习题 16 中的代码运行时有如下的输入，那么接下来的输出是什么？

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
```

18. 假设有下面可正常运行的代码：

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin >> next;
while (next != '\n')
{
    if ((count % 2) == 0)
        cout << next;
```

```
count++;
cin >> next;
}
```

如果运行时输入如下所示，接下来的输出是什么？

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
```

## 字符操作函数

在处理文本数据的时候，经常会遇到将小写字母转换成大写字母，者将大写字母转换成小写字母的需求。在 C++ 中，标准库中的 `toupper` 可以将小写字母转换成大写字母。比如 `toupper('a')` 将返回 'A'。当 `toupper` 函数接收了一个非小写字母的函数时，它将返回这个传入的参数，不会对其做任何改变。所以，`toupper('A')` 仍将返回 'A' 而 `toupper('?')` 仍将返回 '?'。函数 `tolower` 与 `toupper` 类似，区别是 `tolower` 将大写字母转换成小写字母。

函数 `toupper` 和 `tolower` 都定义在头文件为 `<cctype>` 的库中，所以如果想使用这些函数，需要包含下面的代码：

```
#include <cctype>
```

注意，`<cctype>` 库中所有的定义都定义在全局命名空间中，所以不需要使用 `using` 语句。示例 9.3 给出了 `<cctype>` 中常用函数的简单说明。

函数 `isspace` 在它的参数是一个空白字符时返回 `true`。空白字符指的是在屏幕上以空白方式显示的字符。空白符包括空格符、制表符和换行符 '\n'。如果函数的参数不是空白符，那么 `isspace` 函数将返回 `false`。所以 `isspace(' ')` 返回 `true` 而 `isspace('a')` 返回 `false`。

示例 9.3 <cctype> 中的常用函数

函数	描述	样例
<code>toupper</code> ( <code>char_Exp</code> )	返回参数 <code>char_Exp</code> 表示的字母的大写形式（作为整型值返回）	<pre>char c = toupper('a'); cout &lt;&lt; c; Outputs : A</pre>
<code>tolower</code> ( <code>char_Exp</code> )	返回参数 <code>char_Exp</code> 表示的字母的小写形式（作为整型值返回）	<pre>char c = tolower('A'); cout &lt;&lt; c; Outputs : a</pre>
<code>isupper</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是大写字母则返回 <code>true</code> ，否则返回 <code>false</code>	<pre>if (isupper(c))     cout &lt;&lt; "Is uppercase."; else     cout&lt;&lt;"Is not uppercase." ;</pre>
<code>islower</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是小写字母则返回 <code>true</code> ，否则返回 <code>false</code>	<pre>char c = 'a'; if (islower(c))     cout &lt;&lt; c &lt;&lt; " is lowercase."; Outputs : a is lowercase.</pre>

<code>isalpha</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是字母表中的字母则返回 true, 否则返回 false	<pre>char c = '\$'; if (isalpha(c))     cout &lt;&lt; "Is a letter."; else     cout &lt;&lt; "Is not a letter."; <b>Outputs :</b> Is not a letter.</pre>
<code>isdigit</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是 0 ~ 9 之间的数字则返回 true, 否则返回 false	<pre>if (isdigit('3'))     cout &lt;&lt; "It's a digit."; else     cout &lt;&lt; "It's not a digit."; <b>Outputs :</b> It's a digit.</pre>
<code>isalnum</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是字母或者数字则返回 true, 否则返回 false	<pre>if (isalnum('3') &amp;&amp; isalnum('a'))     cout &lt;&lt; "Both alphanumeric."; else     cout &lt;&lt; "One or more are not."; <b>Outputs :</b> Both alphanumeric.</pre>
<code>isspace</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是空白字符 (比如空格符或者换行符) 则返回 true, 否则返回 false	<pre>do {     cin.get(c); } while (! isspace(c));</pre>
<code>ispunct</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是标点符号则返回 true, 否则返回 false	<pre>if (ispunct('?'))     cout &lt;&lt; "Is punctuation."; else     cout &lt;&lt; "Not punctuation.";</pre>
<code>isprint</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是可打印的字符则返回 true, 否则返回 false	
<code>isgraph</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是空白字符以外的可打印字符则返回 true, 否则返回 false	
<code>isctrl</code> ( <code>char_Exp</code> )	如果 <code>char_Exp</code> 表示的字符是控制字符则返回 true, 否则返回 false	

比如下面的代码将读入一个句点结尾的句子, 并将这个句子中所有的空白符使用 '-' 替换, 然后将其输出到屏幕上:

```
char next;
do
{
    cin.get(next);
    if (isspace(next))
        cout << '-';
    else
        cout << next;
} while (next != '.');
```



假设上面程序的输入如下：

```
Ahh do be do.
```

将产生下面的输出：

```
Ahh---do-be-do .
```



### 陷阱：函数 toupper 和 tolower 返回 int 型数值

在 C++ 中所有的字符都可以看作是一个 int 型的数值，每个字符都有一个与之对应的数字。字符被保存在 char 类型的变量中，实际上是对应的数字被保存在了计算机中。在 C++ 中我们可以像使用一个数字一样使用一个 char 类型字符，比如我们可以将 char 类型的值放在 int 型变量中。同样我们也可以将一个 int 类型的值放进 char 型变量中（不过要保证 int 型数值不能太大）。所以，char 类型可以作为字符类型或者小整数数字类型。一般情况下我们并不需要关注这些细节，只需将 char 当作字符类型使用就可以了。但是当我们在使用 <cctype> 库中的一些函数时，char 类型与 int 类型的通用就很重要了。函数 toupper 和 tolower 返回的是 int 型而不是 char 类型，就是说两个函数返回的是字符对应的数字的值而不是字符本身。所以下面的代码返回的实际上是与 'A' 对应的数字的值而不是 'A'：

```
cout << toupper('a');
```

要得到函数 toupper 或者 tolower 返回的值，就指明需要的是 char 类型的值。方法是返回值赋给 char 类型的变量。下面的代码将输出我们常用的真正的字符 'A'：

```
char c = toupper('a');
cout << c;
```

还有一种得到 toupper 和 tolower 函数字符型返回值的方法，就是使用类型转换：

```
cout << static_cast < char >(toupper('a')); ■
```

### 自测练习题

19. 假如有这样一段可以正常运行的代码：

```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    cout << next;
} while ( (! isdigit(next)) && (next != '\n') );
cout << "<END OF OUTPUT";
```

代码运行时的输入如下，那么接下来的输出会是什么？

```
Enter a line of input:
I'll see you at 10:30 AM.
```

20. 写一段可以读入一行文本的代码，在读入的过程中删除这一行文本中的所有大写字

母并输出到屏幕上。

21. 请使用成员函数 `ignore` 重新编写示例 9.2 中的函数 `newLine`。

## 9.3 标准 string 类

我“抓住”我说的和您说的每一句话、每一个单词，赶紧把所有这些句子和单词收进我的文学仓库里去：或许将来用得上！

安东·契诃夫，《海鸥》

在 9.1 节中我们介绍了 C 字符串，这种字符串实质上只是以空字符 ‘\0’ 结尾的一个字符数组。在对 C 字符串的操作过程中要注意到所有字符数组操作需要注意的细节。比如当需要向一个 C 字符串添加新内容的时候，如果数组已经没有足够的存储空间了，那么就需要我们创建新的数组来保存字符串。简而言之，使用 C 字符串要求编程人员了解所有关于 C 字符串在内存中存储的细节。这不仅是一项复杂又艰巨的任务，而且极容易造成代码的错误。ANSI/ISO 标准规定了 C++ 中需要有一个 `string` 类，它的存在使得字符串对于程序员来说可以被看作一种基本数据类型，其具体实现的细节不需要关心，本章就将介绍这种字符串。

### 标准 string 类简介

标准 `string` 类定义在名为 `<string>` 的库中，属于 `std` 命名空间。使用 `string` 类之前需要在代码中包含下面的语句（或者与之等价的其他代码）：

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

`string` 类使得该类的值或者该类型的表达式可以像基本类型的值一样被操作。我们可以使用赋值运算符 “=” 来为一个 `string` 类型的变量赋值，也可以使用 “+” 连接两个字符串。比如 `s1`、`s2` 和 `s3` 是 `string` 类型的对象，且 `s1` 和 `s2` 中已经保存了相应的字符串，那么使用下面的语句可以将 `s2` 的内容连接在 `s1` 后面，形成一个字符串后赋值给 `s3`：

```
s3 = s1 + s2;
```

这时我们并不需要担心 `s3` 没有足够的空间容纳新的字符串，即使新的字符串的长度已经超过了原来 `s3` 的容量，程序也会自动地为 `s3` 增加足够的空间。

在本章的前面我们介绍过，用双引号引起来的字符串属于 C 字符串而不是 `string` 类型的字符串。但是 C++ 提供了自动类型转换机制使引号里的字符串可以自动转换成 `string` 类型的值。因此我们可以直接将引号字符串当作 `string` 变量的值，如下所示：

```
s3 = "Hello Mom!";
```

上面的语句将 `string` 类型的变量 `s3` 的值设定为与 C 字符串 “Hello Mom!” 是具有相同内容的 `string` 类型对象。

+ 连接

## 构造函数

`string` 类有一个默认的构造函数可以将 `string` 类对象的值初始化为空字符串。同时, `string` 类还包含一个带有参数的构造函数, 参数是标准的 C 字符串, 可以是引号字符串。这个带一个参数的构造函数用 C 字符串的内容对 `string` 对象进行初始化。如下所示:

```
string phrase;
string noun("ants");
```

第一行语句声明了一个 `string` 类型变量 `phrase`, 它被初始化为空字符串。第二行也声明了一个 `string` 类型变量 `noun`, 但是它被初始化为内容与 C 字符串 `ants` 相同的字符串。很多程序员喜欢将其称为使用 `ants` 初始化了 `noun`, 但实际上这里包含了自动类型转换, 引号字符串 `ants` 是 C 字符串, 并非 `string` 类型。变量 `noun` 接收的 `string` 类型值与 `ants` 具有同样的字符和顺序, 但并不是以空字符 ‘\0’ 作为结尾的。从理论上讲, 我们无须了解 `string` 类型的值 `noun` 是使用数组存储内容的还是其他数据结构。

还有一种声明 `string` 类型变量并同时使用构造函数进行初始化的方式。下面的两行语句是完全等价的:

```
string noun("ants");
string noun = "ants";
```

示例 9.4 展示了一些 `string` 类使用的细节。值得注意的是, 我们可以像示例 9.4 那样使用运算符 “<<” 输出 `string` 类型的值。

### 示例 9.4 使用 `string` 的程序

```
1 // 标准 string 类使用示例。
2 #include <iostream>
3 #include <string>
4 using namespace std;

5 int main( )
6 {
7     string phrase;           ← 初始化为空字符串。
8     string adjective("fried"), noun("ants"); ← 两种效果相同的初始化。
9     string wish = "Bon appetite!"; ←

10     phrase = "I love" + adjective + " " + noun + "!";
11     cout << phrase << endl
12         << wish << endl;

13     return 0;
14 }
```

#### 示例运行结果

```
I love fried ants!
Bon appetite!
```

考虑示例 9.4 中下面的语句:

将 C 字符串  
常量转为  
string  
类型

```
phrase = "I love " + adjective + " " + noun + "!";
```

这是一种看起来很自然的连接字符串的方法，但实际上 C++ 却做了许多幕后工作。字符串常量 “I love” 并不是 string 类型的对象，像它这样的字符串常量是按照 C 字符串方式存储的（即以空字符作为结束标识符的字符数组）。当 C++ 编译器发现 “I love” 将作为 “+” 的参数时，会去寻找一个与这种参数类型匹配的重载的加号版本。C++ 提供了能接收左边是 C 字符串，右边是 string 类型对象的加号运算符的重载，同时也提供了右边是 C 字符串而左边是 string 类型对象的版本。同样，C++ 也提供了两个参数都是 C 字符串或者都是 string 类型对象的加号的重载版本。

实际上这些加号的重载在 C++ 中并不都是必需的，只要存在一个能接收两个 string 类型对象的版本就足够了。这是因为即使并不存在这么多重载的加号版本，C++ 也可以去寻找能够执行类型转换的构造函数，先将字符串 “I love” 转换成与 “+” 的参数匹配的数据类型。在这里 string 类中包含了一个 C 字符串类型参数的构造函数，但是提供更多的重载是更为有效的方式。

string 类常常被看作是 C 字符串的进化版，但是在实际的 C++ 编程中使用 string 类，还是难以避免要使用到 C 字符串。

### string 类

string 类可以用来表示字符串值。相对于 9.1 节中讨论过的 C 字符串，string 类更加全面、更加灵活。

string 类的定义在一个同样命名为 <string> 的库中，属于 std 命名空间。使用 string 类的程序必须在自己的代码中包含如下的语句（或者与之等价的语句）：

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

或者

```
#include <string>
using std::string;
```

string 类包含一个默认的构造函数，可以将 string 对象的值初始化为空字符串。同时还包含一个接收 C 字符串的构造函数，这个构造函数将会把 string 对象的值初始化为与参数相同的字符串值。比如：

```
string s1, s2("Hello");
```

### string 类的输入和输出

和其他的数据类型一样，我们可以使用 cout 和输出运算符 “<<” 来输出 string 对象的值。这在示例 9.4 中会介绍。但是 string 的输入却有些与众不同。

cin 和输入运算符 “>>” 也适用于 string 类型的对象，但是需要注意的是，输入运算符会忽略最初的空白符，并在读入遇到下一个空白符时停止读入。string 类的输入也具有这个特点，如下面的代码：

```
string s1, s2;
cin >> s1;
```

```
cin >> s2;
```

如果用户输入了下面这样的语句：

```
May the hair on your toes grow long and curly!
```

那么变量 `s1` 得到的字符串值将是“May”，输入语句的第一个空格会被忽略，而变量 `s2` 得到的值将是“the”。可见使用 `cin` 和输入运算符 `>>` 的时候只能一次输入一个单词，并不能输入整行的包含空白符的字符串。可能这正是我们经常需要的，但是有时并非如此。

`getline` 如果需要程序向一个 `string` 类型对象中读入一整行输入语句，我们可以使用函数 `getline`。对 `string` 类型对象使用的 `getline` 函数与前面 9.1 节介绍的 `getline` 成员函数的使用方式有所不同。我们不能使用 `cin.getline`，而是应该将 `cin` 作为 `getline` 函数的第一个参数<sup>1</sup>（这里的 `getline` 函数不是一个成员函数）。

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
cout << line << "END OF OUTPUT\n";
```

如果将上面的程序放在一个可运行的完整程序中，会产生如下的运行结果：

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

如果输入行的前面包含空格，那么这些空格也将会作为字符串的一部分被 `getline` 函数读入到 `string` 类型变量中。这里使用的 `getline` 函数定义在库 `<string>` 中。（在第 12 章中我们可以使用一个文件输入流对象代替 `getline` 函数中的 `cin`，然后就可以从文件输入了。）

不能使用 `cin` 和运算符 `>>` 读入一个空格符。使用 `cin.get` 可以每次读入一个字符，这在 9.2 节中已经介绍过了。函数 `cin.get` 读入的是 `char` 类型而不是 `string` 类型，但是使用 `cin.get` 函数处理 `string` 类型的输入时却很方便。示例 9.5 演示了使用 `getline` 和 `cin.get` 函数进行 `string` 类的输入的方法，其中函数 `newline` 的重要性会在之后的“陷阱：对 `cin` 同时使用 `>>` 和 `getline`”小节中解释。

### string 对象的输入/输出

可以使用 `cout` 和输出运算符 `<<` 来输出 `string` 对象的值，并且可以使用 `cin` 和输入运算符 `>>` 输入 `string` 对象的值，但是使用 `>>` 时只能输入空白符之间的字符串。我们需要使用 `getline` 函数向 `string` 对象中输入一整行的字符串。

#### 示例

```
string greeting("Hello"), response, nextLine;
cout << greeting;
cin >> response;
getline(cin, nextLine);
```

<sup>1</sup> 这有些讽刺，`string` 类型的设计属于先进的面向对象思想，但是提示中使用的 `getline` 仍然是老式的函数调用方法。造成这种现象的原因是一个历史的偶然，`getline` 函数的定义是在 `<iostream>` 库被正式使用之后确定的，所以设计者别无选择，只能留下了 `getline` 这个单独的函数。

**示例 9.5 使用 string 类的程序**


---

```

1  // 演示 getline 和 cin.get 使用方法。
2  #include <iostream>
3  #include <string>
4  using namespace std;

5  void newLine( );
6  int main( )
7  {
8      string firstName, lastName, recordName;
9      string motto = "Your records are our records.";
10     cout << "Enter your first and last name:\n";
11     cin >> firstName >> lastName;
12     newLine( );

13     recordName = lastName + ", " + firstName;
14     cout << "Your name in our records is: ";
15     cout << recordName << endl;
16     cout << "Our motto is\n"
17          << motto << endl;
18     cout << "Please suggest a better (one line) motto:\n";
19     getline(cin, motto);
20     cout << "Our new motto will be:\n";
21     cout << motto << endl;
22     return 0;
23 }
24 // 使用 istream.
25 void newLine( )
26 {
27     char nextchar;
28     do
29     {
30         cin.get(nextchar);
31     } while (nextchar != '\n');
32 }
```

**示例运行结果**

```

Enter your first and last names:
B'Elanna Torres
Your name in our records is: Torres, B'Elanna
Our motto is
Your records are our records.
Please suggest a better (one-line) motto:
Our records go where no records dared to go before.
Our new motto will be:
Our records go where no records dared to go before.
```

---

## 自测练习

22. 假如有这样一段可以正常运行的代码：

```
string s1, s2;
cout << "Enter a line of input:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<END OF OUTPUT";
```

如果代码运行时输入如下，接下来的输出会是什么？

```
Enter a line of input:
A string is a joy forever!
```

23. 假如有这样一段可以正常运行的代码：

```
string s;
cout << "Enter a line of input:\n";
getline(cin, s);
cout << s << "<END OF OUTPUT";
```

如果代码运行时输入如下，接下来的输出会是什么？

```
Enter a line of input:
A string is a joy forever!
```

## 提示：getline 函数的其他版本

到目前为止我们已经了解了下面这些 getline 函数的使用方法：

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
```

这个版本的 getline 函数遇到换行符 ‘\n’ 会停止读入数据。另外，还有一个版本的 getline 函数可以让使用者指定停止读入的标识。举个例子，下面的程序会在读取到第一个问号的时候停止读入：

```
string line;
cout << "Enter some input:\n";
getline(cin, line, '?');
```

从上面的函数调用语句来看，getline 函数是一个 void 类型的函数，但事实上它的返回值是它第一个参数（在上面这段代码中是 cin 对象）的引用。在下面这段示例代码中，读入一行数据输入到 s1 中，然后再读入一个不含空白符的字符串到 s2 中，是合法的：

```
string s1, s2;
getline(cin, s1) >> s2;
```

调用函数 getline(cin, s1) 会返回一个 cin 的引用，所以函数调用之后的操作等价于下面这样的语句：

```
cin >> s2;
```

这样的 getline 函数的用法在 C++ 中并不是必需的，但是会让程序的编写更加方便和灵活。■

### 陷阱：对 cin 同时使用 >> 和 getline

对 cin 同时使用 >> 和 getline 函数的时候一定要多加注意，先看看下面的代码：

```
int n;
string line;
cin >> n;
getline(cin, line);
```

如果上面的程序读入的内容如下所示：

```
42
Hello hitchhiker
```

原本希望变量 `n` 的值被填充为 42，而 `string` 变量 `line` 的值被填充为字符串 “Hello hitchhiker”，但是现实情况是，这段程序确实将变量 `n` 填充为 42 了，但是变量 `line` 的内容却是空字符。发生了什么事情呢？

实际上，`cin >> n` 会忽略输入内容开头的空白符，但是会遇到 42 之后的剩余的行输入，在这个例子中剩余的行输入就是换行符 ‘\n’，它将会作为下一个被读取的字符，就像下面的语句：

```
cin >> n;
```

`getline` 总是会读取当前行剩余的内容，即使该行只剩下一个换行符 ‘\n’。在这个例子中，`getline` 函数遇到换行符之后马上停止输入，所以读入的字符串会是一个空字符串。如果程序出现了奇怪的忽略了输入数据的现象，那么就去看看是否使用了这种混合的输入方式吧。示例 9.5 中出现的 `newLine` 函数或者 `iostream` 库中的 `ignore` 函数可以帮助我们解决这个问题：

```
cin.ignore(1000, '\n');
```

上面的语句可以读入并且丢弃剩余的行内容，直到遇到换行符为止（如果一直没有遇到换行符，那么将读入 1000 个字符）。

对 `cin` 同时使用 >> 和 `getline` 还可能导致其他一些令人困惑的问题，而且如果在用一个 C++ 编译器编译通过的代码中出现了这种混合使用，换用另一种编译器编译可能会出现。在所有的方法都无法满足需求，或者我们想要增加输入处理的灵活性时，可以使用 `cin.get` 将输入字符一个个地读入。

上面所说的问题在本章讨论过的所有版本的 `getline` 函数的使用中都有可能出现。■

### 使用 string 类处理字符串

9.1 节中介绍了使用 C 字符串做各种字符串处理操作的情况，`string` 类提供的对字符串处理的操作与 C 字符串相比，只多不少（可以用来处理字符串的 `string` 类成员函数和其他相关函数有 100 多个）。

可以像访问数组中元素一样访问 `string` 对象中的字符，所以 `string` 对象除了具有字符数组的优点外，还有一些其他的长处，比如容量是可以动态增长的。

假设 `lastName` 是一个 `string` 类对象，那么 `lastName[i]` 就是 `lastName` 表



示的字符串的第  $i$  个字符。示例 9.6 将演示这种数组运算符的使用。

### 与 string 类对象配合使用的 getline 函数

能与 string 类对象一起使用的 getline 函数有如下两个版本：

```
istream& getline(istream& ins, string& strVar, char delimiter);
```

和

```
istream& getline(istream& ins, string& strVar);
```

第一个版本的 getline 从输入流 istream 对象中读入字符串，然后将其保存在 string 类变量 strVar 中，当遇到 delimiter 指定的结束符时停止读入，结束符本身并不被读入。第二个版本的 getline 使用换行符作为固定的结束符，除此之外与第一个版本的 getline 完全一样。

这里使用的 getline 函数返回的是它们的第一个参数（在本章中一直是 cin），但大多数情况下它们会被当作 void 类型的函数使用。

示例 9.6 中使用了成员函数 length。每个 string 类对象都有这个名为 length 的成员函数，这个函数没有参数，返回值是 string 类对象中字符串的长度。string 类对象不仅可以像一个字符数组一样被使用，而且通过调用它的成员函数 length 就可以知道这个数组中有多少个元素。

### 示例 9.6 像操作数组一样操作 string 类对象

```
1 // 演示将一个 string 类对象当作数组使用的程序。
2 #include <iostream>
3 #include <string>
4 using namespace std;

5 int main( )
6 {
7     string firstName, lastName;

8     cout << "Enter your first and last name:\n";
9     cin >> firstName >> lastName;

10    cout << "Your last name is spelled:\n";
11    int i;
12    for (i = 0; i < lastName.length( ); i++)
13    {
14        cout << lastName[i] << " ";
15        lastName[i] = '-';
16    }
17    cout << endl;
18    for (i = 0; i < lastName.length( ); i++)
19        cout << lastName[i] << " "; // 在每一个字符后加一个 "-"。
20    cout << endl;

21    cout << "Good day " << firstName << endl;
22    return 0;
23 }
```

## 示例运行结果

```

Enter your first and last names:
John Crichton
Your last name is spelled:
C r i c h t o n
-----
Good day John

```

对 string 类对象使用方括号数组运算符不会检查索引的合法性。使用非法索引（即超出了 string 类对象所能表示的字符串的长度上限）的后果是不可预知的。程序可能会出现一些难以捉摸的现象，可是却不会给出使用了非法索引的错误提示。不过 string 类有一个 at 成员函数可以对索引的合法性进行检查。成员函数 at 的功能与数组运算符几乎一样，但有两点不同：一是需要调用函数，即使用 a.at(i) 来代替 a[i]；二是 at 函数将会检查索引值的合法性。如果 a.at(i) 中的参数 i 是一个非法索引，那么程序将会在运行时出现错误信息。下面的代码中访问的位置超过了字符串的范围，尽管访问的是一个并不存在的索引位置，但是程序不会产生任何错误信息：

```

string str("Mary");
cout << str[6] << endl;

```

下面的代码会使程序非正常中断，程序员会知道某个地方出现了错误：

```

string str("Mary");
cout << str.at(6) << endl;

```

但是需要注意一点，某些编译器即使使用了 a.at[i]，也未必能给出足够的提示出现了非法索引的错误信息。

可以将索引变量（如 str[i]）赋值为一个 char 类型的值的方式来改变字符串中单个字符的值。由于成员函数 at 返回的是引用，这种方式对 at 函数同样适用。如果要将 string 类对象 str 所表示的字符串中的第三个字符修改为 'X'，下面任意一种方法都是可以的：

```

str.at(2)='X';

```

或者

```

str[2]='X';

```

和普通的数组一样，string 类对象的起始索引也是 0，所以第三个字符的索引是 2。

示例 9.7 给出了类 string 的部分成员函数。

相比于 C 字符串，使用 string 类做字符串的处理有很多方面的优势。比如使用 “==” 比较两个 string 类对象能够得到真正的字符串比较结果，也就是当两个字符串中的字符都相同且顺序也相同时返回 true，否则返回 false。类似的其他比较运算符 “<”、“>”、“<=” 以及 “>=” 使用字典顺序（字典顺序指的是字符顺序，具体规则在附录 C 中的 ASCII 码字符集中给出。如果字符串完全由大小写字母组成，那么字典顺序就是通常意义上的字母顺序）比较字符串。

示例 9.7 标准类 `string` 的成员函数

例子	注解
<b>构造函数</b>	
<code>string str;</code>	默认构造函数, 创建包含空字符串的对象 <code>str</code>
<code>string str("string");</code>	创建包含字符串 "string" 的对象 <code>str</code>
<code>string str(aString);</code>	将 <code>aString</code> 的内容拷贝到 <code>str</code> , <code>aString</code> 也是 <code>string</code> 类对象
<b>元素获取</b>	
<code>str[i]</code>	返回 <code>str</code> 对象中第 <code>i</code> 个字符的引用
<code>str.at(i)</code>	返回 <code>str</code> 对象中第 <code>i</code> 个字符的引用
<code>str.substr(position, length)</code>	返回起始位置为 <code>position</code> , 长度为 <code>length</code> 的子字符串
<b>赋值/修改</b>	
<code>str1 = str2;</code>	为 <code>str1</code> 分配新的存储空间, 将 <code>str1</code> 的内容和大小设置为与 <code>str2</code> 一样, 并释放 <code>str1</code> 原来的存储空间
<code>str1 += str2;</code>	将 <code>str2</code> 的内容追加到 <code>str1</code> 末尾, <code>str1</code> 的长度发生变化
<code>str.empty()</code>	字符串为空字符串则返回 <code>true</code> , 否则返回 <code>false</code>
<code>str1 + str2</code>	返回一个新的字符串, 此字符串由 <code>str2</code> 接在 <code>str1</code> 的末尾组成, 长度会发生变化
<code>str.insert(pos, str2)</code>	将 <code>str2</code> 的字符串插入到 <code>str</code> 的 <code>pos</code> 位置
<code>str.remove(pos, length)</code>	移除 <code>str</code> 的从 <code>pos</code> 开始, 长度为 <code>length</code> 的子字符串
<b>比较</b>	
<code>str1 == str2   str1 != str2</code>	比较字符串是否相同, 返回布尔值
<code>str1 &lt; str2   str1 &gt; str2</code>	根据字典顺序比较的四种比较方式
<code>str1 &lt;= str2   str1 &gt;= str2</code>	
<code>str.find(str1)</code>	返回在 <code>str</code> 中第一次找到 <code>str1</code> 的位置
<code>str.find(str1, pos)</code>	返回在 <code>str</code> 中第一次找到 <code>str1</code> 的位置, 查找的起始位置由 <code>pos</code> 指定
<code>str.find_first_of(str1, pos)</code>	返回 <code>str</code> 中第一次出现 <code>str1</code> 中任意一个字符的位置, 查找起始位置由参数 <code>pos</code> 指定
<code>str.find_first_not_of(str1, pos)</code>	返回 <code>str</code> 中第一次出现不属于 <code>str1</code> 中任意一个字符的位置, 查找起始位置由参数 <code>pos</code> 指定

**= 和 == 对 `string` 对象和 C 字符串来说是不同的**

对于运算符 `=`、`==`、`!=`、`<`、`>`、`<=` 和 `>=` 来说, 作用在标准 C++ 类 `string` 的对象之间时, 得到的比较结果就是我们所期望的比较规则所得到的结果。这与 9.1 节中介绍的作用在 C 字符串之间是不同的。

### 示例：回文检测

回文是指正序反序的内容都一样的字符串。示例 9.8 中的程序可以用来检测输入的字符串是不是回文。这里的回文检测将忽略字符串中左右的空格和标点，并且对大小写是不敏感的。下面是几个回文的例子：

```
Able was I 'ere I saw Elba.
I Love Me, Vol. I.
Madam, I'm Adam.
A man, a plan, a canal, Panama.
Rats live on no evil star.
radar
deed
mom
racecar
```

函数 `removePunct` 中出现了 `string` 类的成员函数 `substr` 和 `find`。成员函数 `substr` 根据指定的起始位置和长度从调用自己的对象中获取子字符串。`removePunct` 函数的前三行声明了函数中需要的变量。`for` 循环将检查参数 `s` 中的每一个字符，看是否能在 `punct` 字符串中找到一样的字符。所以每次循环都会从字符串 `s` 中获取一个长度为 1 的子字符串，然后使用 `find` 在 `punct` 中寻找该字符串。如果子字符并不存在于 `punct` 中，则说明此子字符串不是标点符号，它将会被追加到作为返回值的字符串 `noPunct` 后面。

### 示例 9.8 回文检测程序

```
1 // 回文检测程序。
2 #include <iostream>
3 #include <string>
4 #include <cctype>
5 using namespace std;
6 void swap(char & v1, char & v2);
7 // 将 v1 和 v2 的值互换。
8 string reverse(const string& s);
9 // 返回字符串 s 的反向字符串的拷贝。

10 string removePunct(const string& s, const string& punct);
11 // 返回字符串 s 的拷贝，
12 // punct 中的所有字符将会被移除。

13 string makeLower (const string& s);
14 // 返回字符串 s 的拷贝。
15 // 所有大写字符都替换为小写字符。

16 bool isPal(const string& s);
17 // 字符串 s 是回文则返回 true，否则返回 false。
18 int main( )
19 {
20     string str;
```

```

21     cout << "Enter a candidate for palindrome test\n"
22         << "followed by pressing Return.\n";
23     getline(cin, str);

24     if (isPal(str))
25         cout << "\"" << str + "\" is a palindrome.";

26     else
27         cout << "\"" << str + "\" is not a palindrome.";
28     cout << endl;

29     return 0;
30 }

31
32 void swap(char & v1, char & v2)
33 {
34     char temp = v1;
35     v1 = v2;
36     v2 = temp;
37 }

38 string reverse(const string& s)
39 {
40     int start = 0;
41     int end = s.length( );
42     string temp(s);

43     while (start < end)
44     {
45         end--;
46         swap(temp[start], temp[end]);
47         start++;
48     }

49     return temp;
50 }

51 // 使用 <cctype> 和 <string>。
52 string makeLower(const string& s)
53 {
54     string temp(s);
55     for (int i = 0; i < s.length( ); i++)
56         temp[i] = tolower(s[i]);
57
58     return temp;
59 }
60
61 string removePunct(const string& s, const string& punct)
62 {
63     string noPunct; // 初始化为空字符串。
64     int sLength = s.length( );
65     int punctLength = punct.length( );
66     for (int i = 0; i < sLength; i++)
67     {

```

```

68     string achar = s.substr(i,1); // 只有一个字符的字符串。
69     int location = punct.find(achar, 0);
70     // 在 punct 中寻找源字符串中连续字符的位置。

71     if (location < 0 || location >= punctLength)
72         noPunct = noPunct + achar; // achar 不是标点符号，所以保留。
73 }
74
75 return noPunct;
76 }

77 // 调用函数 makeLower、removePunct。
78 bool isPal(const string& s)
79 {
80     string punct(",:;?!'\ " ); // 包含一个空格。
81     string str(s);
82     str = makeLower(str);
83     string lowerStr = removePunct(str, punct);
84
85     return (lowerStr == reverse(lowerStr));
86 }

```

#### 示例运行结果

```

Enter a candidate for palindrome test
followed by pressing Return.
Madam, I'm Adam.
"Madam, I'm Adam." is a palindrome.

Enter a candidate for palindrome test
followed by pressing Return.
Radar
"Radar" is a palindrome.

Enter a candidate for palindrome test
followed by pressing Return.
Am I a palindrome?
"Am I a palindrome?" is not a palindrome.

```

#### 24. 有如下的代码：

```

string s1, s2("Hello");
cout << "Enter a line of input:\n";
cin >> s1;
if (s1 == s2)
    cout << "Equal\n";
else
    cout << "Not equal\n";

```

如果在运行时有如下的输入，接下来的输出是什么？

```

Enter a line of input:
Hello friend!

```

## 25. 下面代码的运行结果是什么？

```
string s1, s2("Hello");
s1 = s2;
s2[0] = 'J';
cout << s1 << " " << s2;
```

**string类对象和C字符串的转换**

前面的内容已经介绍过，当把C字符串存储在string类型的变量中时，C++会自动做好类型转换。比如下面的代码，是可以正常工作的：

```
char aCString[] = "This is my C-string.";
string stringVariable;
stringVariable = aCString;
```

但是下面这样的代码是无法成功编译的：

```
aCString = stringVariable; // 非法
```

同样下面的代码也是非法的：

```
strcpy(aCString, stringVariable); // 非法
```

string类对象不能作为函数strcpy的第二个参数，并不存在string类对象转换为C字符串的自动转换，这是需要我们自己解决的问题。

要从已存在的string类对象中获取相应的C字符串，必须进行显式转换。这个转换可以通过string类的成员函数c\_str()完成。所以从string类对象向C字符串赋值的正确方法应该是：

```
strcpy(aCString, stringVariable.c_str()); // 合法
```

要注意必须使用strcpy函数进行拷贝。成员函数c\_str()以C字符串的形式返回自己对象包含的字符串值。本章前面的内容中提到过赋值运算符无法正常作用在C字符串上，所以下面的语句看起来似乎正确，但实际上是非法的：

```
aCString = stringVariable.c_str(); // 非法
```

**本章小结**

- C字符串变量实质上是一个字符数组，只是在使用时略有不同。C字符串使用空白符‘\0’作为字符串结束标识。
- C字符串变量应该被当作一个数组来操作，而并非一个简单的变量（比如表示数字或者单个字符的变量）。实际上不能使用赋值运算符“=”对一个C字符串变量赋值，同样，使用“==”比较两个C字符串也是不合法的。要实现这样的功能，必须使用特定的C字符串函数。
- <cctype>库中定义了许多好用的字符操作函数。

- `cin.get` 可以用来读取一个输入的字符，也包括空格。成员函数 `cin.get` 会从输入中读取一个字符，无论这个字符是什么。
- 多个版本的 `getline` 函数可以处理整行字符的输入。
- ANSI/ISO 标准的 `<string>` 库中定义了一个完整的类 `string`，这个类可以用来表示字符串。
- 使用 `string` 类的对象对字符串进行操作比 C 字符串的方式好很多。比如赋值和比较运算符（“=” 和 “==”）在用于 `string` 类对象时能够得到大家预期的结果。

## 自测练习题答案

1. 下面这两种声明是等价的，但与其他三种不同：

```
char stringVar[10] = "Hello";
char stringVar[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

下面这两种也是等价的：

```
char stringVar[6] = "Hello";
char stringVar[] = "Hello";
```

下面的这种声明与其他几种都不是等价的：

```
char stringVar[10] = {'H', 'e', 'l', 'l', 'o'};
```

2. "DoBeDo to you"
3. 根据题目中的描述，变量 `stringVar` 只能容纳六个字符（包括一个空字符 `'\0'`）。而函数 `strcat` 无法检查是否有足够的空间容纳 `stringVar` 中的字符，虽然需要的内存空间比实际分配给 `stringVar` 的空间要大，但是函数 `strcat` 还是会将额外的字符串 "and Good-bye" 写到内存中。这会使内存中不应被改动的内容被修改，这可能造成很严重的后果。
4. 如果 `strlen` 不是预定义好的函数，则可以使用下面的定义：

```
int strlen(const char str[])
// 前提：str 是一个以空字符 '\0' 结尾的字符串。
// 返回值：返回 str 中字符的个数（不包括 '\0'）。
{
    int index = 0;
    while (str[index] != '\0')
        index++;
    return index;
}
```

5. 最多能保存五个字符，因为第六个位置上是空字符 `'\0'`。
6. a. 1  
b. 1  
c. 5 (including the `'\0'`)  
d. 2 (including the `'\0'`)



e. 6 (including the '\0')

7. 两种方式有明显的不同，第一种会在字符串的末尾添加空字符 '\0'，第二种方式则不会添加。

8. 代码如下：

```
int index = 0;
while (ourString[index] != '\0')
{
    ourString[index] = "X";
    index++;
}
```

9. a. 如果 C 字符串变量中没有用来标识结尾的空字符 '\0'，循环就会访问到 C 字符串自身内存空间之外的内存，可能会破坏系统的内存内容。修改 while 循环的结束条件可以保护内存不被非法改写：

```
b. while (ourString[index] != '\0' && index < SIZE)
```

```
10. #include <cstring>
// 使用 strcpy 时所需要的
...
strcpy(aString, "Hello");
```

11. I did it my way!

12. 字符串 "good, I hope." 长度超过了变量 aString 的容量，所以有些 aString 自身以外的内存被改写。

13. 完整的运行对话如下：

```
Enter some input:
The
time is now.
The-time<END OF OUTPUT
```

14. 完整的运行对话如下：

```
Enter a line of input:
May the hair on your toes grow long and curly.
May t<END OF OUTPUT
```

15. 完整的运行对话如下：

```
Enter a line of input:
a b c d e f g
a b END OF OUTPUT
```

16. 完整的运行对话如下：

```
Enter a line of input:
abcdef gh
ace h
```

注意输出的是被拆分成一个个单个字符的输入，空白符也被当作一个字符。

17. 完整的运行对话如下：

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
```

```
01234567891 1
```

需要注意的是，输入的 10 只有 '1' 输出了，这是因为 `cin.get` 函数是按字符逐个读入的，而不是按照数字读入，所以 10 作为两个字符 '1' 和 '0' 被分为两次输入。由于一次只能读入一个字符，所以紧跟在 '1' 后面输入的 '0' 不会被输出，而下一个被输入的字符，也就是空格，将被输出。类似地，11 中只有第一个 '1' 被输出。如果仍然没有理解，可以将输入的字符都写在纸上的表格中，空格符用方块表示，然后每隔一个字符就划掉一个字符，最终剩下的就是上面的运行结果。

18. 代码中有一个无限循环，将一直运行到用户停止输入。布尔表达式 `next != '\n'` 的值一直是 `true`，因为 `next` 的内容总是被下面的语句修改：

```
cin >> next;
```

这里总是跳过 `newline` 字符 '\n' 和所有空格。如果没有其他的额外输入，程序的输出应该如下所示：

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
0246811
```

这段代码会间隔地输出下一个非空白字符。运行结果中的两个 '1' 是输入 10 的第一个字符和输入 11 的第一个字符。

19. 完整的运行对话如下：

```
Enter a line of input:
I'll see you at 10:30 AM.
I'll see you at 1<END OF OUTPUT
```

20. `if` 语句中的条件表达式是 `!isupper(next)` 而不是 `islower(next)`，这是因为如果 `next` 中含有一些非字母的字符（例如空格或者逗号等），`islower(next)` 将会返回 `false`。

```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    if (!isupper(next))
        cout << next;
}while (next != '\n' );
```

21. // 使用了 `iostream`。
- ```
void newLine()
{
```

```
    cin.ignore(10000, '\n' );
}
```

当然，这段程序只能应付长度小于 10 000 的一行，不过任何长度超过 10 000 的文本行都可能引起其他的一些问题。

22. `A*string<END OF OUTPUT`

23. A string is a joy forever!<END OF OUTPUT

24. 完整的运行对话如下：

```
Enter a line of input:
Hello friend!
Equal
```

25. Hello Jello

## 编程练习

1. 编写一个可以读入 100 个字符的程序，纠正其中的大小写和不恰当的空格，然后将结果输出。具体而言，句子中所有的两个以上的连续空格将被压缩为一个空格，句子应该以大写字母开头但是中间不应该再出现大写字母。不要担心应该被大写的名字，它们的开头字母小写也是可以接受的。句子中的换行符要当作空格处理，一个换行符和接下来的空格应该被压缩为一个空格。假设句子是以句点结尾的，而句子的其他位置都不再包含句点。举个例子，下面的输入：

```
the   Answer to life, the Universe, and everything
IS 42.
```

应该生成下面的输出：

```
The answer to life, the universe, and everything is 42.
```

2. 编写一个程序统计一行文本中的单词个数和每个字母出现的次数。这里将单词定义为以任意字母组成的字符串，这些字符串以空格、句点、逗号或者一行的开头或者一行的结尾作为终结。假设所有的输入都是字母、空格、逗号或者句点，统计某个字母在文本中出现的次数时忽略字母的大小写，输出顺序按照字母表的顺序，只需要输出在文本中出现过的字母。

举个例子，如果有如下的输入：

```
I say Hi.
```

程序应该产生下面的输出：

```
3 words
1 a
1 h
2 i
1 s
1 y
```

3. 编写一个程序读入一个人的名字，输入时名字的格式为：名，中间名或起始字母，姓。程序的输入格式如下：

```
Last_Name , First_Name, Middle_Initial.
```

假如有下面的输入：

```
Mary Average User
```

应该产生下面的输出：

```
User, Mary A.
```

而输入

```
Mary A. User
```

应该产生下面的输出：

```
User, Mary A.
```

你的程序应该在中间名的首字母后面加上一个句点，同时程序还应该允许用户不输入中间名或者首字母。比如下面的输入：

```
Mary User
```

应该产生下面的输出：

```
User, Mary
```

如果你想使用 C 字符串完成代码，那我们可以假设一个名字最多包含 20 个字符，当然你也可以用 `string` 类完成。（提示：你可以使用三个字符串变量记录输入，并不需要一个长长的字符串变量。你会发现这样做可以避免使用 `getline` 函数。）

4. 编写程序将一行文本中所有长度为 4 的单词替换为“love”。比如输入下面的字符串：

```
I hate you, you dodo!
```

应该产生下面的输出：

```
I love you, you love!
```

当然输出的句子并没有合理的意义，比如下面的输入：

```
John will run home.
```

应该产生下面的输出：

```
Love love run love.
```

除每个单词的首字母外，其他字母都不需要检查是否是大写。这里的单词定义为由字母表中的字符组成的字符串，它们之间通过空格或者换行符分隔。你的程序应该可以一直正常工作，直到用户自己需要退出为止。

5. 编写一个程序鼓励用户使用中性化的语言，即根据用户输入的内容给出对应的中性化的替换语言。程序需要先读入一个句子并将句子保存在一个 `string` 变量中，然后将句子里所有的阳性代词替换为中性代词。比如将“he”替换为“she or he”。下面的输入：

```
See an adviser, talk to him, and listen to him.
```

将会产生下面的修改过的句子：

See an adviser, talk to her or him, and listen to her or him.

注意要保证句子的首字母大写，代词“his”可以被替换为“her(s)”，不需要区分“her”和“hers”。程序允许用户一直输入句子，直到句子中出现 she 或者 he。程序不能将其他单词中出现的 he 替换掉，比如“here”。这里的单词定义为字母表中的字符组成的字符串，它们之间通过空格或者换行符分隔，假设句子最多包含 100 个字符。

6. 假设有一批出售的 CD，其中包含公开发行过的音乐的 .jpeg 和 .gif 图像，CD 的文件名由一行文字代表，包含曲名和作者名。文件名由曲名开头，然后是一些空格和一个短横线符号“-”，紧接着是一个或者多个空格，最后是作者名。作者名可能只包含姓，或者只包含名，或者既有名又有姓，也可能包含名、姓、中间名三个字段。还有一些曲子并没有具体的作者（no author listed），这里不能将“no author listed”当作一个作者名重新划分。下面是一些曲子名称和作者的例子：

1. Adagio "MoonLight" Sonata- Ludwig Van Beethoven

2. An Alexis- F.H. Hummel and J.N. Hummel

3. A La Bien Aimee- Ben Schutt

4. At Sunset- E. MacDowell

5. Angelus- J. Massenet

6. Anitra's Dance- Edward Grieg

7. Ase's Death- Edward Grieg

8. Au Matin- Benj.- Godard

...

37. The Dying Poet- L. Gottschalk

38. Dead March- G.F. Handel

39. Do They Think of Me At Home- Chas. W. Glover

40. The Dearest Spot- W.T. Wrigton

1. Evening- L. Van Beethoven

2. Embarrassment- Franz Abt

3. Erin is my Home- no author listed

4. Ellen Bayne- Stephen C. Foster

...

9. Alla Mazurka- A. Nemerowsky

...

1. The Dying Volunteer- A.E. Muse
2. Dolly Day- Stephen C. Foster
3. Dolcy Jones- Stephen C. Foster
4. Dickory, Dickory, Dock- no author listed
5. The Dear Little Shamrock- no author listed
6. Dutch Warbler- no author listed

...

我们需要将所有的曲目按照字母表顺序排序，这可以分为下面的几个步骤：

- a. 将所有行最前面的数字编号和句点、空格都去掉，这样曲目名称的首单词就是每行的第一个单词。
- b. 将所有连续出现的空格压缩成一个空格。
- c. 一些曲目名称中可能包含多个 ‘-’，比如：

20. Ba- Be- Bi- Bo- Bu - no author listed

要将每行中除最后一个 ‘-’ 以外的所有该字符替换为空格。

- d. 曲子名称的最后一个单词和“-”之间可能没有空格，如果没有则增加一个空格。
  - e. 按照字母表顺序排序时不应该考虑曲名中开头的“A”、“An”或者“The”。短横线“-”前面出现的这种单词应该被忽略，曲名最后的逗号是无用的，可以在作者名被移动到前面的时候处理，但是代码上的差别比较大。
  - f. 将作者名移动到一行的开头，然后是短横线，之后是曲名。
  - g. 将所有姓氏首字母、姓氏、中间名都移动到名字后面。如果作者一栏的内容是‘no author listed’，则不做上面这个操作。
  - h. 将作者的名按照字母表排序，可以忽略所有重复的名字，比如 CPE Bach 和 JS Bach，但是按中间名再进行排序是一个更高级的功能。具体算法你可以尝试插入排序、选择排序、冒泡排序等。
  - i. 如果你还没有将曲名开头的“A”、“An”或者“The”移动到曲名的后面，则可以使用复合名称排序。
  - j. 保存你程序的设计和实现，今后可以尝试使用 STL 中的 vector 容器来完成这些功能。
7. Caps Lock 键被按下之后，按寻常习惯输入的话，会出现单词首字母小写而其他字母大写的情况。比如这样的单词：lIKE tHIS。编写一个程序扫描用户的输入并输出用户输入的具有这种特征的单词。
- 程序应该可以扫描任意多个字符串，并找到每个字符串中可能存在的所有锁定

了大写的单词，直到用户输入一个空白字符串。

8. 编写一个程序将用户的输入转化为 pig latin (英文字母的顺序被打乱后组成的文字)。假设输入的句子不包含标点符号，转化的规则如下：
  - a. 将以辅音字母开头的单词的开头字母放到词尾并添加“ay”，比如 ball 会变成 allbay，button 会变成 uttonbay。
  - b. 在以元音字母开头的单词的词尾加上“way”。这样 all 就会变成 allway，one 会变成 oneway。
9. 编写一个能比较两个 C 字符串的函数，字符串相等则返回 true，否则返回 false。比较函数应该忽略字母的大小写、标点符号及空白字符。写好后用各种字符串测试你的函数。
10. 编写一个益智问答游戏。创建一个 Trivia 类来表示一个问答题，类中应该有一个 string 保存问题内容，一个 string 保存问题的答案，一个整型表示问题的价值（问题越难，价值越高）。为类编写合适的构造函数和取值函数。在 main 函数中创建一个 Trivia 对象的数组或者 vector，根据你的选择硬编码至少五个问答题。程序将为玩家展示所有的问题，接收玩家输入的问题答案并将答案与标准答案做对比，如果答案正确则为此玩家累计加上回答正确的问题的分数，最后统计玩家获得的总分。
11. 编写一个函数检测两个字符串是否恰好是对方的逆序串。函数应该忽略字母的大小写、标点符号及空白字符。一个字符串的逆序串就是把所有字符顺序反转后得到的字符串，举个例子，“Eleven plus two”是“Twelve plus one”的逆序串，各个单词分别包含一个 v、三个 e、两个 l 等。数写好后用各种逆序的和非逆序的字符串测试你的函数，可以用 string 类，也可以用 C 字符串来实现。
12. 编写一个函数将字符串转化成整数。比如，输入一个字符串“1234”会返回一个整型 1234。如果你做一些调查，就会发现一个叫作 atoi 的函数和一个叫作 stringstream 的类可以为你完成这样的转换。但是这里我们需要你自己编写一个函数，而且要编写一个程序来测试它。
13. 有一种叫作拆字拼字的游戏，原理是用一个单词中的字母组合排列形成新单词。举个例子，对于单词 SWIMMING，我们可以组成新单词 SWIM，WIN，WING，SING，MIMING 等。写一个程序，允许用户输入一个单词，将这个单词所能组成的所有包含在 words.txt 文件中的新单词输出。一个简单的算法是比较每一个单词的字母统计情况。用一个数组为输入的单词中的每一个字母做计数（例如一个 S、一个 W、两个 I、两个 M 等）。对 words.txt 文件中的一个单词也构建一个这样的数组，比较这两个数组就可以得知后者是否可以由前者的字母组成。



# 指针和动态数组

# 10

## 10.1 指针 348

指针变量 349

内存管理基础 355

陷阱：悬空指针 357

动态变量和自动变量 358

提示：定义指针类型 358

陷阱：指针用作传值参数 360

指针的应用 361

## 10.2 动态数组 362

数组变量和指针变量 362

创建并使用动态数组 363

示例：一个返回数组的函数 366

指针运算 368

多维动态数组 368

## 10.3 类、指针和动态数组 371

运算符> 371

this指针 371

重载赋值运算符 372

示例：部分填充数组的类 378

析构函数 380

拷贝构造函数 381



# 第10章 指针和动态数组

记忆是一切理性行为的前提条件。

布莱士·帕斯卡，《思想录》

## 概述

指针是一种能够让程序员更直接地控制计算机内存的数据结构。本章将介绍指针在数组中的应用，并且介绍一种全新的数组类型：动态数组。动态分配数组（简称动态数组）的长度将在程序运行时动态决定，而不是必须在编写代码时指定。

在学习 10.1 节和 10.2 节关于指针和动态数组的知识之前，必须先阅读第 1 ~ 6 章的内容（其中关于向量的知识可以忽略），但无须第 7 ~ 9 章的知识。我们甚至可以在学完第 1 ~ 5 章后，直接学习 10.1 节和 10.2 节，但必须略过其中少数关于类的知识。

10.3 节介绍在类中使用指针和动态分配数据（如动态数组等）时的一些工具。在学习 10.3 节前，应该看完第 1 ~ 8 章的内容，但可以忽略其中关于向量的内容。

对于本章、第 11 章、第 12 章及第 13 章，这 4 章的学习顺序可以随意安排。如果在学习本章前没有学习 11 章关于命名空间的知识，那么就应该先熟悉一下第 11 章中关于“命名空间”的内容。

## 10.1 指针

在漫无目的中寻找指引。

莎士比亚，《哈姆雷特》

### 指针

指针本质上是变量的内存地址。回忆第 5 章中的内容，计算机内存被划分成基本内存单元（称为字节），变量就被保存在由若干相邻的内存基本单元组成的一个序列中。有时操作系统会将这些内存的地址当作变量的标识使用，比如一个变量占据了三个单元的内存单元，那么就可以用第一个内存单元的地址代表这个变量。再比如，当一个变量作为引用传递的实参时，传递给调用函数的就是第一个内存单元的地址而不是变量的名字。通过给出变量在内存中的起始地址来表示一个变量时，这个地址就被称为指针，可以理解为该地址是变量的标识。地址可以代表一个变量，是因为地址通过指明变量存储在什么地方来唯一标识该变量，而不是靠指明变量的名字是什么。

其实之前我们已经在很多地方使用过指针了。正如上面描述的那样，当一个变量作为向函数传递的引用实参时，实际上传递给该函数的是实参的指针。同样在第 5 章中曾经提到过，将一个数组作为参数传递给函数（或者其他等价的情况）时，实际上传递的也是数组第一个元素的指针（那时我们还将这个位置称为内存地址，但实际上和指针是一码事）。这是常见的两种指针用法，但都是由 C++ 自己负责处理的。本章将介绍如何在代码中操作指针，而不是依赖于 C++ 本身。

## 指针变量

### 声明指针 变量

指针可以被保存在变量中。但是尽管指针是一个内存地址而内存地址只是一个序号，也不能将指针存储在 `int` 型或者 `double` 型变量中。用来存储指针的变量必须声明为指针类型。比如下面的语句声明了一个指针变量 `p`，该变量就可以用来存储一个指向 `double` 型变量的指针：

```
double *p;
```

上面声明的变量 `p` 只能用来存储 `double` 型变量的指针，而不能存储其他类型变量的指针，比如 `int` 型或者 `char` 型。每种变量类型都需要不同类型的指针。<sup>1</sup>

要声明一个用来存储某类型指针的变量，通常可以按照声明该类型变量的方式声明这个指针变量，但必须在变量名前面加星号。比如下面的语句先声明了两个可以存储 `int` 型指针的变量 `p1` 和 `p2`，然后又声明了两个普通的 `int` 型变量 `v1` 和 `v2`：

```
int *p1, *p2, v1, v2;
```

所有指针变量的声明前面都必须加星号。如果上面的声明中第二个星号被忽略，那么 `p2` 将不是一个指针变量，而是一个普通的 `int` 型变量。

### 指针变量声明

当声明一个指向某一类型变量的指针时，除了在变量名前写上星号以外，其声明方式和声明该类型的变量类似。

#### 语法

```
Type_Name *Variable_Name1, *Variable_Name2,...;
```

#### 示例

```
double *pointer1, *pointer2;
```

### 地址和数字

指针是一个地址，而地址一定是一个整数，但是指针并不是一个整数。看起来这种说法并不合理，其实是一个抽象的概念问题。C++ 把指针当作一个地址使用，而不是作为一个数字使用。指针值的数据类型不是 `int` 类型或者其他数字类型，一般情况下不能将指针存储在 `int` 类型的变量中。如果这样做了，绝大多数 C++ 编译器会给出错误提示或者警告信息。自然，指针也不能进行一般的数学运算。（在本章后面将会看到，指针可以进行特殊的加法和减法操作，但这些操作的实质都不是普通的整数加法和减法。）

在讨论指针和指针变量的时候，我们常常将其称为指向，而不是编址。当一个指

<sup>1</sup> 在这里将一种类型的指针存放在另一种类型的指针变量中也是可能的，但这是一种非常不好的行为。

针变量（如 p1）包含一个变量（如 v1）的地址时，这个指针常常被称为指向变量 v1 的指针或者变量 v1 的指针。

## & 运算符

指针变量（如前面声明的 p1 和 p2）能够存储变量（如前面声明的 v1 和 v2）的指针。可以使用运算符 & 来获得变量的地址，并将这个地址值赋值给一个指针变量。比如下面的语句会将变量 v1 的地址赋值给指针变量 p1：

```
p1 = &v1;
```

## \* 运算符 解引用 运算符

现在我们已经学习了两种使用变量 v1 的方式：变量 v1 或者 p1 指向的变量。C++ 中“p1 指向的变量”指的就是 \*p。这与声明 p1 时使用了星号一样，但这里的星号却代表着完全不同的含义。这里使用的星号叫作解引用运算符，对指针变量的操作叫作解引用。

将上面的代码片段组合在一起可能会产生奇怪的现象，来看下面的代码：

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

上面的代码将在屏幕上输出如下内容：

```
42
42
```

一旦 p1 保存了指向 v1 的指针，那么 v1 和 \*p1 实际上代表着同一个变量。所以如果设置 \*p1 的值为 42，v1 的值也就变成了 42。

## 指针类型

在 C++ 中描述指针的类型与描述普通数据类型稍微有点不一致（或者至少有些容易让人混淆）。例如，如果想要声明一个形参类型为指向 int 型变量的指针，那么这个形参的类型名就是 int\*，下面是一个示例：

```
void manipulatePointer(int* p);
```

如果想要声明某一指针类型的变量，那么变量名就应该一直带着星号，下面是一个示例：

```
int *p1, *p2;;
```

其实编译器并不关心星号到底是放在 int 右边还是在变量名左边，所以下面的代码对编译器来说与上面的声明是等价的：

```
void manipulatePointer(int *p); // 正确的声明，但不如前一种声明方式好。
```

```
int *p1, *p2; // 正确的声明，但容易出问题。
```

尽管如此，我们仍然建议经常使用第一种声明。其实只要记住在声明的每一个指针变量的前面都应该加上星号就可以了。

可以用来获取变量地址的符号“&”和我们在声明函数时表明一个参数是引用型参数时所使用的符号是一样的，这并非巧合。回顾前面提到过的引用参数实质上是通过

向调用的函数中传递参数变量的地址来实现的，所以符号“&”的这两种用法本质上非常接近，尽管它们不是完全相同的。

### “\*”和“&”运算符

在指针变量前面使用“\*”运算符可以得到该指针指向的变量。这里使用的“\*”运算符叫作解引用运算符。

而在普通变量前面使用运算符“&”则可以获得该变量的地址，即产生一个指向该变量的指针，这里的“&”运算符就称为取址运算符。假设有下面的声明语句：

```
double *p, v;
```

下面的语句将给变量 p 赋值，使它指向变量 v：

```
p = &v;
```

\*p 就是 p 指向的变量，上面的赋值语句执行之后，\*p 和 v 实际上表示同一个变量。例如下面的语句中虽然没有出现变量 v 的名字，但结果是 v 的值被设置为 9.99：

```
*p = 9.99;
```

在赋值语句  
中的指针

可以将一个指针变量的值赋值给另一个指针变量。比如，如果 p1 仍然指向变量 v1，那么下面的赋值语句将使 p2 也指向变量 v1：

```
p2 = p1;
```

假设我们没有修改变量 v1 的值，那么下面的语句也将在屏幕上产生输出 42：

```
cout << *p2;
```

一定要注意，不要将下面两个赋值语句混为一谈：

```
p1 = p2;
```

和

```
*p1 = *p2;
```

上面的语句中，两个指针变量前面都带有星号时，对它们的操作就不再是对指针 p1 和 p2 的操作，而是对它们所指向的变量的操作。示例 10.1 中的图给出了具体的演示，其中变量用方框表示，变量的值在方框中。这里没有给出指针变量中存储的真正的地址值，因为这并不重要。重要的是该地址值表示了一个特定的变量。我们用箭头来指向该地址所代表的变量，而不会写出真正的地址值。

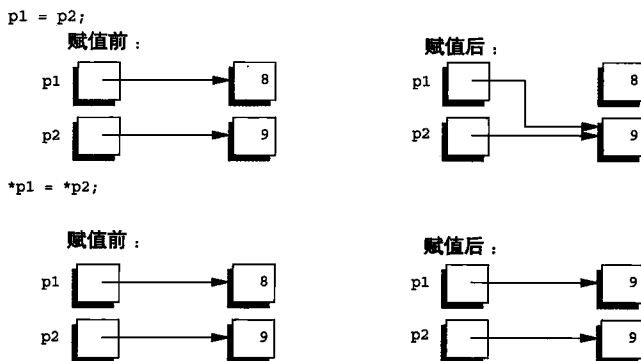
### 指针变量的赋值

如果 p1 和 p2 都是指针变量，那么下面的语句：

```
p1 = p2;
```

将把变量 p1 的值修改为 p2 中存储的内存地址。通常可以这样理解：赋值语句让 p1 指向了 p2 目前所指向的内容。

### 示例 10.1 指针变量的赋值操作



既然变量可以通过指针来访问，那么即使一个变量在程序中没有使用标识符来命名，也是可以访问的。运算符 **new** 可以用来生成一个没有使用标识符命名的变量。这个无名的变量可以通过指针来访问。例如下面的语句将生成一个 `int` 型变量，并将该变量的地址赋给指针变量 `p1`（即让指针 `p1` 指向这个新的无名变量）：

```
p1 = new int;
```

这个新的无名变量可以通过 `*p1` 访问（就像 `p1` 所指向的其他变量一样）。我们可以对这个无名变量做任何其他 `int` 型变量可以接受的操作，比如下面的代码从键盘上读入一个整数到这个无名变量后将它加 7，然后将结果输出：

```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

动态变量

上面的 **new** 运算符会生成一个新的无名变量并返回一个指向该变量的指针。**new** 运算符后面添加的类型就是这个新的无名变量的数据类型。通过 **new** 运算符产生的变量叫作**动态分配变量**或者简称为**动态变量**，因为这种变量是可以在程序运行时动态生成和销毁的。示例 10.2 演示了一些简单的指针和动态变量的操作。示例 10.3 对示例 10.2 中的程序运行进行了图解。

### 示例 10.2 基本指针操作

```
1 // 指针使用和动态变量的演示。
2 #include <iostream>
3 using namespace std;

4 int main( )
5 {
6     int *p1, *p2;
```

```

7      p1 = new int ;
8      *p1 = 42;
9      p2 = p1;
10     cout << "*p1 == " << *p1 << endl;
11     cout << "*p2 == " << *p2 << endl;

12     *p2 = 53;
13     cout << "*p1 == " << *p1 << endl;
14     cout << "*p2 == " << *p2 << endl;

15     p1 = new int ;
16     *p1 = 88;
17     cout << "*p1 == " << *p1 << endl;
18     cout << "*p2 == " << *p2 << endl;

19     cout << "Hope you got the point of this example!\n";
20     return 0;
21 }

```

### 示例运行结果

```

*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!

```

### new 运算符

new 运算符创建一个新的特定类型的动态变量，并且返回指向该变量的指针。例如，下面的代码将创建一个 MyType 类型的动态变量，并用指针 p 指向该动态变量：

```

MyType *p;
p = new MyType;

```

如果所创建的变量的类型是一个类，那么这个类的默认构造函数将被调用。当然我们可以通过传入不同类型和数量的参数来调用不同版本的构造函数，下面是一个示例：

```

MyType *mtPtr;
mtPtr = new MyType(32.0, 17); // 调用构造函数 MyType(double, int)。

```

对于类型不是类的动态变量来说，也可以使用类似的方法来初始化变量，示例如下：

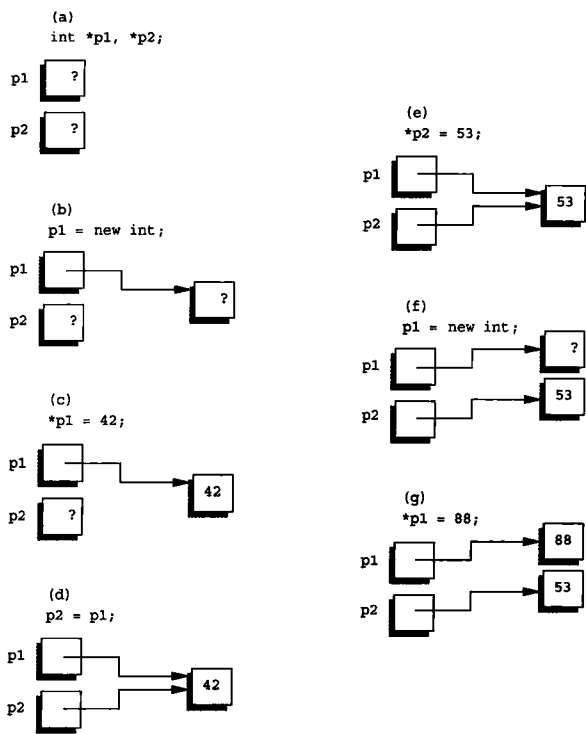
```

int *n;
n = new int (17); // 将 *n 初始化为 17。

```

在早期的 C++ 编译器中，如果操作系统没有足够的剩余内存来创建新的变量，new 运算符将会返回一个特定的名为 NULL 的指针。但是在现代的 C++ 标准中，遇到这种没有足够内存的情况，new 运算符将会中断整个程序。

示例 10.3 示例 10.2 的图示说明



当 new 运算符创建的动态变量的类型是类的时候，该类的构造函数会被自动调用。如果没有指明调用哪个构造函数，那么将调用默认构造函数。下面的代码将会调用类的默认构造函数：

```
SomeClass *classPtr;  
classPtr = new SomeClass; // 调用类的默认构造函数。
```

如果调用时给出了构造函数的参数，就可以调用不同版本的构造函数。示例如下：

```
classptr = new SomeClass(32.0, 17);  
// 调用构造函数 SomeClass(double, int)。
```

当创建的动态变量的类型不是类的时候，也可以使用类似的方式来初始化变量。示例如下：

```
double *dPtr;  
dPtr = new double (98.6); // 将 *dPtr 的值初始化为 98.6。
```

指针参数

指针是一种完善的数据类型，我们可以像使用其他数据类型一样使用它。特别是，

函数的参数可以声明为指针类型，函数的返回值也可以是一个指针。例如，下面的函数有一个指向 `int` 型变量的指针类型的形参，并且返回一个整型指针：

```
int* findOtherPointer(int* p);
```

### 自测练习题

1. 在 C++ 中什么是指针？
2. 给出三种以上 “\*” 运算符的用法，并描述每一种的名字和用途。
3. 下面的代码将产生什么输出？

```
int *p1, *p2;
p1 = new int ;
p2 = new int ;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

如果将上面代码中的语句做如下的替换：

```
*p1 = 30;
```

换成：

```
*p2 = 30;
```

那么将输出什么结果？

4. 下面的代码将产生什么输出？

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2; // 这与自测练习题 3 中的不同。
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

### 内存管理基础

自由存储  
或堆

在内存中有一块我们称之为自由存储或堆的区域，是专门用来保存动态分配的变量的。所有在程序中动态创建的变量都使用堆中的内存保存。如果程序中创建了太多的动态变量，就可能将堆中的内存空间全部用完。这种情形下，所有其余的 `new` 运算符的调用都将失败。在不同的编译器版本中，在堆中内存耗尽（所有为动态变量准备的内存都已经被分配）之后再调用 `new` 运算符会产生不同的错误。在老式的 C++ 编译



器中如果已经没有足够的内存用来创建新的动态变量，那么 new 运算符将返回一个名为 NULL 的值。在较新的 C++ 编译器中，完全符合新的 C++ 标准的情形下，new 运算符的调用将会使程序中止。在第 18 章中将会介绍在堆中的内存被 new 运算符耗尽之后，如何在不中止程序的情况下进行特殊处理。<sup>2</sup>

如果编译器的版本比较老，那么就可以通过检查 new 运算符返回的值是否等于 NULL 来判断 new 运算符的调用是否成功。比如下面的代码用来尝试创建新的动态变量并检查是否成功。如果 new 运算符的调用失败了，程序将会在给出错误信息后自行中止。

```
int *p;
p = new int;
if (p == NULL)
{
    cout << " Error: Insufficient memory.\n";
    exit(1);
}
// 如果 new 运算符的调用成功了，程序将继续运行。
```

(需要记住的是，由于上面的代码中使用了 exit 函数，所以应该包含该函数所在库的头文件 <cstdlib> 或者是在程序中使用头文件 <stdlib.h>。)

NULL 是 0

常量 NULL 实际上的值是 0。我们之所以习惯将其写作 NULL，是因为这样能够使代码的意思更容易理解，可以将 NULL 赋值给一个指针变量。在本书后面的内容中还将讨论 NULL 的其他用途。常量 NULL 在不止一个标准库中都有定义，例如 <iostream> 和 <cstdlib> 等，因此在使用 NULL 时可以使用 include 语句包含 <iostream> 或者 <cstdlib> (或者其他的库)。

如前所述，NULL 实际上的值是数字 0。NULL 的定义会被 C++ 预处理器处理，将代码中所有的 NULL 替换为 0。编译器实际上不会读取到真正的字面意义上的“NULL”，因此也就不存在命名空间的问题，使用 NULL 不需要使用任何 using 语句。<sup>3</sup> 尽管本书倾向于在代码中使用 NULL 而不是数字 0，但同时也有其他一些人与作者持相反观点，更支持使用 0 表示空指针，而不是 NULL。

#### NULL

NULL 是一个特殊的指针常量，可以将它赋值给指针变量来表明该指针变量为空 (即没有指向任何变量)。NULL 可以被赋值给任何类型的指针变量。常量 NULL 在许多库中都有定义，比如 <iostream> 等 (常量 NULL 的值实际上是整数 0)。

(注意不要将 NULL 指针和用来表示 C 字符串结束的空字符 ‘\0’ 弄混。二者并没有任何相同的地方。一个的值是整数 0，而另外一个字符 ‘\0’。)

在创建新的动态变量的时候，新版本的编译器无须进行上面的显式检查。因为在

<sup>2</sup> new 运算符会抛出一个异常，如果不捕获此异常，程序将会中止。这个异常可以被捕获，这将在第 18 章中讨论。

<sup>3</sup> 实际上，NULL 的定义使用了 #define，这是一种继承自 C 语言的定义方法，由预处理器负责。

新版本的编译器中调用 `new` 运算符创建动态变量失败时，程序将会自动给出错误信息并中止。不过，不管是什么版本的编译器，为程序添加上面的检查代码都没有坏处，而且能够使程序的逻辑更加合理。

C++ 中每个运行实例的自由存储的大小是不一样的。但一般情况下自由存储的空间是足够大的，而且一个好的程序一般不会将自由存储中所有的内存都用尽。尽管如此，一个好的习惯是将所有不再使用的自由存储内存进行回收。当程序中已经肯定不会再使用某个动态变量时，该变量使用的自由存储内存可以被回收回去，这样后面的代码中出现动态变量时就可以使用足够的自由存储。运算符 `delete` 能够将动态变量删除并将该变量所占用的内存空间释放到自由存储中。假设 `p` 是一个指向动态变量的指针变量，下面的语句可以将 `p` 所指向的动态变量销毁掉，并将该变量所占用的自由存储内存空间释放到自由存储中：

`delete`

```
delete p;
```

### delete 运算符

`delete` 运算符可以删除一个动态变量并将该变量所占用的内存空间释放到自由存储中。释放的内存空间可以再次被使用，来创建新的动态变量。下面的语句将指针 `p` 所指向的动态变量删除：

```
delete p;
```

调用运算符 `delete` 之后，指针变量（比如上面的 `p`）的值会变为一个随机值。（在本章随后的内容中还会看到，要删除一个动态数组时，`delete` 运算符的使用方式会有所不同。）

### 陷阱：悬空指针

悬空指针

当对一个指针使用了 `delete` 操作后，指针变量所指向的动态变量将会被销毁。不过指针变量的值将会变为一个不确定的随机值，也就是说我们无法知道该指针到底指向什么地方。如果其他指针变量指向的动态变量被销毁，这些指针变量也会变为不确定的随机值。这种不确定的指针变量被称为悬空指针（dangling pointer）。假设 `p` 是一个悬空指针，程序对 `p` 使用解引用运算符后得到的结果是不可预料的，而且常常是灾难性的。在指针变量使用解引用运算符时，必须保证该指针变量指向了某个变量。

C++ 中并没有内置检查机制来判断一个指针变量是不是悬空指针。一个避免出现悬空指针的办法是将所有的悬空指针变量都赋值为 `NULL`。这样程序在对指针变量使用解引用运算符前，就可以通过判断该指针是否等于 `NULL` 来检查指针变量是否指向了一个真正的变量。这种方法应该在每一个 `delete` 运算符调用以后使用，将所有的悬空指针都赋值为 `NULL`。需要注意的是，在对一个指针变量使用 `delete` 以后，其他与这个指针指向同一个变量的指针也都会变成悬空指针，所以应该将所有悬空指针都赋值为 `NULL`。每个程序员都应该跟踪所有的悬空指针，并将其赋值为 `NULL`，或者确保悬空指针变量不会被解引用。■

## 动态变量和自动变量

通过 `new` 运算符创建的变量叫作**动态变量**（或**动态分配变量**），因为它们是在程序运行时创建和销毁的。局部变量（即在一个函数定义内部定义的变量）虽然也具有某些动态特性，但与动态变量相比是有很多区别的。如果一个变量的声明包含在函数体内，那么只有在 C++ 程序调用该函数时这个变量才会被创建，而当函数执行完成后该变量会被自动销毁。一般情况下一个程序的主体是一个名为 `main` 的函数，在 `main` 函数中声明的局部变量也有相同的特性（由于对 `main` 函数的调用一直持续到整个程序的结束，所以在 `main` 函数中声明的变量只有在程序结束时才被销毁。但在对局部变量的处理机制上 `main` 函数和其他函数是完全一样的）。这些局部变量也叫作**自动变量**，因为它们的动态特性是由操作系统自动控制的。当局部变量所在的函数被调用时，这些局部变量就会自动创建；而当函数调用完成时，这些变量则会被自动销毁。

自动变量

全局变量

在任何函数（包含 `main` 函数）和类定义之外声明的变量叫作**全局变量**。这些全局变量有时也叫作**静态变量**，因为相对于动态变量和自动变量来说，全局变量的确是静态的。我们已经在第 3 章中简要地介绍了全局变量，正如当时所说的，在程序中全局变量并不是必需的，而且本书中的程序也不会使用全局变量。<sup>4</sup>

### 提示：定义指针类型

我们可以为指针的数据类型定义一个类型名，这样就可以使用该类型名来声明指针变量。这使得指针变量的定义和其他类型变量一样，而无须在每个指针变量前使用星号。下面的语句定义了一个名为 `IntPtr` 的数据类型，可以用 `IntPtr` 来定义指向 `int` 型变量的指针变量：

```
typedef int* IntPtr;
```

有了上面的定义之后，下面的两种声明指针变量的方式完全等价：

```
IntPtr p;
```

和

```
int *p;
```

typedef

我们可以使用 `typedef` 来为任意数据类型定义别名。例如，下面的语句将类型名 `Kilometers` 的意义定义为和 `double` 完全一样：

```
typedef double Kilometers;
```

只要有了上面的定义，就可以使用下面的语句来定义 `double` 变量：

```
Kilometers distance;
```

给已经存在的类型创建一个别名有时是非常有用的。不过 `typedef` 的主要用途还是用来定义指针变量类型。

值得注意的是 `typedef` 语句并不会产生新的数据类型，它仅仅是为一个已经存在

<sup>4</sup> 在声明某个类中的变量时使用的 `static` 修饰符与本节讨论的静态/动态修饰词的含义是不同的。

类型定义了一个新的名字。只要之前给出了 Kilometers 的定义，那么 Kilometers 类型的变量就可以使用 double 类型的变量来代替，Kilometers 和 double 实际上是同一种数据类型的两个名字。

使用像前面 IntPtr 那样的预先定义好的指针类型名有两个优点。首先，可以避免忘记星号的失误。前面已经提到过，声明两个指针变量 p1 和 p2 的时候，下面的语句是错误的：

```
int *p1, p2;
```

因为 p2 前面的星号被忘记了，这会造成 p2 被声明为一个普通的 int 型变量，而不是一个指针变量。如果将星号紧跟在 int 后面书写，其结果也是一样的。C++ 允许将星号紧跟在类型名后，比如下面的语句是合法的：

```
int* p1, p2;
```

虽然上面的语句是合法的，但是却容易造成误会。该语句容易让人觉得 p1 和 p2 都被声明成为了指针变量。但实际上只有 p1 是指针变量，而 p2 只是普通的 int 型变量。对 C++ 编译器来说，将星号紧跟在类型名后书写与放在变量名前书写的效果是完全相同的，将 p1 和 p2 一起声明成指针变量的一个正确的方法如下：

```
int *p1, *p2;
```

还有一个更加简单又不容易出错的声明方式，即使用已经定义好的类型名 IntPtr 来声明指针变量，方式如下：

```
IntPtr p1, p2;
```

使用预先定义的指针类型别名的第二个优点，会在定义某些函数的时候体现。如果函数的参数是指针的引用，那么在函数声明时就需要在参数中同时使用 \* 和 &，这很容易导致混淆。如果使用预先定义的指针类型名，那么声明一个参数为指针的引用就变得很简单。定义方式就如同定义其他类型的引用参数一样。下面给出一个声明例子：

```
void sampleFunction(IntPtr& pointerVariable); ■
```

## 自测练习

5. 下面的声明容易让人产生怎样的错误理解？

```
int* IntPtr1, IntPtr2;
```

6. 假定生成了一个动态变量：

```
char *p;
p = new char ;
```

如果指针变量 p 的值没有改变（即仍然指向同一个动态变量），那么应该如何销毁这个动态变量，并将其占用的内存空间释放，以便重新使用？

7. 请书写一个名为 NumberPtr 的类型定义，该类型为指向 double 类型的动态变量的指针。同时声明一个名为 myPoint 的指针变量，其类型为 NumberPtr。
8. 请描述 new 运算符的原理。它的返回值是什么？它的错误消息提示是什么？

## 类型的定义

可以为一个数据类型起一个新的名字，然后用这个类型名来声明变量。这是通过关键字 `typedef` 来实现的。这种类型的定义常常放在 `main` 函数和其他函数体之外，一般情况下放在文件的开始部分。这样该类型定义就是在全局范围内有效的，变量可以被整个程序使用。我们常常使用类型定义来重新定义指针类型的名字，如下面的示例所示。

### 语法

```
typedef Known_Type_Definition New_Type_Name;
```

### 示例

```
typedef int* IntPtr;
```

在此之后类型名 `IntPtr` 就可以用来声明指向 `int` 型变量的指针变量，下面是一个例子：

```
IntPtr pointer1, pointer2;
```

## 陷阱：指针用作传值参数

当传值参数是指针的时候，情况有时是非常微妙甚至是非常麻烦的，比如示例 10.4 中的函数调用。函数 `sneaky` 的参数 `temp` 是一个传值参数，因此它是一个局部变量。当这个函数被调用时，`temp` 的值先被赋值为实参 `p` 的值，然后函数体开始执行。由于 `temp` 是一个局部变量，因此所有对变量 `temp` 的修改在函数 `sneaky` 之外都是没有意义的，指针变量 `p` 的值应该不会改变。但是实际上，运行结果显示指针变量 `p` 的值发生了改变。在调用函数 `sneaky` 之前，`*p` 的值是 77，而调用结束后 `*p` 的值变成了 99。到底发生了什么事？

示例 10.5 对函数的运行情况进行了图解。尽管从运行结果上来看 `p` 的值似乎被修改了，但实际上 `p` 的值在调用 `sneaky` 函数后并没有变化。在这里指针 `p` 关联的值有两个：`p` 的指针值和 `p` 所指向的变量的值。变量 `p` 的值是一个指针值（即一个内存地址）。在调用函数 `sneaky` 之后，变量 `p` 的指针值（即相同的内存地址）并没有变化。`sneaky` 函数的调用仅修改了 `p` 所指向的变量的值，并没有修改变量 `p` 本身的值。

假设参数类型是一个包含指针类型成员变量的类或者结构体，而且函数的这个参数是传值参数，也可能会发生上面所描述的情况。然而，对于类类型我们可以通过定义拷贝构造函数来避免这种不受控的值的变化的，这将在本章后面的内容中讨论。■

### 示例 10.4 指针类型的传值参数

```
1 // 用来演示指针类型的传值参数的程序。
2 #include <iostream>
3 using namespace std;
4
5 typedef int* IntPtr;
6
7 void sneaky(IntPtr temp);
```

```

6  int main( )
7  {
8      IntPtrter p;

9      p = new int ;
10     *p = 77;
11     cout << "Before call to function *p == "
12         << *p << endl;

13     sneaky(p);

14     cout << "After call to function *p == "
15         << *p << endl;

16     return 0;
17 }
18 void sneaky(IntPtrter temp)
19 {
20     *temp = 99;
21     cout << "Inside function call *temp == "
22         << *temp << endl;
23 }

```

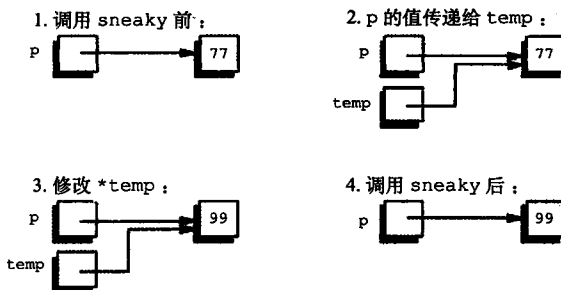
### 示例运行结果

```

Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99

```

### 示例 10.5 函数 `sneaky(p)` 的调用



### 指针的应用

在第 17 章中将会讨论如何使用指针来创建更丰富的数据结构。本章仅仅讨论指针的一个基本用法，就是对数组的引用。指针可以被用来创建和引用一种特殊的数组：动态数组。动态数组将在接下来的 10.2 节中讨论。

## 10.2 动态数组

### 动态分配 数组

在本节中我们将会看到数组变量实际上是一个指针。同样还将介绍在程序中如何使用动态数组。所谓**动态分配数组**（可以简单地称之为**动态数组**）是指数组的大小不在编写代码时确定，而是在程序运行时动态决定的数组。

### 数组变量和指针变量

第5章介绍了数组在内存中的具体存储方式。当时我们使用术语“内存地址”来解释数组，但实际上内存地址就是一个指针。因此在C++中，数组变量实际上就是某种类型的指针变量，它指向数组的第一个变量。在下面的两个变量声明中，p和a都是指针变量：

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

示例10.6演示了a和p都是指针变量的情况。既然1是指向某个整型变量（这里就是变量a[0]）的指针，那么a的值就可以赋值给指针变量p：

```
p = a;
```

经过上面的赋值操作后p就和a一样指向了同一个存储区域。所以p[0]，p[1]，…，p[9]的值就是数组的索引变量a[0]，a[1]，…，a[9]的值。只要指针变量所指向的内存是一个数组，那么数组运算符同样可以用于该指针变量。经过上面的赋值操作后，你可以将指针变量p当作一个数组名。同样，也可以把数组名a当作一个指针变量，但是这里有一个非常重要的限制：不能更改数组变量代表的指针变量的值。如果指针变量p2已经指向某个变量，我们会觉得下面的语句是合法的，但事实上并非如此：

```
a = p2; // 非法，不能为a分配新的值。
```

这是因为数组变量的数据类型并不是int\*，而是一个const int\*，即常量类型。所有的数组变量（如这里的a）都是一个用const修饰的变量，这种变量的值不能被随意更改。

### 示例 10.6 数组和指针变量

```
1 // 用来演示数组可以作为指针变量的程序。
2 #include <iostream>
3 using namespace std;
4
5 typedef int* IntPtr;
6
7 int main( )
8 {
9     IntPtr p;
10    int a[10];
11    int index;
12
13    for (index = 0; index < 10; index++)
```

```

11         a[index] = index;
12     p = a;
13     for (index = 0; index < 10; index++)
14         cout << p[index] << " ";
15     cout << endl;
16     for (index = 0; index < 10; index++)
17         p[index] = p[index] + 1;
18     for (index = 0; index < 10; index++)    注意对数组 p 的改动会影响数组 a。
19         cout << a[index] << " ";
20     cout << endl;
21     return 0;
22 }

```

#### 示例运行结果

```

0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10

```

### 创建并使用动态数组

第 5 章介绍的数组存在一个问题，那就是我们必须在程序编写的时候就确定数组的大小。但实际情况中，在程序运行前我们可能难以知道自己需要多大的数组。例如，一个用来存储学生 ID 的数组，每次程序运行时每个班学生的数量是不同的。如果使用前面介绍的普通数组，就必须估计一下数组可能需要的最大空间的大小，然后将数组定义得足够大。不过这样做可能存在两个问题：其一是我们的估计可能不够大，导致程序在某些情况下不能运行。其二是可能许多情况下数组中存在大量不会使用的空间，这导致大量的内存被浪费。使用动态数组能够避免出现这些问题。如果在程序中使用动态分配的数组来保存学生的 ID 号，就可以将学生的人数作为程序的一个输入，程序就可以创建一个大小与输入的学生人数正好相符的动态数组。

#### 创建动态数组

动态数组也是通过运算符 `new` 来创建的，其创建和使用都非常简单。由于数组变量实际上是一个指针变量，因此可以使用 `new` 运算符来创建数据类型为数组的动态变量，并将创建的动态数组作为一般数组对待。下面的代码创建了一个包含 10 个元素的 `double` 类型动态数组：

```

typedef double* DoublePtr;
DoublePtr d;
d = new double [10];

```

要获得其他类型的动态数组，只需将上面的 `double` 替换为其他数据类型即可。当然，也可以使用类或者结构体类型来代替上面的 `double`。如果用其他数字来代替上面代码中的 10，就可以获得任意大小的动态数组。

上面的例子中有一些不太明显的事情需要注意一下。首先，指向动态数组的指针类型和指向数组中单个元素的指针类型是完全一样的。比如指向 `double` 类型数组元



素的指针类型和指向单个 double 变量的指针类型是相同的。而指向数组的指针实际上就是指向数组首个变量的指针。上面的例子创建了包含 10 个元素的数组，同时 p 指向了 10 个数组变量中的第一个变量。

其次，需要注意在调用 new 运算符创建动态数组时，数组的大小在类型（本例中为 double）后面的方括号中给出，括号中的数字告诉操作系统为新建的动态数组分配多大的内存空间。如果在上面的例子中省略方括号和其中的 10，那么操作系统将会仅分配一个 double 类型变量的存储空间给 p，而不是分配一个包含 10 个 double 型变量的数组。

示例 10.7 演示了如何在程序中使用动态数组。程序可以查看存储在动态数组中的数字。数组的大小将在程序运行时决定。程序运行时会询问用户需要多少个数字，根据输入的数字使用 new 运算符来创建动态数组。动态数组的大小由变量 arraySize 表示。动态数组的大小不一定是一个常量，它也可以是一个变量，其值在程序运行时才能够确定（如示例 10.7 所示）。

### 示例 10.7 动态数组

```

1  // 搜索键盘输入的数字。
2  #include <iostream>
3  using namespace std;

4  typedef int* IntPtr;

5  void fillArray(int a[], int size);
6  // 前提条件：size 是数组 a 的大小。
7  // 运行结果：a[0] 到 a[size-1] 之间的值
8  // 使用键盘输入数字。

9  int search(int a[], int size, int target);
10 // 前提条件：size 是数组 a 的大小。
11 // 运行结果：a[0] 到 a[size-1] 之间的值
12 // 如果查找目标在数组中，返回第一个结果的索引。
13 // 如果不在数组中，返回 -1。

14 int main( )
15 {
16     cout << "This program searches a list of numbers.\n";

17     int arraySize;
18     cout << "How many numbers will be on the list? ";
19     cin >> arraySize;
20     IntPtr a;
21     a = new int [arraySize];

22     fillArray(a, arraySize);

23     int target;
24     cout << "Enter a value to search for: ";
25     cin >> target;
26     int location = search(a, arraySize, target);
27     if (location == -1)

```

常见的数组类型参数。

动态数组的使用和普通数组是一样的。

```

28     cout << target << " is not in the array.\n";
29     else
30         cout << target << " is element " << location << " in the
            array.\n";
31
32     delete [] a;
33
34     return 0;
35 }
36 // 使用库 <iostream>。
37 void fillArray(int a[], int size)
38 {
39     cout << "Enter " << size << " integers.\n";
40     for ( int index = 0; index < size; index++)
41         cin >> a[index];
42 }
43
44 int search(int a[ ], int size, int target)
45 {
46     int index = 0;
47     while ((a[index] != target) && (index < size))
48         index++;
49     if (index == size) // 如果 a 中没有搜索的目标。
50         index = -1;
51     return index;
52 }

```

### 示例运行结果

```

This program searches a list of numbers.
How many numbers will be on the list? 5
Enter 5 integers.
1 2 3 4 5
Enter a value to search for: 3
3 is element 2 in the array.

```

**delete []**      示例 10.7 中的 delete 语句用来销毁 a 所指向的动态数组。由于程序在这里就结束了，所以 delete 语句并不是必需的。如果之后程序还包括其他操作，那么就必须使用 delete 语句将 a 指向的动态数组占用的内存释放回自由存储中。用于释放动态分配内存的 delete 语句和我们前面所使用过的 delete 语句类似，所不同的是在指针变量名前面添加一对空的方括号，下面是一个例子：

```
delete []a;
```

方括号告诉 C++ 将删除一个动态分配的数组，因此操作系统将检查数组的大小并将数组中所有的元素删除。如果省略上面语句中的方括号，就无法将整个数组删除，下面是一个例子：

```
delete a;
```

上面的语句是错误的，但许多编译器却无法给出任何错误信息。C++ 标准对于这种情况没有给出具体的规定，这意味着各版本的编译器可以根据自身规则（是方便编

译者作者，而不是方便程序员)对此给出任何反应。

因此，对于类似下面的动态数组：

```
arrayPtr = new MyType[37];
```

应该总是使用下面的方式释放内存：

```
delete [] arrayPtr;
```

同时，应该注意方括号在 delete 语句中出现的位置：

```
delete [] arrayPtr; // 正确
delete arrayPtr[]; // 错误！
```

对一个指针使用 new 运算符可以创建一个动态数组，例如示例 10.7 中的指针 a。一旦调用 new 为一个指针赋值后，就不应该为该指针分配其他的指针值，因为重新为这个指针赋值会导致操作系统在使用 delete 运算符来释放动态数组所占用的内存时发生错误。

动态数组是通过 new 运算符和一个指针变量来创建的。如果程序不再需要使用该动态数组时，应该调用 delete 将其所占用的内存释放回自由存储中。除了这点不一样以外，动态数组可以被当作普通数组一样使用。

### 示例：一个返回数组的函数

在 C++ 中，函数的返回值类型不能是数组类型。例如，下面的声明是非法的：

```
int [] someFunction( ); // 非法的
```

但如果你希望声明一个可以返回数组的函数，那么就只能返回一个该数组类型的指针，并将该指针指向需要返回的数组。因此，函数的声明应该如下所示：

```
int* someFunction( ); // 合法的
```

示例 10.8 给出了一个返回指向数组的指针的函数的示例。

### 示例 10.8 返回一个指向数组的指针

```
1  #include <iostream>
2  using namespace std;

3  int* doubler(int a[], int size);
4  // 前提条件：size 是数组 a 的大小。
5  // 数组 a 中的所有元素都已经赋值。
6  // 返回值：一个指向和数组 a 大小相同的数组的指针，
7  // 这个数组中的元素的值都是 a 中对应元素的两倍。

8  int main()
9  {
10     int a[] = {1, 2, 3, 4, 5};
11     int* b;

12     b = doubler(a, 5);
```

```

13     int i;
14     cout << "Array a:\n";
15     for (i = 0; i < 5; i++)
16         cout << a[i] << " ";
17     cout << endl;
18     cout << "Array b:\n" ;
19     for (i = 0; i < 5; i++)
20         cout << b[i] << " ";
21     cout << endl;

22     delete [] b; ←—————— 程序马上将结束的时候使用的这个 delete
23     return 0;                  并不是必需的,但是在其他许多情况下,使用
24 }                               delete 还是必要的。

25 int* doubler(int a[], int size)
26 {
27     int* temp = new int [size];

28     for (int i =0; i < size; i++)
29         temp[i] = 2*a[i];

30     return temp;
31 }

```

#### 示例运行结果

```

Array a:
1 2 3 4 5
Array b:
2 4 6 8 10

```

#### 自测练习题

9. 请编写一个指针变量的类型定义,该指针变量用来指向一个动态分配数组,数组的元素数据类型为 char, 类型名为 CharArray。
10. 假定程序中包含下面创建动态分配数组的代码:
 

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

 因此指针变量 entry 指向了这个动态分配数组。请编写程序从键盘上输入 10 个数为数组元素赋值。
11. 假定程序中包含自测练习题 10 中的创建动态分配数组的代码,并且指针变量 entry 的值始终都没有改变。请编写用于删除动态分配数组并将内存释放回堆中的代码。
12. 下面的代码会输出什么?

```

int a[10];
int arraySize = 10;
int *p = a;
int i;
for (i = 0; i < arraySize; i++)
    a[i] = i;
for (i = 0; i < arraySize; i++)

```

```
cout << p[i] << " ";
cout << endl;
```

## 指针运算

指针也可以进行某些运算，但这些运算都是针对地址进行的，而不是通常意义上的数学运算。比如假设程序中包含下面的代码：

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

上面的语句执行之后，`d` 的值就是变量 `d[0]` 的地址。表达式 `d+1` 的值就等价于 `d[1]` 的地址，`d+2` 的值就是 `d[2]` 的地址，依此类推。需要注意的是，虽然变量 `d` 的值是一个地址而地址的值实际上是一个数字，但表达式 `d+1` 并非简单地在 `d` 代表的数字上加 1。如果一个 `double` 类型变量需要 8 个字节（8 个内存基本单元）保存，而且包含的地址值为 2000，那么 `d+1` 的值就是地址 2008。这里的数据类型 `double` 可以替换为其他任意数据类型，指针的加法是以一个该类型变量所占据的字节长度为单元移动单位的。

指针的运算提供了另外一种操作数组的方法。如果 `arraySize` 是指针变量 `d` 所指向的动态数组的大小，那么下面的语句可以用来输出数组中所有的元素：

```
for (int i = 0; i < arraySize; i++)
    cout << *(d + i) << " ";
```

下面的代码与它是完全等价的：

```
for (int i = 0; i < arraySize; i++)
    cout << d[i] << " ";
```

指针的运算仅限于加法和减法运算，不能进行乘法和除法运算。我们可以把一个指针加上一个整数，或者减去一个整数，或者将两个相同类型的指针相减。两个指针相减的结果是两个地址间的连续变量的个数。需要记住的是，当两个指针相减时，这两个指针指向的变量必须属于同一个数组。指向不同数组的两个指针做减法是没有意义的。

指针变量还可以使用自增和自减运算符（“++”和“--”）进行运算。例如，`d++` 将会使 `d` 指向数组中下一个变量的地址，而 `d--` 则会使 `d` 的值变为上一个变量的地址。

## 多维动态数组

我们还可以创建多维的动态数组。多维数组实际上就是数组的数组（二维数组），或者数组的数组的数组（三维数组）。创建一个二维动态数组，实际上就是创建一个数组的数组。为了创建一个 `int` 型二维数组，首先要创建一个为 `int*` 类型的一维数组，即它的元素是 `int` 型的一维数组。然后再为所有的元素数组创建一个 `int` 类型的一维动态数组。

使用类型定义可以使程序比较清晰。下面是一个普通的 `int` 型一维动态数组的类型定义：

```
typedef int* IntArrayPtr;
```

要获得一个三行四列 `int` 型二维数组，首先需要有一个数据类型为 `IntArrayPtr` 的数组。例如：

```
IntArrayPtr *m = new IntArrayPtr[3];
```

上面创建的是一个含有三个指针的数组，每一个指针都可以指向一个 `int` 型动态数组，下面是一个示例：

```
for (int i = 0; i < 3; i++)
    m[i] = new int[4];
```

最终我们得到的数组 `m` 就是一个三行四列的二维动态数组。示例 10.9 中给出了一个简单的程序示例。

### 如何使用动态数组

- 定义一个指针类型：定义一个指向的变量类型与该数组元素类型相同的指针。如果动态数组是一个 `double` 类型数组，那么指针类型的定义如下：

```
typedef double* DoubleArrayPtr;
```

- 声明一个指针变量：使用前面定义的指针类型声明一个指针变量。该指针变量指向内存中的动态数组，它可以作为该动态数组的名字使用。

```
DoubleArrayPtr a;
```

(如果没有定义指针类型，使用语句 `double *a;` 来代替上面的语句。)

- 调用 `new` 运算符：使用 `new` 运算符来创建动态数组：

```
a = new double [arraySize];
```

在上面的语句中，动态数组的大小在括号中给出。其数值可以是一个 `int` 型变量或者其他 `int` 型表达式。在上面的例子中，`arraySize` 是一个 `int` 型变量，它的值可以在程序运行时再确定。

- 当作普通数组使用：指针变量（如上面的 `a`）可以像使用普通数组那样被使用。例如，数组元素变量可以有相同的书写方式：`a[0]`、`a[1]` 等。不应该给该指针变量赋值新的值，而应该像使用数组变量一样使用它。
- 调用 `delete[]`：当动态数组变量在程序中使用完后，应该使用 `delete` 和运算符“`[]`”来删除动态分配的数组，将其使用的内存释放到自由存储中，以便其他程序使用。比如：

```
delete [] a;
```

### 示例 10.9 二维动态数组

```
1 #include <iostream>
2 using namespace std;

3 typedef int* IntArrayPtr;

4 int main( )
```

```

5 {
6     int d1, d2;
7     cout << "Enter the row and column dimensions of the array:\n"
8     cin >> d1 >> d2;

9     IntArrayPtr *m = new IntArrayPtr[d1];
10    int i, j;
11    for (i = 0; i < d1; i++)
12        m[i] = new int [d2];
13    //m是一个d1行d2列的二维数组。

14    cout << "Enter " << d1 << " rows of "
15        << d2 << " integers each:\n";
16    for (i = 0; i < d1; i++)
17        for (j = 0; j < d2; j++)
18            cin >> m[i][j];

19    cout << "Echoing the two-dimensional array:\n";
20    for (i = 0; i < d1; i++)
21    {
22        for (j = 0; j < d2; j++)
23            cout << m[i][j] << " ";
24        cout << endl;
25    }

26    for (i = 0; i < d1; i++)
27        delete [] m[i];
28    delete [] m;

29

30    return 0;
31 }

```

注意这里每有一个 new, 就要有一个 delete[] 与之对应。  
(程序马上将结束的时候使用的这个 delete 并不是必需的, 但是在其他许多情况下, 使用 delete 还是必要的。)

### 示例运行结果

```

Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2

```

一定要注意示例 10.9 中的 delete 操作。动态数组 m 是一个数组的数组, 数组的每一个元素都是使用 new 运算符创建的动态数组, 因此每一个元素数组都必须使用 delete[] 释放其占用的内存空间, 然后才能将 m 本身使用 delete[] 释放。也就是说所有使用 new 运算符创建的动态数组都需要使用 delete[] 来释放 (这里的程序在调用 delete[] 之后就马上结束了, 所以省略接下来的 delete[] 语句是安全的, 只不过我们需要在这里演示 delete[] 的用法)。

delete[]

## 10.3 类、指针和动态数组

合作将永不停歇。

某广告印刷品

动态数组的基本类型可以是一个类。而类的成员变量也可以是一个动态数组。我们可以将二者按照任意一种方式联合起来使用。当联合使用类和动态数组时，还需要考虑一些事情，但是根本方法却和前面已经使用过的一样。本书所讨论的大部分技术内容不仅仅适用于包含动态数组的类，而对所有的动态分配结构都适用，例如将在第17章中讨论的链表等。

### 运算符->

箭头运算符

C++ 有一个只能和指针一起使用的运算符，用来指明结构体或者类的某个成员。这个运算符就是**箭头运算符**“->”，它结合了解引用运算符“\*”和点运算符的功能，可以用来指明一个指针所指向的结构的成员或者类对象的成员。如果有下面这样的结构体定义：

```
struct Record
{
    int number;
    char grade;
};
```

下面的代码将创建一个数据类型是 Record 的动态分配变量，并将该变量的两个成员变量分别赋值为 2001 和 'A'：

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

注意

```
p->grade
```

和

```
(*p).grade
```

上面代码中的表达式 p->grade 和表达式 (\*p).grade 代表的意义是完全相同的，不同的是第一种表达方法相对比较方便，也更加常见。

### this 指针

定义类的成员函数时经常需要使用到调用对象，this 指针就是预先定义好的指向调用对象的指针。假设有下面的类定义：

```
class Sample
{
public:
```



```

...
void showStuff() const;
...
private:
    int stuff;
    ...
};

```

下面两种对成员函数 showStuff 的定义是等价的：

```

void Sample::showStuff() const
{
    cout << stuff;
}
// 下面这样的定义写法并不是一个好的习惯，不过在这里可以演示 this 指针的使用：
void Sample::showStuff()
{
    cout << this ->stuff;
}

```

需要注意的是，this 并不是调用对象的名字，而是指向当前调用对象的指针名字。this 指针的值无法改变，它始终指向当前调用它的对象。

正如上面代码中的注释所说的，一般情况下并不需要使用 this 指针。但是在某些情况下使用 this 指针可以使编程更加方便。this 指针的一个常见用法就是赋值运算符“=”的重载（我们将在后面讨论赋值运算符的重载）。

由于 this 指针始终指向调用对象，因此不能在静态成员函数的定义中使用。静态成员函数一般都不需要调用对象，所以也就不存在 this 指针。

## 重载赋值运算符

本书常常将赋值运算符当作一个 void 类型的函数来使用。但是预定义的赋值运算符实际上返回了一个引用，这使得赋值运算符的使用有一些特殊的地方。

可以将赋值运算符连续书写： $a=b=c;$ ，这个表达式等价于  $a=(b=C);$ ，其中第一个赋值运算符  $b=c$  将返回赋值后的变量 b 的引用。因此表达式：

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

也会将 c 的值传递给 a。为了保证重载后的赋值运算符仍然支持这种使用方式，就需要赋值运算符在重载后返回一个和其左侧变量相同类型的值。接下来我们会看到，使用 this 指针可以方便地做到这一点。然而这个限制仅仅是赋值运算符需要返回一个和其左侧变量相同的类型，并不要求一定是引用。在下面介绍的赋值运算符的另外一种使用方式中，将会解释为什么一定要返回一个引用。

预定义赋值运算符要返回引用的目的，是让我们可以使用返回值直接调用成员函数。例如：

```
(a = b).f();
```

其中，f 是一个成员函数。如果希望赋值运算符的重载版本也可以通过这种方式来调用成员函数，就必须返回引用。这并不是一个重要的理由，因为这种调用方式仅仅是一个很少使用到的特点而已。不过习惯上我们在重载赋值运算符时仍然返回引用，因

为返回一个引用并不比返回一个变量麻烦多少。

假设有下面的类定义（此类可能用来处理一些特殊的字符，这比使用预定义的 `string` 类来处理要容易）：

```
class StringClass
{
public :
    ...
    void someProcessing( );
    ...
    StringClass& operator=(const StringClass& rtSide);
    ...
private :
    char *a; // string 类中的动态字符数组。
    int capacity; // 动态数组 a 的大小。
    int length; // a 的字符数。
};
```

= 必须是  
成员

正如第 8 章中所说的，重载赋值运算符时运算符必须作为类的成员重载。这也是上面赋值运算符的重载定义中只有一个参数的原因。假设有下面的语句：

```
s1 = s2; // s1 和 s2 是 StringClass 类类型。
```

= 的调用  
对象

上面的语句中 `s1` 就是调用对象，而 `s2` 是成员运算符“=”的参数。

随后的赋值运算符的重载定义允许连续的赋值：

```
s1 = s2 = s3;
```

语句的返回值可以用来调用成员函数，下面是一个示例：

```
(s1 = s2).someProcessing();
```

下面是 `StringClass` 类中赋值运算符的重载定义，定义中使用 `this` 指针访问运算符左边的对象（即调用对象）：

```
// 这个版本的定义并不能满足所有的使用情形。
StringClass& StringClass::operator=(const StringClass& rtSide)
{
    capacity = rtSide.capacity;
    length = rtSide.length;
    delete [] a;
    a = new char [capacity];
    for (int i = 0; i < length; i++)
        a[i] = rtSide.a[i];
    return *this ;
}
```

这个版本的赋值运算符使用在运算符两边是同一个对象的情况下会发生错误，例如：

```
s = s;
```

当执行上面的赋值语句时，下列语句就会执行：

```
delete [] a;
```

但由于调用对象是 `s`，所以上面语句的含义如下：

```
delete [] s.a;
```

这样指针 `s.a` 就变成了一个不确定的悬空指针。赋值运算将破坏对象 `a` 的内存结构，这可能将导致程序的崩溃。

对于许多类来说，赋值运算符的重载对于运算符两边都是同一个对象的情况是需要特别处理的。因此，我们应该始终检查是否会出现这种情况，保证所定义的赋值运算符重载可以应对各种情况。

要避免这种错误，可以在重载中使用 `this` 指针来检查赋值运算符两边是否是同一个对象的情况，示例如下：

```
// 修复了 bug 后的最终版本。
StringClass& StringClass::operator=(const StringClass& rtSide)
{
    if (this == &rtSide)
        // 如果右边的对象和左边的是同一个
        {
            return *this ;
        }
    else
    {
        capacity = rtSide.capacity;
        length = rtSide.length;
        delete [] a;
        a = new char [capacity];
        for (int i = 0; i < length; i++)
            a[i] = rtSide.a[i];

        return *this ;
    }
}
```

一个包含赋值运算符重载的例子将出现在下面的代码示例中（见示例 10.10 至示例 10.12）。

### 示例 10.10 一个包含动态数组成员的类型定义

```
1 // 这个类的对象将可以表示一个部分填充的数组。
2 class PFArrayD
3 {
4 public :
5     PFArrayD( );
6     // 初始的容量是 50。
7
8     PFArrayD(int capacityValue);
9
10    PFArrayD(const PFArrayD& pfaObject); ← 拷贝构造函数。
11
12    void addElement(double element);
13    // 前提条件：数组并没有被完全填充。
14    // 运行结果：参数 element 表示的值被添加到数组中。
15
16    bool full( ) const { return (capacity == used); }
17    // 数组被填满则返回 true，否则返回 false。
18
19    int getCapacity( ) const { return capacity; }
```

```

15     int getNumberUsed( ) const { return used; }

16     void emptyArray( ){ used = 0; }
17     // 清空数组。

18     double& operator[](int index);
19     // 访问数组中的元素，索引范围从 0 到元素个数-1。

20     PFArrayD& operator =(const PFArrayD& rightSide); ← 重载运算符。

21     ~PFArrayD( ); ← 析构函数。
22 private :
23     double *a; // 为 double 型数组定义一个指针。
24     int capacity; // 数组的长度。
25     int used; // 数组中已经使用的位置的个数。

26 };

```

---

### 示例 10.11 类 PFArrayD 的成员函数

---

```

1  // 类 PFArrayD 中的成员函数。
2  // 下面的 include 语句是必需的。
3  // #include <iostream>
4  // using std::cout;

5  PFArrayD::PFArrayD( ) :capacity(50), used(0)
6  {
7      a = new double [capacity];
8  }

9  PFArrayD::PFArrayD(int size) :capacity(size), used(0)
10 {
11     a = new double [capacity];
12 }

13 PFArrayD::PFArrayD(const PFArrayD& pfaObject)
14 :capacity(pfaObject.getCapacity( )), used(pfaObject.getNumberUsed( ))
15 {
16     a = new double [capacity];
17     for (int i = 0; i < used; i++)
18         a[i] = pfaObject.a[i];
19 }

20 void PFArrayD::addElement(double element)
21 {
22     if (used >= capacity)
23     {
24         cout << "Attempt to exceed capacity in PFArrayD.\n";
25         exit(0);
26     }
27     a[used] = element;
28     used++;

```

```

29 }
30
31 double& PFArrayD::operator[](int index)
32 {
33     if (index >= used)
34     {
35         cout << "Illegal index in PFArrayD.\n";
36         exit(0);
37     }
38
39     return a[index];
40
41 PFArrayD& PFArrayD::operator=(const PFArrayD& rightSide)
42 {
43     if (capacity != rightSide.capacity)
44     {
45         delete [] a;
46         a = new double [rightSide.capacity];
47
48         capacity = rightSide.capacity;
49         used = rightSide.used;
50         for (int i = 0; i < used; i++)
51             a[i] = rightSide.a[i];
52
53     return *this;
54 }
55
56 PFArrayD::~PFArrayD()
57 {
58     delete [] a;
59 }
60

```

注意：这里一定要检查赋值运算符两边是否是相同的对象。

### 示例 10.12 使用类 PFArrayD 的应用程序

```

1 // 展示 PFArrayD 类的代码。
2 #include <iostream>
3 using namespace std;
4
5 class PFArrayD
6 {
7     <类定义的其余部分与示例 10.10 中相同。>
8 };
9
10 void testPFArrayD()
11 {
12     // 测试 PFArrayD 类。
13
14 int main()
15 {
16     cout << "This program tests the class PFArrayD.\n";
17     char ans;
18     do

```

在第 11 章的 11.1 节中，我们将会演示如何将这个长文件分成三份短文件，分割后的文件内容大概就是这个示例及示例 10.10、示例 10.11。

```

15     {
16         testPFArrayD( );
17         cout << "Test again? (y/n) ";
18         cin >> ans;
19     } while ((ans == 'y') || (ans == 'Y'));

20     return 0;
21 }
22 <假设这里就是 PFArrayD 类的成员函数的定义。>
23 void testPFArrayD( )
24 {
25     int cap;
26     cout << "Enter capacity of this super array: ";
27     cin >> cap;
28     PFArrayD temp(cap);

29     cout << "Enter up to " << cap << " nonnegative numbers.\n";
30     cout << "Place a negative number at the end.\n";

31     double next;
32     cin >> next;
33     while ((next >= 0) && (!temp.full( )))
34     {
35         temp.addElement(next);
36         cin >> next;
37     }

38     cout << "You entered the following "
39         << temp.getNumberUsed( ) << " numbers:\n";
40     int index;
41     int count = temp.getNumberUsed( );
42     for (index = 0; index < count; index++)
43         cout << temp[index] << " ";
44     cout << endl;
45     cout << "(plus a sentinel value.)\n";
46 }

```

### 示例运行结果

This program tests the class PFArrayD.

Enter capacity of this super array: 10

Enter up to 10 nonnegative numbers.

Place a negative number at the end.

1.1

2.2

3.3

4.4

-1

You entered the following 4 numbers:

1.1 2.2 3.3 4.4

(plus a sentinel value.)

Test again? (y/n) n

### 示例：部分填充数组的类

示例 10.10 和示例 10.11 中定义了一个名为 `PFArrayD` 的类，用来表示一个部分填充的 `double` 型数组。<sup>5</sup> 和示例 10.12 的演示程序一样，类 `PFArrayD` 的对象不但可以像普通数组一样使用方括号来访问保存在其中的元素，还可以记录数组中包含了多少个元素。因此，类 `PFArrayD` 的对象就如同一个只有部分位置上有元素的数组一样。成员函数 `getNumberUsed` 返回数组中已经被使用的元素位置的个数，这个类在 `for` 循环中可以这样使用：

```
PFArrayD stuff(cap); //cap 是一个 int 型变量。
<一些为对象 stuff 的成员赋值的代码。>
for (int index = 0; index < stuff.getNumberUsed(); index++)
    cout << stuff[index] << " ";
```

类 `PFArrayD` 的对象包含了一个作为成员变量的动态数组。这个数组用来存储元素的值。实际上这个动态数组成员变量是一个指针变量。在类的构造函数中，这个成员变量被赋值成指向一个动态数组的指针。类 `PFArrayD` 中还包含两个 `int` 型成员变量：成员变量 `capacity` 用来记录动态数组的大小；而成员变量 `used` 则用来记录当前已经使用的数组位置的个数。和一般的部分填充的数组一样，所有的数组元素都是按顺序排列的，起始位置为 0，然后是 1, 2, …，依此类推。

类 `PFArrayD` 的对象可以被看作是一个部分填充的 `double` 型数组，而且还具备一些普通 `double` 型数组和动态 `double` 型数组都没有的优点。与标准的数组不同的是，当使用非法的索引值时它可以给出错误信息。而且类 `PFArrayD` 的对象无须使用额外的 `int` 型变量记录数组中当前所占用的元素位置。（也许你会说，这里当然有个 `int` 型变量，只不过它是成员变量。不过这个成员变量是一个私有成员变量，使用 `PFArrayD` 的程序员并不需要知道该成员变量的存在。）

类 `PFArrayD` 的对象只能用来存储 `double` 类型的值。在第 16 章中讨论有关模板的内容时，会看到将这里定义的类替换为适用于任意类型的模板类是非常容易的一件事，在那之前我们只处理 `double` 类型的元素。

在类 `PFArrayD` 的定义中，许多知识在之前都已经介绍过了，但这里仍然有三个新的知识点：拷贝构造函数、析构函数及赋值运算符的重载。我们随后将解释赋值运算符的重载，而拷贝构造函数和析构函数将在下面两个小节中讨论。

如果要探究为什么需要重载赋值运算符，我们先来假设示例 10.10 和示例 10.11 中不存在关于赋值运算符重载的定义。如果 `list1` 和 `list2` 按照如下方式声明：

```
PFArrayD list1(10), list2(20);
```

假设 `list2` 已经通过调用成员函数 `list2.addElement` 增加了一些数组元素，在没有重载赋值运算符的情况下，下面的语句虽然不会出现错误，但其运行结果也可能不是我们所期望的：

<sup>5</sup> 如果你已经看过了第7章中 `vector` 相关的内容，你会注意到这是一个比较弱的 `vector` 版本。尽管 `vector` 类完全可以替代这里的类，但这个例子中包含了很多本章介绍的技术，同时也可以帮助你了解 `vector` 类的实现原理。

```
list1 = list2;
```

如果没有重载赋值运算符，默认的赋值运算符将作用在上面的语句中。通常我们会以为，这个预定义的赋值运算符将把 list2 中所有的成员变量的值拷贝到 list1 对应的成员变量中。因此，list1.a 的值就等于 list2.a，而 list1.capacity 的值也等于 list2.capacity，list1.used 的值也等于 list2.used。但实际上并不是这样的。

成员变量 list1.a 是一个指针，而赋值运算符将它的值设置为与 list2.a 相等。因此 list1.a 和 list2.a 将指向内存中的同一个位置。之后当改变数组 list1.a 的元素值的时候，也将改变数组 list2.a 的元素值。同样地，如果改变数组 list2.a 的元素值，那么也将改变数组 list1.a 的元素值，这并不是我们所期望的。我们常常希望赋值运算符与右边变量的副本没有任何关系。解决这个问题方法就是重载赋值运算符，使之按照我们希望的方式操作类 PFArrayD 的对象，这将在示例 10.10 和 10.11 中演示。

示例 10.11 中赋值运算符的重载如下所示：

```
PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
{
    if (capacity != rightSide.capacity)
    {
        delete [] a;
        a = new double [rightSide.capacity];
    }

    capacity = rightSide.capacity;
    used = rightSide.used;
    for (int i = 0; i < used; i++)
        a[i] = rightSide.a[i];

    return *this ;
}
```

重载赋值运算符时，它必须作为类的成员而不能是类的友元存在。这也是上面的定义中只有一个参数的原因。假如有下面的代码：

```
list1 = list2;
```

在上面的语句中，list1 是调用对象，而 list2 是赋值运算符 = 的参数。

需要注意的是，程序将会检查两个对象的成员 capacity 是否相等，如果不等，那么赋值运算符左边的对象（即调用对象）的数组成员变量将会被 delete 运算符删除，然后使用 new 运算符来创建一个和右边对象容量相同的新数组。这不但能保证赋值运算符左边的对象包含一个正确大小的数组，更重要的作用是：如果赋值运算符两边是同一个对象，可以保证其中的成员变量 a 所指向的数组不会被删除。为了体现重要性，假定赋值运算符的重载定义如下：

```
// 一个有 bug 的版本。
PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
{
    delete [] a;
```



```

a = new double [rightSide.capacity];

capacity = rightSide.capacity;
used = rightSide.used;
for (int i = 0; i < used; i++)
    a[i] = rightSide.a[i];

return *this ;
}

```

当赋值运算符两边是同一个对象时，这个版本的重载定义就会出现。例如使用下面的语句：

```
myList = myList;
```

当执行赋值运算时，首先执行的语句是：

```
delete [] a ;
```

由于调用对象是 myList，所以上面的语句等价于：

```
delete [] myList.a;
```

经过上面的操作，指针 myList.a 就变成不确定的随机值。赋值破坏了对象 myList 的内存结构，而如果使用示例 10.11 中给出的重载定义，就不会有这样的出现问题。

### 浅拷贝和深拷贝

在一个赋值运算符的重载或者拷贝构造函数的定义中，如果函数的代码仅仅是将成员变量的值从一个对象中拷贝到另外一个对象中，这种方式叫作浅拷贝。默认的赋值运算符和默认的拷贝构造函数执行的都是浅拷贝。如果类中不包含指针或者动态分配的数据，浅拷贝是完全没有问题的。但是如果类中包含指向动态数组（或者其他动态数据结构）的成员变量，那么一般情况下仅仅进行浅拷贝是不够的，这时，每一个指针成员变量所指向的内容都应该产生一个副本，这样才能得到一个独立的对象副本。在示例 10.11 中那样的情况下，这种拷贝方式叫作深拷贝，我们在重载赋值运算符或者定义一个拷贝构造函数时，应该进行深拷贝操作。

### 析构函数

使用动态变量时存在一个问题：除非在程序中使用合适的 delete 语句，否则动态变量是不会自动销毁的。哪怕是一个局部指针变量指向的动态变量，当函数调用结束时，只有局部指针变量会自动销毁，而该指针指向的动态变量在不使用 delete 语句的情况下是不会自动销毁的。如果在程序中不使用 delete 来销毁动态分配的变量，动态变量会始终占据分配的内存空间，这可能会导致程序耗尽自由存储中的内存而结束。而且如果类的具体执行过程之中出现了动态变量，那么使用该类的程序员不会知道该动态变量的存在，也就无法显式地调用 delete 销毁它们。实际上，由于数据成

员一般是私有成员，因此程序员就算知道该动态变量的存在，也无法访问指针变量然后对它们做 delete 操作。为了解决这个问题，C++ 中有一种名为析构函数的特殊函数。

#### 析构函数

**析构函数**是一个特殊的成员函数，当一个类的对象将结束自己的生命周期时会自动调用类的析构函数。如果有一个局部变量，该变量是一个包含析构函数的类的对象，那么当函数调用结束时，这个类的析构函数将会被自动调用。如果析构函数是定义好的，那么该析构函数将会调用 delete 来销毁所有的本对象创建的动态分配的数据。具体的操作可能是通过调用一次或者多次 delete 实现的。虽然我们可能还会希望析构函数能够同时执行其他的善后工作，但释放动态分配的内存空间是析构函数最主要的工作。

#### 析构函数 名字

在示例 10.10 中，成员函数 ~PFFArrayD 是类 PFFArrayD 的析构函数。与构造函数类似，析构函数也具有与类相同的名字，但析构函数名字前面还有一个符号，以便告知编译器它是析构函数而不是构造函数。和构造函数一样，析构函数也没有返回值类型的，也不是 void 类型函数。和构造函数不同的是，析构函数不能有参数。此外，一个类只能有一个析构函数，不能重载类的析构函数。除了上面所说的区别外，析构函数的定义与其他成员函数是一样的。

注意示例 10.11 中析构函数 ~PFFArrayD 的定义，~PFFArrayD 调用了 delete 来删除成员指针变量 a 所指向的动态数组。在示例 10.12 程序中的函数 testPFFArrayD，其中局部变量 temp 包含了一个它的成员变量 temp.a 所指向的动态数组。如果类 PFFArrayD 中没有定义析构函数，那么函数 testPFFArrayD 的调用结束时，尽管该动态数组已经毫无用处了，但它仍将占据分配给它的内存空间。而且程序中的 do-while 循环语句每执行一次，就会产生一个无用的动态数组占据内存。如果循环次数足够多，那么它将会耗尽所有自由存储中的可分配内存，最终将导致程序非正常结束。

#### 析构函数

析构函数是一个特殊的成员函数，当一个类的对象结束了自己的生命周期时，析构函数将会被自动调用。这意味着如果一个类的对象是一个函数中的局部变量，那么该函数执行结束时析构函数就会被自动调用。析构函数用于删除该对象创建的所有动态分配的变量，将这些动态变量占用的内存释放，以便重新使用。与此同时，析构函数也可以执行一些其他的清理工作。析构函数的名字由符号“~”和类名组成，类名紧跟在符号“~”之后。

#### 拷贝构造函数

#### 拷贝构造 函数

**拷贝构造函数**是一个只包含一个参数的构造函数，该参数的类型是该拷贝构造函数所在类的类型。拷贝构造函数的参数必须是一个引用参数，并且需要使用 const 来修饰。除此以外，拷贝构造函数可以像其他构造函数一样定义和使用。假如一个使用示例 10.10 中定义类 PFFArrayD 的程序包含下面的代码：

```
PFFArrayD b(20);
```

```
for (int i = 0; i < 20; i++)
    b.addElement(i);
PFArrayD temp(b); // 调用拷贝构造函数初始化对象。
```

上面的对象 `b` 是通过包含一个 `int` 参数的构造函数初始化的。而对象 `temp` 则是通过带有一个 `const PFArrayD&` 类型参数的构造函数来初始化的，拷贝构造函数就如同其他构造函数一样被调用。

应该为类定义一个拷贝构造函数，以便在使用已经赋值的对象来初始化该类的对象时能产生一个完整的独立于实参的对象副本。因此在下面的声明中：

```
PFArrayD temp(b);
```

成员变量 `temp.a` 的值不会简单地被设置为变量 `b.a` 的值，这样做只会导致两个指针变量指向同一个动态数组。拷贝构造函数的定义在示例 10.11 中有所演示。在演示的拷贝构造函数的定义中，创建了一个新的动态数组，并将原来的动态数组中所有的内容拷贝到新的动态数组中。因此在上面的声明中，`temp` 被初始化后数组成员变量和 `b` 的数组成员变量是不一样的。这两个数组成员变量 `temp.a` 和 `b.a` 都包含相同的 `double` 类型元素，但是当数组成员变量的值被改变时，却不会影响到另外一个。因此，任何对 `temp` 的修改都不会影响到对象 `b`。

我们已经看到，拷贝构造函数可以像其他构造函数那样被调用。在某些其他情况下，拷贝构造函数会被自动调用。简单总结起来就是，当 C++ 需要产生一个对象的副本时，就会自动调用拷贝构造函数。在下面的三种情况下拷贝构造函数都会被自动调用：

1. 声明一个类的对象时，使用其他已存在的对象来初始化。（此时拷贝构造函数的调用和其他构造函数一样。）
2. 当函数的返回值数据类型是一个类的时候。
3. 当函数的传值参数的数据类型是类的时候。这种情况下参数传递的方式由拷贝构造函数的定义决定。

如果没有为类定义拷贝构造函数，C++ 将会自动为类产生一个拷贝构造函数。不过这个默认的拷贝构造函数只能简单地拷贝成员变量的值，而对于那些包含指针和动态数据的类来说会有很多问题。因此，如果类的成员变量中包含指针、动态数组或者其他动态数据时，就应该自己为类定义拷贝构造函数。

要弄清楚为什么需要拷贝构造函数，首先来看一看如果类 `PFArrayD` 中没有定义拷贝构造函数将会产生什么影响。如果在类 `PFArrayD` 中没有定义拷贝构造函数，并且在某个函数定义中使用了传值参数，例如：

```
void showPFArrayD(PFArrayD parameter)
{
    cout << "The first value is:"
          << parameter[0] << endl;
}
```

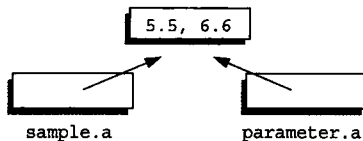
考虑下面调用函数的代码：

```
PFArrayD sample(2);
sample.addElement(5.5);
sample.addElement(6.6);
```

为什么需要  
拷贝构造  
函数

```
showPArrayD(sample);
cout << "After call: " << sample[0] << endl;
```

由于 PArrayD 类没有定义拷贝构造函数，因此只能使用默认的拷贝构造函数将类中的成员变量简单地拷贝。当函数调用执行时，sample 的值被拷贝到局部变量 parameter 中，因此 parameter.a 就等于 sample.a。由于它们是指针变量，因此，在函数调用时，parameter.a 和 sample.a 会指向同一个动态数组，下面是一个示例：



函数调用结束后 PArrayD 类的析构函数将被自动调用，用来释放对象 parameter 所占用的内存，而析构函数的定义中有下面的语句：

```
delete[] a;
```

由于析构函数是对象 parameter 调用的，所以上面的语句和下面的语句是等价的：

```
delete[] parameter.a;
```

函数调用结束后，上面图示的情况就变成下面的样子：



由于 sample.a 和 parameter.a 指向的是同一个动态数组，删除 parameter.a 就等于删除了 sample.a，所以当程序运行到下面一条语句时，sample.a 已经是一个不确定的随机值了：

```
cout << "After call: " << sample[0] << endl;
```

因此上面 cout 语句的输出也是不确定的。也许碰巧 cout 语句会输出我们想要的结果，但由于 sample.a 是不确定的，所以上面的语句早晚会出现问题。另外，若对象 sample 是某个函数中的局部变量，也会产生问题。当 sample 所在的函数调用结束时，sample 的析构函数将会被自动调用，而析构函数的调用就等价于下面的语句：

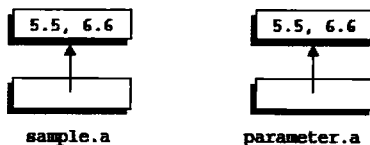
```
delete [] sample.a;
```

但是 sample.a 所指向的动态数组在之前已经被删除了，而现在操作系统试图第二次删除它。两次调用 delete 来删除同一个动态数组（或者其他使用 new 创建的变量）将会产生严重的系统错误，这将导致整个程序崩溃。

上面就是类没有定义拷贝构造函数所导致的后果。幸运的是，在前面 PArrayD 类的定义中，已经给出了拷贝构造函数的定义。因此，在下面的函数调用中，拷贝构造函数会被自动调用：

```
showPFArrayD(sample);
```

拷贝构造函数的定义决定了实参 `sample` 的值按照何种方式传递给传值参数 `parameter`。参数传递之后二者的情况如下：



从上面的图示中可以看出，二者是完全独立的数据。因此任何对 `parameter.a` 的修改都不会影响到实参 `sample`，在调用析构函数时也不会有问题。即使在 `parameter` 调用了析构函数后 `sample` 也调用析构函数，二者所删除的动态数组也不是同一份数据。

#### 返回值

当一个函数返回某个类类型的返回值时，拷贝构造函数将会被自动调用，将拷贝一份 `return` 语句指定的值。如果该类没有定义拷贝构造函数，那么就会出现和上面所提到的传值参数类似的问题。

当你需要一  
个拷贝构造  
函数时  
赋值语句

到底什么时候需要自己定义拷贝构造函数呢？如果类定义中存在指针成员及使用 `new` 运算符所产生的动态分配内存，就需要自己定义拷贝构造函数。那些不包含指针成员及动态分配内存的类则无须定义拷贝构造函数。

与普遍的猜测刚好相反，在使用赋值运算符将一个对象的值传递给另外一个对象时，不会调用拷贝构造函数。<sup>6</sup> 如果我们不希望使用默认的赋值运算符，就可以像示例 10.10 和示例 10.11 那样重载赋值运算符的定义。

### 拷贝构造函数

拷贝构造函数是只有一个引用类型参数的构造函数，而且该引用参数的类型和该类的类型相同。拷贝构造函数的参数必须是引用传递参数，而且通常也是一个常量参数（即使用 `const` 修饰的参数）。当函数返回某个类类型的返回值时，拷贝构造函数将会被自动调用。此外，在向实参类型是类类型而且参数是传值参数的函数传递参数时，也会自动调用拷贝构造函数。拷贝构造函数也可以像其他构造函数一样使用。

所有包含指针成员并使用 `new` 运算符分配内存的类都应该定义拷贝构造函数。

<sup>6</sup> 在C++中，初始化和赋值是有区别的。初始化会使用拷贝构造函数创建一个新的对象，而赋值操作是访问并修改一个已有的对象，以使它成为赋值符右边的值的相同副本。

### 三大条

拷贝构造函数、赋值运算符及析构函数可以被称作三大条。专家们说如果类的定义中需要三者中的任何一个，那么这三个都是必要的。如果其中任何一个没有定义，那么编译器将会产生默认的版本，但默认的版本可能不会按照我们所期望的那样正常工作。因此最好自己定义它们。编译器自动产生的拷贝构造函数和赋值运算符对于只包含基本类型成员变量（比如 int 和 double 等）的情况可以正常工作。但对于那些包含指针成员并使用 new 运算符分配内存的成员的类来说，一定要自己定义拷贝构造函数。

### 自测练习

13. 如果有一个名为 MyClass 的类有构造函数，那么构造函数的名字是什么？如果该类定义了析构函数，那么析构函数的名字又是什么？
14. 如果将示例 10.11 中的析构函数定义改写为下面的函数，那么示例 10.12 中程序的运行结果会有什么变化？

```
PFArrayD::~PFArrayD()
{
    cout << "\nGood-bye cruel world! The short life of\n"
          << "this dynamic array is about to end.\n";
    delete [] a;
}
```

15. 下面是类 PFArrayD 的拷贝构造函数定义中的第一行。其中标识符 PFArrayD 一共出现了三次，每一次的意思都有所不同。这三个标识符的意思分别是什么？

```
PFArrayD::PFArrayD(const PFArrayD& pfaObject)
```

16. 回答下面关于析构函数的问题。
  - a. 什么叫作析构函数？其名字的规范是什么？
  - b. 何时将会调用析构函数？
  - c. 析构函数实际上做了什么工作？
  - d. 析构函数应该完成什么工作？
17.
  - a. 详细解释为什么在只有内置类型数据时，不需要重载赋值运算符。
  - b. 为什么类只有内置类型数据时不需要重新定义拷贝构造函数？
  - c. 为什么类只有内置类型数据时不需要重新定义析构函数？

## 本章小结

- 指针是一个内存地址，它提供了一种额外的变量的标示方法：通过记录变量在计算机内存中的地址来标示变量。
- 动态变量（也叫作动态分配变量）是在程序运行时创建（和销毁）的变量。
- 用来保存动态变量的内存存在计算机内存中是一类特殊的内存，叫作自由存储（或自由存储区）。当程序中不再使用一个动态变量时，其所占据的内存空间可以被释放，然后放回自由存储中供其他程序重复使用。释放内存是通过 delete 语句实现的。
- 动态分配数组（也简称动态数组）是可以在程序运行过程中改变大小的数组。一个动态数组可以被看作是动态变量的数组类型。
- 析构函数是类的一个特殊成员函数。当一个类的对象的生命周期即将结束时，析构函数将被自动调用。析构函数的主要用途是用来释放该对象所占据的内存空间，以便这些内存能够被再次使用。
- 拷贝构造函数是包含一个和该类类型相同的引用传递参数的构造函数。如果定义了拷贝构造函数，那么函数返回某个类的值时，或者在将某个类对象作为传值参数时，都将自动调用拷贝构造函数。任何包含指针成员并使用 new 运算符分配空间的类都应该定义拷贝构造函数。
- 重载赋值运算符时，重载的运算符必须作为类的成员存在。必须保证当赋值运算符两边是同一个对象时，重载后的赋值运算符仍然能够正常使用。

## 自测练习答案

1. 指针是一个变量的内存地址。

```
2. int *p; // 这是声明一个 int 型变量的指针。
   *p=17; // 这里，* 是解引用运算符。该语句将 17 赋值给 p 所指向的内存单元。
   void func (int* p); // 将 p 声明为指针类型的参数。
```

```
3. 10 20
   20 20
   30 30
```

如果将 \*p1=30; 替换为 \*p2=30;，运行结果不变。

```
4. 10 20
   20 20
   30 20
```

5. 对于初学者或者是在粗心大意的情况下，该语句好像是声明了两个指向 int 的指针，即 int\*。然而不幸的是，\* 号是与标识符一起使用的，而不是和类型名（即，不是和 int）一起使用的。语句声明的结果是：intPtr1 是一个 int 指针，而 intPtr2 则是一个普通的 int 型变量。

6. `delete p;`
7. `typedef double* NumberPtr;  
NumberPtr myPoint;`
8. `new` 运算符使用其参数指定的类型，从堆中为该类型的变量分配存储空间。如果堆中有足够的内存空间，那么 `new` 运算符将返回指向所分配的内存空间的指针。如果堆中没有足够的内存空间，那么 `new` 运算符将返回 `NULL`，或者终止程序，具体的行为取决于我们所使用的编译器。
9. `typedef char* CharArray;`
10. `cout << "Enter 10 integers:\n";  
for (int i = 0; i < 10; i++)  
cin >> entry[i];`
11. `delete [ ] entry;`
12. 0 1 2 3 4 5 6 7 8 9
13. 构造函数的名字是 `MyClass`，与类的名字相同。而析构函数的名字为 `~MyClass`。
14. 运行结果将做如下更改：
- ```
This program tests the class PFArrayD.
Enter capacity of this super array: 10
Enter up to 10 nonnegative numbers.
Place a negative number at the end.
1.1
2.2
3.3
4.4
-1
You entered the following 4 numbers:
1.1 2.2 3.3 4.4
(plus a sentinel value.)
Good-bye cruel world! The short life of
this dynamic array is about to end.
Test again? (y/n) n
```
15. 在作用域符 “`::`” 之前的 `PFArrayD` 是类的名字，而其后的 `PFArrayD` 是成员函数的名字（记住，构造函数是与类名相同的成员函数）。而圆括号中的 `PFArrayD` 则是参数 `pfaObject` 的类型。
16. a. 析构函数是类的成员函数。析构函数的名字以符号 `~` 开头，紧跟着类的名字。  
b. 当某个类的对象超出其作用域范围时，析构函数被自动调用。  
c. 析构函数可以完成任何程序员编写的工作。  
d. 析构函数一般用来删除类的构造函数为动态变量分配的内存空间。析构函数也用来完成其他一些清除工作。
17. 对于赋值运算符和拷贝构造函数，如果只有内置的数据类型，那么默认的拷贝机制正好是我们所期望的，因此无须重载。对于析构函数来说，如果只有内置的数据类型，那么就没有动态分配的内存（没有指针），因此默认的不做任何工作的析构函数也是我们所期望的。



## 编程练习

1. 重新阅读示例 10.9 中的代码。请用该示例中所使用的思想编写一个名为 `TwoD` 的类，来实现 `double` 类型的二维动态数组。你应该声明一个 `double` 类型的指针作为私有成员，指向动态数组，同时还应该包含两个 `int` 型值 `MaxRows` 和 `MaxCols`。

应该提供一个默认的构造函数，使二维动态数组有默认的行数和默认的列数。还需包含一个带参数的构造函数，允许程序员指定动态数组的行数和列数。

另外，还应该提供一个 `void` 类型的成员函数，用来设定数组中特定的行和列的元素值。此外，还要包含一个用来获取数组中特定的行和列的元素值的函数，获取的值为 `double` 类型。

注意：为二维数组重载“[]”运算符几乎是很难甚至是不可能的事，因此，这里我们简单地使用取值和赋值函数来完成“[]”的功能。

将“+”运算符作为友元进行重载，使之能够用于两个二维数组的加法运算。此运算符返回的 `TwoD` 对象的第 *i* 行和第 *j* 列的元素，是运算符左边对象的第 *i* 行和第 *j* 列的元素与运算符右边对象的第 *i* 行和第 *j* 列的元素的和。

为该类定义拷贝构造函数，重载赋值运算符，并定义析构函数。

将该类所有不改变数据成员的成员函数声明为 `const` 成员。

2. 使用动态数组，实现一个包含多项式加法、减法和乘法的多项式类。

讨论：多项式中的变量仅仅是用来保存各项的系数的。因此，多项式关注的焦点是系数数组和对应的指数。考虑下面的多项式：

$$x^2 + x + 1$$

其中  $x^2$  项到哪里去了？一个简单的实现多项式类的方法是使用一个 `double` 类型的数组来存储多项式的系数。而数组的索引就是与该项对应的指数。如果多项式中不存在该项，那么只需简单地将系数设为 0。

这里有一种用来表示缺少许多项的高阶多项式的技术，被称为系数矩阵技术。除非你已经掌握了该技术或者能够很快学会，否则在这里最好不要使用该技术。

定义默认的构造函数、拷贝构造函数及带有参数的构造函数，使我们可以构造任意的多项式。

提供重载的赋值运算符及析构函数。

提供下面的操作：

多项式 + 多项式，常量 + 多项式，多项式 + 常量，

多项式 - 多项式，常量 - 多项式，多项式 - 常量，

多项式 \* 多项式，常量 \* 多项式，多项式 \* 常量。

提供用于分配系数和提取系数的函数，以指数为索引。

提供计算多项式值的函数，多项式值为 `double` 类型。

你可以自己决定这些函数的实现方式，即成员函数、友元函数还是独立的普通函数。

- 编写一个程序，输入为用户键入的 C 字符串，将字符串的内容倒转。程序中应当用到两个指针。“头”指针应该被设置为字符串首字符的地址，“尾”指针应该被设置为字符串末字符（即空字符前的字符）的地址。程序应当交换这些指针引用的字符，递增“头”指针使它指向下一个字符，递减“尾”指针，使它指向倒数第二个字符，等等。直到所有字符都交换完，整个字符串都倒转为止。
- 创建一个名为 `Student` 的类，该类包含三个成员变量：

`name` —— 一个存储学生名字的字符串。

`numClasses` —— 一个整数，记录了学生当前注册的课程数。

`classList` —— 一个动态字符串数组，用于存储学生注册的课程的名称。

为 `Student` 类编写合适的构造函数、取值函数和赋值函数，以及以下内容：

- 一个函数，输入为来自用户的值，包括课程名称列表。此函数应当支持输入任意数量的课程。
- 一个函数，输出所有课程的名称和列表。
- 一个函数，用于将课程数清零并清空 `classList` 列表。
- 一个重载赋值运算符，可正确地制作课程列表的新副本。
- 一个析构函数，用于释放已分配的所有内存。

编写一个 `main` 函数来测试上述所有函数。

- 在第 7 章的编程练习 8 和第 5 章的编程练习 7 的基础上，编写一个生成某班学生学分直方图的程序。程序要对应每个学生有一个整型的输入作为学生的学分，学分保存在一个 `vector` 中。学分的输入应该一直持续到用户输入 -1。程序会扫描保存学分的 `vector`，计算出直方图。在计算过程中学分的最小值是 0，最大值要在用户输入阶段获取。用一个动态数组保存直方图并将直方图输出到终端。
- 使用动态数组时可能会遇到一个问题，就是一旦使用 `new` 创建了数组，数组的大小就无法再改变，毕竟数组无法像 `vector` 一样动态地添加或删除元素。现在要求你创建一个名叫 `DynamicStringArray` 的类，这个类应该包含若干成员函数，可以完成与 `vector` 相似的操作。这个类应该具备如下的成员：
  - 私有成员变量 `dynamicArray`，一个 `string` 类型的动态数组。
  - 私有成员变量 `size`，动态数组的大小。
  - 默认构造函数，可以将动态数组初始化为 `NULL`，`size` 初始化为 0。

- 一个能返回 `size` 的成员函数。
- 一个名为 `addEntry` 的成员函数，接收一个输入的 `string`，创建一个比 `dynamicArray` 长度加 1 的新动态数组，将 `dynamicArray` 中的元素都拷贝到新数组中，将接收的新 `string` 添加到新数组的末尾，修改 `size` 的值，删除旧数组 `dynamicArray`，重新设置 `dynamicArray` 为新数组。
- 一个名为 `deleteEntry` 的成员函数，接收一个输入的 `string`，在 `dynamicArray` 中搜索这个字符串。如果没有找到就返回 `false`；如果找到了，创建一个长度比 `dynamicArray` 小 1 的新动态数组，将除了 `string` 以外的其他元素拷贝到新数组中，修改 `size` 的值，删除 `dynamicArray`，返回 `true`。
- 一个叫作 `getEntry` 的成员函数，接收一个整型参数作为索引值，返回 `dynamicArray` 中索引位置上的元素值。如果输入的索引值超过了 `dynamicArray` 的范围，则返回 `NULL`。
- 一个拷贝构造函数可以将动态数组深拷贝。
- 重载赋值运算符使得动态数组能正确地传递给目标对象。
- 一个析构函数可以正确地释放动态数组的内存空间。

在一个合适的测试程序中嵌入你写的类。



# 分散编译和命名空间



## 11.1 分散编译 392

封装回顾 392

头文件和实现文件 393

示例：DigitalTime类 399

提示：可重用的组件 401

使用#ifndef 401

提示：定义其他库 403

## 11.2 命名空间 404

命名空间和using指令 404

创建一个命名空间 406

using声明 409

限定名称 410

提示：为命名空间取名 412

示例：一个定义在命名空间中的类 412

无名称的命名空间 413

陷阱：混淆全局命名空间和无名称命名空间 418

提示：用无名称命名空间代替static修饰符 419

提示：隐藏帮助函数 419

嵌套命名空间 420

提示：应该如何指明使用的命名空间 420

# 第 11 章 分散编译和命名空间

把我的书斋里那些比一个公国更宝贵的书带给我。

莎士比亚,《暴风雨》

## 概述

本章将介绍两个关于如何组织 C++ 代码的主题,我们将探讨如何合理地将程序代码分割成不同的部分。11.1 节将介绍分散编译,这里我们要将一段 C++ 程序代码分割成不同的文件,这样,当程序中有一部分需要修改的时候,需要重新编译的源程序文件只有修改过的那一部分,而且分割的各个部分也可以被其他程序方便地再次使用。

11.2 节我们讨论命名空间,在第 1 章中我们已经对它做过简单的介绍。命名空间通过对相同的类名称、函数名称或其他名称做不同的解释,使这些名称可以重复使用。被命名空间分割的不同代码段中可以存在相同的名称,而这些名称在不同的空间中表达不同的含义。这使得常用的局部变量名称可以被更频繁地使用。

本章的内容可以提前阅读,学习本章不需要有第 5 章数组、第 9 章字符串、第 10 章指针和动态数组的知识,以及 7.3 节中关于向量的知识。

## 11.1 分散编译

能化解此事的只有你的“如果”,一句高尚的“如果”。

莎士比亚,《皆大欢喜》

C++ 允许一个程序分成不同的部分被写在不同的源文件中,这些源文件可以被分散编译,最后在程序执行前(或者执行时),分散的编译结果可以被连接起来。可以将一个类的定义(以及和它相关的函数的定义)放在独立于使用该类的程序的源文件中。我们可以建立一个类库,使得许多其他的程序段可以方便地使用这个类。这个类只需要被编译一次,就可以被许多程序使用,它就像那些库(例如头文件 `iostream` 和 `cstdlib` 所指定的库)中预先定义好的类一样。而且,我们还可以将定义一个类的内容分开写在两个文件中,这样,我们就可以将一个类中用于说明的部分和实现的部分分开。如果我们需要对类的某些实现做修改,那我们只需修改包含这部分实现的源文件。那些使用该类的文件则无须更改,甚至无须重新编译。本节将介绍如何进行这种类的分散编译。

## 封装回顾

封装的原则告诉我们将类的说明部分和具体实现部分分离的方法。这种分离使得改变类的实现部分时,无须对使用该类的程序进行任何修改。实现这种分离的方法可以总结为下面三条:

1. 所有的成员变量设置为类的私有成员。
2. 所有与该类相关的基本操作都定义为该类的公有成员函数、友元函数、普通函数，或者运算符重载。类的定义、函数和运算符重载的声明放在一起，加上相关的注释被称为类的接口部分。注释中应该详细说明这些函数和重载的运算符的功能。
3. 让使用这个类的程序员不能获得这些基本操作实现的细节。实现包括函数的定义和运算符重载的定义（也包括那些帮助函数和其他函数定义所需要的项）。

C++ 中要遵循上面三条规则的最好办法就是将类的接口和实现放在不同的源文件中。和我们预想的一样，包含了接口部分的源文件叫作接口文件，而包含了类的实现的部分叫作实现文件。在不同的 C++ 版本中，设置以及编译这些文件的具体细节可能会有所不同，但是基本的结构是相同的。其实在所有的系统中，这些结构最终都将被编译成相同的结果，区别只在于不同的系统中编译和连接这些文件的指令不同。关于这些文件的具体细节将在下一节中讨论。

一个典型的类会包含私有成员变量，我们之前提到了将类的接口和类的实现分开放在不同的文件中，这种情况下类的私有成员（比如私有成员函数）会带来一些麻烦。根据我们之前的定义，类的公有成员属于接口，而私有成员却属于实现，由于 C++ 不允许将一个类的定义拆开放在两个文件中，这里就出现了问题。这种情况下唯一的办法就是将整个类的定义放在接口文件中，由于使用这个类的程序员仍然不能访问类的私有成员，所以私有成员应有的隐蔽性仍然有效。

## 头文件和实现文件

示例 11.1 是类 `DigitalTime` 的接口文件。`DigitalTime` 类用来表示一天中的某一个时间，比如 9:30。接口文件中只有公有成员属于类的接口部分。虽然接口文件中也有私有成员部分，但是它们都属于类的实现部分，它们前面的关键字 `private:` 提示了哪些成员是不属于公有接口的私有成员。程序员使用 `DigitalTime` 类的时候只需要从文件的开头部分和类定义中公有部分的注释中获取信息。文件开头部分的注释提到了这里的时间使用 24 小时制，所以下午 1:30 应该写为 13:30。其他的使用细节都在类的公有成员函数的注释中给出了。

我们将这些接口的代码放在一个名叫 `dtime.h` 的文件中，后缀 `.h` 表示这是一个头文件。接口文件都应该是头文件，所以都应该以后缀名 `.h` 结尾。任何使用类 `DigitalTime` 的程序都应该使用 `include` 指令来包含这个头文件，代码如下：

```
#include "dtime.h"
```

使用 `include` 指令必须标明使用的头文件是系统的库提供的还是自己编写的，系统库中预定义的头文件的文件名应该用尖括号，比如 `<iostream>`。自己编写的头文件应该使用双引号，比如 `"dtime.h"`。这样做是为了让编译器知道该去哪里寻找头文件。对于写在尖括号里的文件，编译器将在所有的系统预定义的头文件路径中搜索头文件。对于写在双引号中的文件，编译器则搜索所有的自定义头文件路径。

任何使用 `DigitalTime` 类的程序都应该使用上面的 `include` 语句包含头文件 `dtime.h`。这可以使程序顺利编译成功，但还不能运行。编写完成成员函数和运算符重

接口  
实现

接口文件  
实现文件

私有成员变量  
是实现的一  
部分

头文件

include

## 文件名

载的定义之后，程序才能正常执行。通常我们将这些函数和运算符的定义放在另一个叫作实现文件的文件中。虽然编译器对实现文件的名字没有要求，但习惯上我们让它的文件名与头文件相同，但后缀名不同。比如，我们将类 `DigitalTime` 的接口部分放在名为 `dttime.h` 的文件中，将实现部分放在名为 `dttime.cpp` 的文件中。实现文件的后缀名取决于我们使用的 C++ 编译器的版本，和其他 C++ 文件的后缀名一样（或者是 `.cc`、`.cxx` 和 `.hxx`）。示例 11.2 将给出 `DigitalTime` 类的实现，在介绍完这些不同文件之间的联系之后，我们来讨论示例 11.2 中类定义的细节。

### 示例 11.1 `DigitalTime` 类的接口文件

```

1  // 这是一个名为 dttime.h 的头文件，是类 DigitalTime 的接口。
2  // 这个类表示一个 24 小时制的时间。
3  // 比如 9:30 表示上午 9:30，而 14:45 表示下午 2:45。
4  #include <iostream>
5  using namespace std;
6  class DigitalTime
7  {
8  public:
9      DigitalTime(int theHour, int theMinute);
10     DigitalTime( );
11     // 将时间的值初始化为 0:00，表示午夜。

12     int getHour( ) const;
13     int getMinute( ) const;
14     void advance(int minutesAdded);
15     // 将时间设置为 minutesAdded 分钟之后。

16     void advance(int hoursAdded, int minutesAdded);
17     // 将时间设置为 hoursAdded 小时 minutesAdded 分钟之后。

18     friend bool operator ==(const DigitalTime& time1,
19                             const DigitalTime& time2);

20     friend istream& operator >>(istream& ins, DigitalTime& theObject);

21     friend ostream& operator <<(ostream& outs, const DigitalTime&
22     theObject);
23 private:
24     int hour;
25     int minute;
26     static void readHour(int& theHour);
27     // 前提条件：
28     // 接下来的输入是一个时间，格式就像 9:45 或者 14:45。
29     // 运行结果：theHour 被设置为读入时间的小时数。
30     // 冒号会被忽略，接下来读入的是时间的分钟部分。

31     static void readMinute(int& theMinute);
32     // readHour 函数读入时间的小时部分后，读入时间的分钟部分。

33     static int digitToInt(char c);

```

这些成员变量和帮助函数是实现的一部分。它们不是接口，关键字 `private` 指出了它们并不是公共接口的一部分。

```

33     // 前提条件：c 是一个 0 ~ 9 的数字。
34     // 返回数字字符的整型值，比如 digitToInt('3') 将返回 3。
35
36 };

```

---

## 示例 11.2 DigitalTime 类的实现文件

---

```

1  // 这是 DigitalTime 类的实现文件 dtime.cpp 的内容。
2  // DigitalTime 类的接口部分在头文件 dtime.h 中。
3  #include <iostream>
4  #include <cctype>
5  #include <cstdlib>
6  using namespace std;
7  #include "dtime.h"

8  // 使用 iostream 和 cstdlib。
9  DigitalTime::DigitalTime(int theHour, int theMinute)
10 {
11     if (theHour < 0 || theHour > 24 || theMinute < 0 || theMinute > 59)
12     {
13         cout << "Illegal argument to DigitalTime constructor.";
14         exit(1);
15     }
16     else
17     {
18         hour = theHour;
19         minute = theMinute;
20     }

21     if (hour == 24)
22         hour = 0; // 午夜时间初始化为 0:00。
23 }

24 DigitalTime::DigitalTime()
25 {
26     hour = 0;
27     minute = 0;
28 }

29 int DigitalTime::getHour() const
30 {
31     return hour;
32 }
33
34 int DigitalTime::getMinute() const
35 {
36     return minute;
37 }

38 void DigitalTime::advance(int minutesAdded)
39 {
40     int grossMinutes = minute + minutesAdded;
41     minute = grossMinutes % 60;

```



```

42     int hourAdjustment = grossMinutes / 60;
43     hour = (hour + hourAdjustment)%24;
44 }

45 void DigitalTime::advance(int hoursAdded, int minutesAdded)
46 {
47     hour = (hour + hoursAdded) % 24;
48     advance(minutesAdded);
49 }

50 bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
51 {
52     return (time1.hour == time2.hour && time1.minute == time2.minute);
53 }

54 // 使用库 iostream.
55 ostream& operator <<(ostream& outs, const DigitalTime& theObject)
56 {
57     outs << theObject.hour << ':';
58     if (theObject.minute < 10)
59         outs << '0';
60     outs << theObject.minute;
61     return outs;
62 }
63
64 // 使用库 iostream.
65 istream& operator >>(istream& ins, DigitalTime& theObject)
66 {
67     DigitalTime::readHour(theObject.hour);
68     DigitalTime::readMinute(theObject.minute);
69     return ins;
70 }

71 int DigitalTime::digitToInt(char c)
72 {
73     return (static_cast<int>(c) - static_cast<int>('0')) ;
74 }

75 // 使用库 iostream, ctype 和 cstdlib.
76 void DigitalTime::readMinute(int& theMinute)
77 {
78     char c1, c2;
79     cin >> c1 >> c2;

80     if (!(isdigit(c1) && isdigit(c2)))
81     {
82         cout << "Error: illegal input to readMinute\n";
83         exit(1);
84     }
85     theMinute = digitToInt(c1)*10 + digitToInt(c2);

86     if (theMinute < 0 || theMinute > 59)
87     {
88         cout << "Error: illegal input to readMinute\n";
89         exit(1);

```

```

90     }
91 }
92
93 // 使用库 iostream, ctype 和 cstdlib.
94 void DigitalTime::readHour(int& theHour)
95 {
96     char c1, c2;
97     cin >> c1 >> c2;
98     if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
99     {
100         cout << "Error: illegal input to readHour\n";
101         exit(1);
102     }
103
104     if (isdigit(c1) && c2 == ':')
105     {
106         theHour = DigitalTime::digitToInt(c1);
107     }
108     else // (isdigit(c1) && isdigit(c2))
109     {
110         theHour = DigitalTime::digitToInt(c1)*10
111             + DigitalTime::digitToInt(c2);
112         cin >> c2; // 忽略 ':'.
113         if (c2 != ':')
114         {
115             cout << "Error: illegal input to readHour\n";
116             exit(1);
117         }
118
119         if (theHour == 24)
120             theHour = 0; // 午夜时间初始化为 0:00.
121
122         if ( theHour < 0 || theHour > 23 )
123         {
124             cout << "Error: illegal input to readHour\n";
125             exit(1);
126         }
127     }
128 }

```

任何使用 DigitalTime 类的程序都必须包含这样的 include 指令：

```
#include "dttime.h"
```

应用程序  
文件  
或驱动文件

DigitalTime 类的实现文件和使用它的程序文件都必须使用 include 指令包含其接口文件。程序所在的文件（即包含了 main 函数的文件）通常被称为应用程序文件或者驱动文件。示例 11.3 中包含了一个应用程序文件，这是一个使用 DigitalTime 类的简单例子。

编译并运行  
这个程序

上面这个例子中的完整程序一共包括了三个文件，运行这个程序的具体细节取决于我们使用的操作系统。但是所有的操作系统上，基本的步骤都是一样的。我们须要编译类的实现文件和包含 main 函数的应用程序文件；而接口文件是不需要直接编译的，

接口文件就是示例 11.1 中给出的 `dttime.h` 文件。编译器会认为接口文件已经包含在另外两个文件中了，所以不需要单独编译。回想一下，我们会发现类的实现文件和应用程序文件中都包含了下面的指令：

```
#include "dttime.h"
```

编译器编译程序的时候，会自动调用预处理器将 `include` 指令中的文件 `dttime.h` 内容替换进来。因此，编译器可以编译到 `dttime.h` 中的内容，而不需要再将 `dttime.h` 文件做单独的编译（编译器实际上编译了两遍 `dttime.h` 文件，一次是编译类实现文件的时候，一次是在编译应用程序文件的时候）。这里所说的内容替换只是概念上的替换，编译器会认为 `dttime.h` 文件的内容被拷贝到了所有使用 `include` 指令来包含 `dttime.h` 文件的文件中。但是，如果我们检查编译生成的结果，我们只能发现 `include` 指令，实际上是找不到 `dttime.h` 文件的内容的。

连接  
连接器

类的实现文件和应用程序文件编译完成后，还需要将两个文件连接在一起才能使它们正常工作，我们将这个步骤称作连接。连接的工作是通过一个叫作连接器的工具完成的。连接器使用的细节取决于操作系统。一般情况下程序运行的时候可以自动调用连接器，所以并不需要做特别的工作。文件连接成功后，程序就可以正常运行了。

make  
工程

听起来这似乎是一个相当复杂的过程，但是我们可以找到大量方便的工具，它们可以自动或者半自动地管理这些具体的工作。在任何一个操作系统中，这些具体的工作已经形成了一套常规的流程。在 UNIX 系统中，我们通过 `make` 工具来完成这些工作。而在很多 IDE（Integrated Development Environment，被称为集成开发环境）中，这些相关的文件都被安排在一个被称为工程的位置。

为什么使用  
分开的文件

示例 11.1、示例 11.2 和示例 11.3 实际上包含了一个完整的程序，这个程序被分散放在了三个不同的代码文件中。其实我们也可以将这三个文件的内容完全都放在一个文件中，然后只编译和运行这一个文件就可以了，不必使用 `include` 指令，也不需要连接不同文件的编译结果，但是我们为什么还要如此麻烦地使用三个分开的文件呢？这当然是因为这么做有很多优点。类 `DigitalTime` 的实现文件和接口文件独立于应用程序文件而存在，这使得我们可以在其他应用程序中方便地使用 `DigitalTime` 类，而不需要重写这个类的定义。而且，不论有多少程序使用这个类，它的实现文件只需要被编译一次。还有一个更大的优点是，由于类 `DigitalTime` 的实现文件和接口文件是分开的，当类的实现有改动的时候，我们不必修改使用了这个类的应用程序文件，甚至不需要重新编译它们。如果改动了类的实现文件，我们只需要重新编译这个实现文件，然后将程序所有的编译结果重新连接一遍。这样不但节省了重新编译文件的时间，而且避免了很多重复书写代码的工作。我们可以在许多应用程序中使用这个类，而无须在所有的应用程序中重写这个类的代码。我们可以改变类的实现，而无须改变使用这个类的应用程序。

### 示例 11.3 使用 `DigitalTime` 类的应用程序

```
1 // 这是一个应用文件 timedemo.cpp，演示了如何使用类 DigitalTime。
2 #include <iostream>
3 using namespace std;
4 #include "dttime.h"
```

```

5 int main( )
6 {
7     DigitalTime clock, oldClock;

8     cout << "You may write midnight as either 0:00 or 24:00,\n"
9         << "but I will always write it as 0:00.\n"
10        << "Enter the time in 24-hour notation: ";
11    cin >> clock;

12    oldClock = clock;
13    clock.advance(15);
14    if (clock == oldClock)
15        cout << "Something is wrong.";
16    cout << "You entered " << oldClock << endl;
17    cout << "15 minutes later the time will be "
18        << clock << endl;

19    clock.advance(2, 15);
20    cout << "2 hours and 15 minutes after that\n"
21        << "the time will be "
22        << clock << endl;

23    return 0;
24 }

```

### 示例运行结果

```

You may write midnight as either 0:00 or 24:00,
but I will always write it as 0:00.
Enter the time in 24-hour notation: 11:15
You entered 11:15
15 minutes later the time will be 11:30
2 hours and 15 minutes after that
the time will be 13:45

```

---

### 示例：DigitalTime 类

之前的示例 11.1、示例 11.2 和示例 11.3 描述了如何将一个完整的程序分成三个文件：类 DigitalTime 的接口文件、实现文件和一个使用该类的应用程序文件。现在我们开始讨论类实现的具体细节。下面这个例子中并不包含新的知识，如果你对示例 11.2 中的一些类的实现细节不是很清楚，这个例子会起到一些帮助作用。

类实现的大部分细节都是简洁易懂的，但是仍然有两个需要解释的地方。值得我们注意的是，类的成员函数 advance 被重载了，因此文件中有两个函数的定义。此外，在重载运算符“>>”的代码中出现了两个帮助函数 readHour 和 readMinute，这两个帮助函数的定义中又使用了第三个叫作 digitToInt 的帮助函数。我们来讨论一下这两点。

类 DigitalTime（如示例 11.1 和示例 11.2 中所示）包含两个名称为 advance 的成员函数，其中一个版本只包含一个参数，该参数有一个 int 值来表示应该前进的

时间的分钟数。Advance 的另一个版本有两个参数，一个参数表示小时数，另一个表示分钟数，应该前进的总时间是小时数加上分钟数。值得注意的是，第二个版本的 advance 函数的定义中出现了一个参数版本的 advance。看一下示例 11.2 中包含的两个参数版本的函数定义，先使用参数 hoursAdded 将时间前进若干个小时，然后调用一个参数的 advance 函数让时间前进参数 minuteAdded 指定的分钟数。这看起来好像有些奇怪，但完全是合法的。编译器会认为，这两个重名的 advance 函数实际上是两个不同的函数，只不过有一个共同的名字。

现在我们来看一下帮助函数。帮助函数 readHour 和 readMinute 将用户输入按字符逐个输入，然后将输入的字符转换成整型值存放在成员变量 hour 和 minute 中。由于 readHour 和 readMinute 函数每次只能读入小时数和分钟数的一位，所以它们读入的是字符型值。这样做比直接读入整型值麻烦很多，好处就是使错误检查成为可能，并且能在发现输入不正确的时候输出相应的错误信息。帮助函数 readHour 和 readMinute 中使用了另外一个叫作 digitToInt 的帮助函数。此函数可以将一个数字字符转换成数字，比如可以将 '3' 转换成 3。这个函数的具体实现已经在之前的第 7 章的自测练习题 3 作为答案给出了。

### 总结：类的定义分散在多个文件中

在定义一个类的时候，可以将类的定义和类方法的实现分开放在不同的文件中，这些文件可以独立于任何应用程序被单独编译，然后在多个应用程序中重复使用。类的文件可以按照下面的方法安排：

1. 类的定义放在一个被称为接口文件的头文件中。头文件的后缀名是 .h。接口文件中还可以包含一些其他函数的重载运算符的声明，这些函数和运算符定义了类的一些基本操作，但不会出现在类定义中。另外，我们应该在这里加入一些如何使用这些函数或者运算符的注释。
2. 上面提到的所有函数和运算符（不论是成员函数、友元函数，还是其他的一般函数）的定义都放在一个被称为实现文件的文件中。这个文件必须使用 include 指令来包含上面提到的接口文件。在 include 指令中应该使用双引号来标明接口文件的文件名，如下所示：

```
#include "dttime.h"
```

类的接口文件和实现文件一般具有相同的文件名和不同的后缀名。接口文件的后缀名一般为 .h，而实现文件的后缀名应该与应用程序文件的后缀名相同。如果能让实现文件在其他应用程序中也可以使用，它应该被单独编译。

3. 当需要在一个程序中使用一个类的时候，程序的主要代码（类成员函数以外的函数定义、常量声明等）一般被单独放在一个文件中，这个文件通常被称为应用程序文件或者驱动文件。这个文件同样需要使用 include 指令来包含之前提到过的接口文件。如下所示：

```
#include "dttime.h"
```

应用程序文件与类的实现文件是分开编译的。我们可以编写很多应用程序文件，它们都使用同一个类的接口文件和实现文件。执行一个完整程序时需要将实现文件编译后产生的目标文件和应用程序编译后产生的目标文件连接在一起（在大部分操作系统中，连接一般是自动或者半自动完成的）。

一个程序中如果使用了很多个类，那么程序就拥有很多个接口文件和实现文件，每一个实现文件都需要单独编译。

### 提示：可重用的组件

在几个分开的文件中定义的类可以看作一个软件的组件，它可以在许多不同的程序中重复使用。组件的重用将节省很多精力，因为对于所有要使用这个组件的程序来说，这个组件是不需要重新设计、重新实现、重新测试的，而且已经被重用过的组件肯定比那些仅仅使用过一次的组件可靠很多。其原因有二：首先，对于那些将被反复使用多次的可重用组件，程序员可以花费大量的时间和精力设计和实现；其次，一个组件被使用了很多次，说明它经历了足够多的测试，每次对组件的重用都可以看作是对该组件的一次测试。将组件在不同的程序中多次使用是发现组件中存在的 bug 的最佳方法。■

### 使用#ifndef

前面已经介绍了将一个完整程序的内容分散放在三个文件的方法，其中两个文件是每个类都必需的接口和实现，另外一个文件是应用程序的文件。当然，一个程序也可以被放在多于三个的文件中。比如，这个程序可能使用了很多个类，每个类都有自己的两个文件。假设一个程序包含很多个文件，其中不止一个文件使用了 include 指令包含类的接口文件：

```
#include "dtime.h"
```

这样，一个文件包含了另一个文件，被包含的文件同样可能也包含其他的文件。下面这种情况很容易出现：程序的某个文件中可能不止一次包含了 dtime.h 中类的定义。尽管这些重复的类定义是完全相同的内容，但是 C++ 是不允许一个类被多次定义的。如果在多个工程中使用了一个相同的头文件，想知道在某个文件中是否多次包含了一个类的定义几乎是不可能的。为了避免这种情况发生，C++ 提供了一种方法标记那些将被其他文件包含的代码段：如果一个文件中已经包含了这部分代码，那么将不再引入它。这是一种直观的方法，只是使用时那些标识符有些难以理解。下面通过一个具体的例子介绍一些细节。

```
#define 下面的指令定义了 DTIME_H；
```

```
#define DTIME_H
```

上面的定义表示编译器的预处理器会将 DTIME\_H 加入列表，表示 DTIME\_H 是已被

定义的。这里所说的“定义”可能并不准确，因为 `DTIME_H` 并没有被定义为任何值，而是仅仅被放在列表中。这么做的意义在于我们可以使用另外一个指令检查 `DTIME_H` 是否已经被定义了，同时可以借此判断一段代码段是否会被编译器处理。我们可以使用任何（非关键字）标识符来替代这里的 `DTIME_H`，但是使用什么样的标识符需要遵循一个标准的规则。

下面的指令可以用来检测 `DTIME_H` 是否已被定义。

```
#ifndef DTIME_H
```

如果 `DTIME_H` 已经被定义了，那么在上面这个指令和第一次出现如下这个指令的位置之间的所有代码将被编译器忽略：

```
#endif
```

之前这两条预编译指令可以理解为：如果 `DTIME_H` 没有被定义，那么编译器将从 `#ifndef` 开始直至下一个 `#endif` 之间的所有代码。`#ifndef` 指令中的 `n` 表示否定，即还未定义（这很容易让我们想到还会有一个与之对应的 `#ifdef` 指令，但是很难有机会使用它）。

现在来看一下这样的代码：

```
#ifndef DTIME_H
#define DTIME_H
< 某个类的定义 >
#endif
```

如果上面的代码存在于头文件 `dtime.h` 中，那么无论程序中包含多少次下面的语句：

```
#include "dtime.h"
```

类 `DigitalTime` 仍然被认为只定义了一次。

当程序代码中第一次出现如下的内容时：

```
#include "dtime.h"
```

标记 `DTIME_H` 就被定义了，同时类 `DigitalTime` 也会被定义。如果编译器再次遇到如下的内容：

```
#include "dtime.h"
```

程序将再包含一份 `dtime.h` 文件的内容，但当编译器遇到文件中的：

```
#ifndef DTIME_H
```

由于 `DTIME_H` 已经被定义了，所以 `dtime.h` 中的代码都被编译器忽略，直到遇到下面的指令：

```
#endif
```

因此 `DigitalTime` 不会被重复定义。

在示例 11.4 中，我们重新修改了示例 11.1 中的头文件 `dtime.h`，增加了防止重复定义的指令。使用示例 11.4 中的 `dtime.h` 文件，即使一个程序多次使用 `include` 语句来包含这个头文件，类 `DigitalTime` 的定义也只会出现一次。

```
#include "dtime.h"
```

使用其他的标识符来替代 `DTIME_H` 也是合法的，但是一个约定俗成的方式是将要被包含的头文件的名字全部大写，并用下画线代替文件名中的点。我们应该遵守这种统一的方式，这可以使我们的代码更容易被别人读懂，而且不必记住这个标识符的名字，因为采用这种方式，标识符的名字将完全取决于头文件的文件名。

这些指令不仅可以用在头文件中，用在其他文件中也是合法的，但是本书只在头文件中使用。

#### 示例 11.4 避免类的重复定义

```
1 // 这是头文件 dtime.h, 也是类 DigitalTime 的接口文件。
2 // 这个类的值表示一天中的某个时间, 时间以 24 小时计。
3 // 比如 9:30 表示上午 9:30, 而 14:45 表示下午 2:45。

4 #ifndef DTIME_H
5 #define DTIME_H

6 #include <iostream>
7 using namespace std;

8 class DigitalTime
9 {
10     < 类 DigitalTime 的定义与示例 11.1 中的定义完全相同。 >
11 };
12 #endif //DTIME_H
```

#### #ifndef

可以在头文件（类的接口文件）中使用 `#ifndef` 指令避免一个类（或其他内容）的重复定义，如示例 11.4 所示。如果该头文件被包含了多次，只有第一次的定义是有效的，其他的都将被忽略。

#### 提示：定义其他库

分散编译并非只能用在定义类上。假如我们有一系列需要的相关函数，想要把这些函数组成一个自己设计的库，就可以将函数的声明（原型）以及相关的注释放在一个头文件中，将所有的函数定义放在一个实现文件中，像定义类一样。这样，我们就可以像使用分散编译的类一样使用这个库。■

#### 自测练习

1. 假设要在一个程序中使用自己定义的一个类，而且使用本章中介绍的将类和程序



分开定义在不同文件中的方式，请说明下面的各个部分应该定义在程序的接口文件、实现文件还是应用文件中。

- a. 类的定义。
  - b. 使用这个类的函数的声明，此函数不是类的成员函数，也不是友元函数。
  - c. 操作这个类的运算符重载声明，此运算符不是类的成员，也不是友元。
  - d. 使用这个类的函数的定义，此函数不是类的成员函数，也不是友元函数。
  - e. 一个操作这个类的友元函数的定义。
  - f. 这个类的成员函数的定义。
  - g. 操作这个类的运算符重载定义，此运算符不是类的成员，也不是友元。
  - h. 操作这个类的运算符重载定义，此运算符是类的友元。
  - i. 程序的 main 函数。
2. 下面哪些文件的后缀名是 .h？类的接口文件、类的实现文件、使用该类的应用程序文件。
  3. 当一个类的代码被分开放在多个文件中时，就产生了类的接口文件，还是实现文件？这两种文件是否都需要编译？
  4. 假如使用分开定义的方式定义了一个类，并在一个程序中使用了这个类，当这个类的实现文件被修改后，哪些文件需要重新编译？类的接口文件、类的实现文件，还是应用文件？
  5. 假如修改了示例 11.1 和示例 11.2 中的类 DigitalTime 的实现部分，用一个 int 型私有变量 minutes 代替 hour 和 minute 来表示时间，记录时间从 0:00 开始，记录经过时间的分钟数。比如 1:30 被记作 90 分钟，因为从 0 点到 1:30 总共经过了 90 分钟。无须重写整个程序，指出需要修改的地方并修改之即可。

## 11.2 命名空间

名称有什么重要的呢？玫瑰不叫玫瑰，依然芳香如故。

莎士比亚，《罗密欧与朱丽叶》

命名空间

当一个程序是由不同程序员编写的类和函数组成时，很可能出现不同程序员对不同的事物使用相同名称的情况，命名空间就是来解决这个问题的。命名空间是一个名称定义的集合，比如类的定义和变量的声明等。命名空间可以被打开和关闭，因此，当一个命名空间中有一个名称定义和其他命名空间中的名称发生冲突的时候，就应该关闭一个命名空间。

### 命名空间和 using 指令

之前我们已经使用过 std 命名空间了。std 命名空间包括所有在标准 C++ 库（例如 iostream）中定义的名称。举个例子，我们把下面的代码放在一个文件的开始部分：

```
#include <iostream>
```

`iostream` 库中所有的名称（例如 `cin` 和 `cout` 之类的名称）都定义在 `std` 命名空间中。所以，除非程序指明了要使用 `std` 命名空间，否则程序不会识别这些位于 `std` 命名空间中的名称。要使 `std` 命名空间中所有的名称在代码中都是可用的，需要使用如下的 `using` 指令：

```
using namespace std;
```

为什么一定需要这个 `using` 指令呢？让我们从相反的角度思考一下，如果程序不包含上述的 `using` 指令，那么程序中 `cin` 和 `cout` 就可以被定义成不同于库中标准的内容（我们可能想重新定义自己的功能不同于标准版本的 `cin` 和 `cout`）。它们的标准定义存在于 `std` 命名空间中，但如果不使用 `using` 指令指明，程序不会知道任何 `std` 命名空间中的名称定义，因此我们自己定义的 `cin` 和 `cout` 成了唯一的定义。

全局命名空间

我们编写的任何代码都将属于某个命名空间。如果我们没有显式地将代码放在任何指定的命名空间中，那么这些代码就处于全局命名空间中。截至目前，我们的代码还没有指明放在某个命名空间中，所以所有名称定义都存在于全局命名空间中。全局命名空间不需要使用 `using` 指令，因为它是一直被使用的。我们可以认为程序自动地包含了 `using` 指令来使用全局命名空间。

值得注意的是，一个程序中可以使用多个命名空间。比如，程序中我们一直在使用全局命名空间，同时也经常使用 `std` 命名空间。如果一个名称在两个命名空间中都有定义，而程序又同时使用了这两个命名空间，会发生什么呢？这样的程序将产生错误（可能是编译时的错误，也可能是运行时错误，这取决于具体情况）。我们可以在两个命名空间中使用相同的名字，但是在程序中一次只能使用一个命名空间中的名字。但这不是说一个程序中不能出现两个命名空间。可以在一个程序中的不同位置使用两个命名空间中的一个。

例如，假设有 `NS1` 和 `NS2` 两个命名空间，而函数 `myFunction` 是在两个命名空间中都定义为不带任何参数的 `void` 类型函数，下面的代码是合法的：

```
{
    using namespace NS1;
    myFunction();
}
{
    using namespace NS2;
    myFunction();
}
```

上面的代码中，`myFunction` 函数的第一次调用中使用命名空间 `NS1` 中的函数定义，而第二个对 `myFunction` 函数的调用将使用命名空间 `NS2` 中的函数定义。

作用域

回顾上面的语句块。语句块是包括在一对大括号“{}”中的一系列语句、声明和其他代码。在一个语句块开头部分的 `using` 指令的作用范围仅仅是这个语句块。因此，上面代码中的第一个 `using` 指令仅作用于第一个语句块，而第二个 `using` 指令则作用于第二个语句块。我们常常将第一个语句块称为使用 `NS1` 的 `using` 指令的作用域，而使用 `NS2` 的 `using` 指令的作用域是第二个语句块。由于存在作用域规则，因此我们可以在同一个程序（比如前面包含两个以上语句块的程序）中使用两个包含相同定义名称的命名空间。

一般来说，我们将 using 指令放在语句块的最起始位置，也可以将 using 指令放在语句块中间，不过我们必须真的清楚该指令的具体作用域。在这种情况下，using 指令的作用域是从使用 using 指令的地方开始直至它所在的语句块结束。我们还可以在多个语句块中使用 using 指令来指明同一个命名空间，这个命名空间的作用域就可能跨越多个不连续的语句块。如果一个语句块中使用了 using 指令，那么这个语句块很可能就是某个函数体定义。

如果将 using 指令放在一个文件的起始位置（之前我们就是这么做的），那么该 using 指令的作用域将是整个文件。虽然 using 指令通常应该放在一个文件的起始部分（或者一个语句块的起始部分），不过还有一条更为完整的作用域规则是：在所有语句块之外的 using 指令的作用域是从该指令出现的地方开始，直至整个文件的结束。

### using 指令的作用域规则

using 指令的作用域是该指令所在的语句块（更准确地说，是从 using 指令出现的位置到语句块结尾）。如果 using 指令出现在所有语句块之外，那么该指令将作用于其后的整个文件代码。

## 创建一个命名空间

### 命名空间组

要将一段代码加入某个命名空间中，只需将该代码段放在命名空间组中，就像下面这样：

```
namespace Name_Space_Name
{
    Some_Code .
}
```

如果程序中包含这种命名空间组，则称为将 Some\_Code 中定义的名称放在命名空间 Name\_Space\_Name。这些名字（即命名空间中的名字定义）可以通过使用 using 指令来使用：

```
using namespace Name_Space_Name;
```

例如，下面从示例 11.5 中摘录的代码将函数 greeting 的声明放进了命名空间 Space1 中。

```
namespace Space1
{
    void greeting();
}
```

如果仔细看看示例 11.5 中的代码，就会发现函数 greeting 的定义也放在了命名空间 Space1 中。具体方法如下所示：

```
namespace Space1
{
    void greeting( )
    {
```

```

        cout << "Hello from namespace Space1.\n";
    }
}

```

值得注意的是，一个命名空间可以拥有多个命名空间组。在示例 11.5 中，命名空间 Space1 有两个命名空间组，而命名空间 Space2 同样也包含另外两个命名空间组。

一个命名空间中的所有名称在该命名空间组中都是可用的，同时这些名字也可以通过 using 指令在命名空间组以外被使用。例如，命名空间 Space1 中的函数声明和函数定义可以使用下面的语句使之在其他程序段中是可用的。

```
using namespace Space1
```

正如示例 11.5 所示。

### 示例 11.5 命名空间使用举例

```

1
2 #include <iostream>
3 using namespace std;

4 namespace Space1
5 {
6     void greeting( );
7 }

8 namespace Space2
9 {
10     void greeting( );
11 }

12 void bigGreeting( );
13 int main( )
14 {
15     {
16         using namespace Space2;
17         greeting( );
18     }

19     {
20         using namespace Space1;
21         greeting( );
22     }

23     bigGreeting( );

24     return 0;
25 }

26
27 namespace Space1
28 {
29     void greeting( )
30     {
31         cout << "Hello from namespace Space1.\n";

```

这段代码中的名称使用命名空间 Space2、std，以及全局命名空间中的定义。

这段代码中的名称使用命名空间 Space1、std，以及全局命名空间中的定义。

这里的名称只使用 std 和全局命名空间中的定义。

```

32     }
33 }

34 namespace Space2
35 {
36     void greeting( )
37     {
38         cout << "Greetings from namespace Space2.\n";
39     }
40 }
41 void bigGreeting( )
42 {
43     cout << "A Big Global Hello!\n";
44 }

```

### 示例运行结果

```

Greetings from namespace Space2.
Hello from namespace Space1.
A Big Global Hello!

```

### 将一个定义放入命名空间中

将一个名称的定义放在命名空间组中，就可以将这个名称放在该命名空间中了，其语法如下：

```

namespace Namespace_Name
{
    Definition_1
    Definition_2
    .
    .
    .
    Definition_Last
}

```

一个命名空间可以包含多个命名空间组（这些组甚至可以存在于不同的文件中），在一个组里的所有定义都是属于同一个命名空间中的。

### 自测练习题

- 考虑示例 11.5 中的程序，我们能否在函数 bigGreeting 处使用名字 greeting？
- 在练习题 6 中，我们看到不能在全局命名空间中出现如下的函数定义：

```
void greeting();
```

那么能否在全局命名空间中定义这个函数呢？

```
void greeting(int howMany );
```

## using声明

本章将介绍 using 指令的另外一种用法，可以让某个命名空间中特定的某个名字在程序中变得可用，而不是将所有命名空间中的名称引入。在第 1 章中已经出现过这种使用方法，这里可以看成是对前面知识的复习和强调。

假设我们现在面临这样的情形。现在有两个命名空间：NS1 和 NS2，我们想要使用 NS1 中定义的函数 fun1 和 NS2 中定义的函数 fun2，但在 NS1 和 NS2 中都定义了一个名为 myFunction 的函数（假设这里出现的所有函数都不带任何参数，因此也就不会有函数重载的情况发生），那么我们将不能使用下面的 using 语句：

```
using namespace NS1;
using namespace NS2;
```

因为这样做可能会导致函数 myFunction 定义的类型（如果函数名 myFunction 在程序中一直没有出现过，那么大多数编译器都不会检测到可能发生的错误，程序将被正常编译和运行程序）。

因此，我们需要告诉编译器只使用命名空间 NS1 中的函数 fun1 和命名空间 NS2 中的函数 fun2，而无须引入 NS1 和 NS2 中的其他函数名称。我们在这之前已经使用过这种方法了。下面就是我们需要的答案。

```
using NS1::
    fun1;
using NS1::fun1;
using NS2::fun2;
```

这种用法可以归纳如下：

```
using Name_Space::One_Name;
```

使用这种声明方式可以使某个空间 Name\_Space 中的一个名字 One\_Name 变得可用，但该命名空间中的其他名字仍然不可用。这种方式被称为 **using 声明**。

注意，在上面 using 声明中所使用的使用域说明符“::”和我们在定义类的成员函数时所使用的作用域说明符一样。这两种作用域说明符的用法非常相似。例如，示例 11.2 中有下面的函数定义：

```
void DigitalTime::advance(int hoursAdded, int minutesAdded)
{
    hour = (hour + hoursAdded)%24;
    advance(minutesAdded);
}
```

在上面的定义中，作用域说明符指明函数 advance 是属于类 DigitalTime 的，而不是其他类。类似地，下面的语句：

```
using NS1::fun1;
```

表明我们要使用的 fun1 函数是定义在命名空间 NS1 中的，而不是定义在其他命名空间中的。

using 声明和 using 指令有两点不同，如下面的 using 声明：

```
using std::cout;
```

以及 using 指令的用法：

```
using namespace std;
```

这两种用法的区别是：

1. 第一种 using 声明仅仅使命名空间中的一个名字对于当前代码可用，而第二种用法中的 using 指令则使命名空间中的所有名字都可用。
2. 使用 using 声明来使某个名字可用之后，该名字在代码中就不能被再次定义。然而，第二种使用 using 指令的方式则是潜在地占用了 std 命名空间中所有名称的定义。

上述第一点区别是显而易见的，但第二点区别却有些微妙。例如，假设命名空间 NS1 和 NS2 中都包含函数 myFunction 的定义，但除此之外没有其他名字冲突，那么下面的语句不会引起任何错误：

```
using namespace NS1;
using namespace NS2;
```

但注意前提是引起冲突的名字 myFunction 在程序代码（上面 using 指令的作用域之内）中一直不会出现。

而另外一方面，即使是 myFunction 函数在程序中一直没有使用，下面的语句仍然是非法的：

```
using NS1::myFunction;
using NS2::myFunction;
```

有时这种微妙的区别是非常重要的，但一般在常见的代码中我们很少碰到。所以，我们经常所说的 using 指令既代表上面第二种使用 **using 指令** 的方式来指明一个命名空间，也代表第一种 using 声明的意思。

## 限定名称

本节我们将要介绍一种前面未曾提到过的限定名称的方法。假设我们需要使用定义在命名空间 NS1 中的名字 fun1，而且只需使用该名字一次（或者很少的几次），那么我们就可以使用命名空间的作用域说明符来修饰该函数名（或者其他类型的名字），示例如下：

```
NS1::fun1();
```

这种方式常常用于指明函数的参数类型，例如：

```
int getInput(std::istream inputStream)
...
```

在上面的 getInput 函数中，参数 inputStream 是 istream 类型，而 istream 则定义在 std 命名空间中。如果这里所使用的类型名字 istream 是程序中唯一用到的 std 命名空间中定义的名字（或者其他用到的名字都使用了类似的限定 std::），那么，在程序中就不需要使用下面的 using 指令：

```
using namespace std;
```

或者

```
using std::istream;
```

需要注意的是，可以在另外一个命名空间的作用域范围内使用 `std::stream`，即使在这个命名空间中也定义了相同的 `istream` 名字。在这种情况下，`std::istream` 和 `istream` 将拥有不同的含义。例如：

```
using namespace MySpace;
void someFunction(istream p1, std::istream p2);
```

假定 `istream` 在命名空间 `MySpace` 中也存在定义，那么 `p1` 将是命名空间 `MySpace` 中定义的 `istream` 类型，而 `p2` 则是命名空间 `std` 中定义的 `istream` 类型。

## 自测练习

### 8. 下面程序的输出是什么？

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

namespace Hello
{
    void message( );
}

namespace GoodBye
{
    void message( );
}

void message( );

int main( )
{
    using GoodBye::message;

    {
        using Hello::message;
        message( );
        GoodBye::message( );
    }

    message( );

    return 0;
}

void message( )
{
    cout << "Global message.\n";
}

namespace Hello
{
    void message( )
    {
        cout << "Hello.\n";
    }
}
```



```

    }
}

namespace GoodBye
{
    void message( )
    {
        cout << "Good-Bye.\n";
    }
}

```

9. 声明一个名为 `wow` 的 `void` 型函数。该函数有两个参数：第一个参数的类型为命名空间 `speedway` 中定义的 `speed` 类型，第二个参数为命名空间 `indy500` 中定义的 `speed` 类型。

### 提示：为命名空间取名

一个好的办法是将姓或者某个唯一的字符串放在我们所定义的命名空间中，这样可以减少和其他命名空间重名的机会。当多个程序员同时在一个工程中工作时，不同的命名空间应该有不同的名字，这一点非常重要。否则在同一个作用域范围内，很容易出现相同名字的重复定义。这也就是在示例 11.9 中的命名空间 `DTimeSavitch` 包含作者名字 `Savitch` 的原因。■

### 示例：一个定义在命名空间中的类

示例 11.6 和示例 11.7 重写了类 `DigitalTime` 的头文件 `dtime.h` 和实现文件。`DigitalTime` 类的定义被放在了一个名为 `DTimeSavitch` 的命名空间中，注意命名空间 `DTimeSavitch` 跨越了两个文件 `dtime.h` 和 `dtime.cpp`。一个命名空间可以跨越多个文件。

如果按照示例 11.6 和示例 11.7 重写了类 `DigitalTime`，那么示例 11.3 中使用类 `DigitalTime` 的应用程序也必须以某种方式指明命名空间 `DTimeSavitch`。例如：

```
using namespace DTimeSavitch;
```

或者

```
using DTimeSavitch::DigitalTime;
```

### 示例 11.6 将类定义放在命名空间中（头文件）

```

1 // 这是头文件 dtime.h。      这个类还有一个更完善的版本的定义，将在示例 11.8 和示例 11.9 中出现。
2 #ifndef DTIME_H
3 #define DTIME_H

4 #include <iostream>
5 using std::istream;
6 using std::ostream;

```

```

7 namespace DTimeSavitch
8 {
9
10     class DigitalTime
11     {
12
13         < DigitalTime 类的定义在示例 11.1 中 >。
14     };
15
16 } // DTimeSavitch
17 #endif //DTIME_H

```

注意命名空间 DTimeSavitch 跨越了两个文件，另一处将会在示例 11.7 中出现。

## 无名称的命名空间

### 编码单元

编码单元可以是文件，比如类的实现文件。还有那些通过 include 指令被包含的文件，如类的接口文件，都可以看做编码单元。每个编码单元中都存在一个无名称的命名空间。无名称的命名空间组的书写方式和其他命名空间组的书写方式相同，但是不会给出任何名称。例如：

```

namespace
{
    void sampleFunction()
    {
        .
        .
        .
    }
} // 无名称的命名空间

```

### 示例 11.7 将一个类放入无名称的命名空间（实现文件）

```

1 // 这是实现文件 dtime.cpp。
2 #include <iostream>
3 #include <cctype>
4 #include <cstdlib>
5 using std::istream;
6 using std::ostream;
7 using std::cout;
8 using std::cin;
9 #include "dtime.h"

```

可以单独使用 using 指令：  
using namespace std;  
来替换这里的四个 using 指令。  
但是四个 using 的用法是更合适的。

```

10 namespace DTimeSavitch
11 {
12
13     < 假设示例 11.2 中的所有定义都放在这里。 >
14
15 } // DTimeSavitch

```

所有在无名称的命名空间中定义的名字都只在这个命名空间所在的编码单元中有效，所以这些名字在该编码单元之外还可以被再次使用。例如示例 11.8 和示例 11.9

给出了类 `DigitalTime` 的接口文件和实现文件的重写版本。帮助函数 `readHour`、`readMinute` 及 `digitToInt` 都定义在无名称命名空间，所以它们都只在该编译单元中有效。示例 11.10 演示了无名称命名空间中的名字如何在编译单元之外的地方再次使用。在示例 11.10 中，名为 `readHour` 的函数在应用程序中被重新定义，作为一个新的函数被使用。

如果我们观察示例 11.9 中类的实现文件，就会发现三个帮助函数（`digitToInt`、`readHour` 及 `readMinute`）在无名称命名空间之外被使用时，并没有附加命名空间限制，这是因为任何在无名称命名空间中定义的名称都可以在该编译单元中引用而无须附加命名空间限定词（实际上也只能这么做，因为无名称命名空间根本就没有名字可以用作限定）。

### 示例 11.8 将帮助函数隐藏在一个命名空间中（接口文件）

```

1 // 这是头文件 dtime.h，它是类 DigitalTime 的接口文件。
2 // 该类表示一个 24 小时制的时间。
3 // 比如 9:30 代表上午 9:30，而 14:45 代表下午 2:45。
4 #ifndef DTIME_H
5 #define DTIME_H
6 #include <iostream>
7 using std::istream;
8 using std::ostream;
9 namespace DTimeSavitch
10 {
11     class DigitalTime
12     {
13     public:
14         DigitalTime(int theHour, int theMinute);
15
16         DigitalTime( );
17         // 将时间初始化为 0:00，表示午夜。
18
19         getHour( ) const;
20         getMinute( ) const;
21
22         void advance(int minutesAdded);
23         // 将时间累加 minutesAdded 代表的分钟数。
24         void advance(int hoursAdded, int minutesAdded);
25         // 将时间累加 hoursAdded 代表的小时数和 minutesAdded 代表的分钟数。
26         friend bool operator ==(const DigitalTime& time1,
27                                const DigitalTime& time2);
28         friend istream& operator >>(istream& ins,
29                                     DigitalTime& theObject);
30         friend ostream& operator <<(ostream& outs,
31                                     const DigitalTime& theObject);
32     private:
33         int hour;
34         int minute;

```

这是我们 `DigitalTime` 类的最终版本，这是个足够好的版本，是你应该使用的。  
 示例 11.9 会给出如何实现这个接口。  
 注意在接口文件中并不需要提到帮助函数。

```

31     };
32 } //DTimeSavitch
33 #endif //DTIME_H

```

### 示例 11.9 将帮助函数隐藏在一个命名空间中 (实现文件)

```

1  // 这是类 DigitalTime 的实现文件 dttime.cpp。
2  // 类 DigitalTime 的接口定义在头文件 dttime.h 中。
3  #include <iostream>
4  #include <cctype>
5  #include <cstdlib>
6  using std::istream;
7  using std::ostream;
8  using std::cout;
9  using std::cin;
10 #include "dttime.h"

```

指定一个无名称的命名空间。

```

11 namespace ←
12 {
13     int digitToInt(char c)
14     {
15         return ( int(c) - int('0') );
16     }

```

无名称的命名空间中的名称定义仅限于本地编译单元，所以这些帮助函数对 dttime.cpp 文件来说是本地的。

```

17 // 使用 istream、cctype 和 cstdlib。
18 void readMinute(int& theMinute)
19 {
20     char c1, c2;
21     cin >> c1 >> c2;
22     if (!(isdigit(c1) && isdigit(c2)))
23     {
24         cout << "Error: illegal input to readMinute\n";
25         exit(1);
26     }
27
28     theMinute = digitToInt(c1)*10 + digitToInt(c2);
29
30     if (theMinute < 0 || theMinute > 59)
31     {
32         cout << "Error: illegal input to readMinute\n";
33         exit(1);
34     }
35
36 // 使用 istream、cctype 和 cstdlib。
37 void readHour(int& theHour)
38 {
39     char c1, c2;
40     cin >> c1 >> c2;
41     if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )

```

```

41         {
42             cout << "Error: illegal input to readHour\n";
43             exit(1);
44         }

45         if (isdigit(c1) && c2 == ':')
46         {
47             theHour = digitToInt(c1);
48         }
49         else //(isdigit(c1) && isdigit(c2))
50         {
51             theHour = digitToInt(c1)*10 + digitToInt(c2);
52             cin >> c2; // 忽略 ':'
53             if (c2 != ':')
54             {
55                 cout << "Error: illegal input to readHour\n";
56                 exit(1);
57             }
58         }

59         if (theHour == 24)
60             theHour = 0; // 将时钟重置为 0 点。

61         if (theHour < 0 || theHour > 23)
62         {
63             cout << "Error: illegal input to readHour\n";
64             exit(1);
65         }
66     }
67 } // 无名命名空间

68 namespace DTimeSavitch
69 {
70
71     // 使用库 istream.
72     istream& operator >>(istream& ins, DigitalTime& theObject)
73     {
74         readHour(theObject.hour);           在同一个编译单元中 (这里是文件 dtime.cpp),
75         readMinute(theObject.minute);       使用无名命名空间中的名称不需要再声明。
76         return ins;
77     }
78     ostream& operator <<(ostream& outs, const DigitalTime& theObject)
79     < 函数定义与示例 11.2 中完全相同。 >

80     bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
81     < 函数定义与示例 11.2 中完全相同。 >

82     DigitalTime::DigitalTime(int theHour, int theMinute)
83     < 函数定义与示例 11.2 中完全相同。 >

84     DigitalTime::DigitalTime( )
85     < 函数定义与示例 11.2 中完全相同。 >

86     int DigitalTime::getHour( ) const
87     < 函数定义与示例 11.2 中完全相同。 >

```

```

83     int DigitalTime::getMinute( ) const
        < 函数定义与示例 11.2 中完全相同。 >

84     void DigitalTime::advance(int minutesAdded)
        < 函数定义与示例 11.2 中完全相同。 >

85     void DigitalTime::advance(int hoursAdded, int minutesAdded)
        < 函数定义与示例 11.2 中完全相同。 >

86     } // DTimeSavitch

```

---

### 示例 11.10 将帮助函数隐藏在一个命名空间中 (应用程序文件)

```

1  // 这是一个应用程序文件 timedemo.cpp.
2  // 演示了如何将帮助函数隐藏在一个命名空间中。
3  #include <iostream>
4  #include "dttime.h"
5  void readHour(int& theHour);
6  int main( )
7  {
8      using std::cout;
9      using std::cin;
10     using std::endl;
11
12     using DTimeSavitch::DigitalTime;
13     int theHour;
14     readHour(theHour);
15     DigitalTime clock(theHour, 0), oldClock;
16
17     oldClock = clock;
18     clock.advance(15);
19     if (clock == oldClock)
20         cout << "Something is wrong.";
21     cout << "You entered " << oldClock << endl;
22     cout << "15 minutes later the time will be "
23         << clock << endl;
24
25     clock.advance(2, 15);
26     cout << "2 hours and 15 minutes after that\n"
27         << "the time will be "
28         << clock << endl;
29
30     return 0;
31 }
32
33 void readHour(int & theHour)
34 {
35     using std::cout;
36     using std::cin;

```

如果这里替换为一个 using 指令，也是可以的。不过很多专家建议要尽可能地减少每个 using 的作用范围，这里我们就举了一个这样的例子。

这里的 readHour 函数和示例 11.9 中定义的不是同一个函数。

我们之前都是在 main 函数中声明 using，所以它们的作用域只限于 main 函数。这里，我们需要再次声明 using，以便得我们可以在 readHour 函数中使用 cin 和 cout。

```

34     cout << "Let's play a time game.\n"
35         << "Let's pretend the hour has just changed.\n"
36         << "You may write midnight as either 0 or 24,\n"
37         << "but, I will always write it as 0.\n"
38         << "Enter the hour as a number (0 to 24): ";
39     cin >> theHour;
40 }

```

### 示例运行结果

```

Let's play a time game.
Let's pretend the hour has just changed.
You may write midnight as either 0 or 24,
but, I will always write it as 0.
Enter the hour as a number (0 to 24): 11
You entered 11:00
15 minutes later the time will be 11:15
2 hours and 15 minutes after that
the time will be 13:30

```

### 无名称命名空间

可以使用无名称命名空间将某个名称的定义局限在某个编码单元中。每个编码单元里都存在一个无名称命名空间。所有在这个无名称命名空间中定义的标识符名称都只在该编译单元中有效。将某个定义放在没有名字的命名空间组，就可以将该定义放在无名称命名空间中，语法示例如下：

```

namespace
{
    Definition1
    Definition2
    .
    .
    .
    Definition_Last
}

```

我们可以在该编码单元中使用无名称命名空间中的任何名字，而无须使用命名空间限制符。完整的例子见示例 11.8、示例 11.9 和示例 11.10。

### 陷阱：混淆全局命名空间和无名称命名空间

不要混淆全局命名空间和无名称命名空间。如果名字定义没有被放在任何命名空间组中，那么该名字定义属于全局命名空间。想让名字定义属于无名称命名空间，就必须将名字存放在以下面代码开头的命名空间组中，该命名空间组没有名字。

```

namespace
{

```

全局命名空间中的名字及无名称命名空间中的名字都不需要限定符就可以直接使用。但是，全局命名空间中的名字的作用域是整个代码的作用域（即所有的程序文件），而无名称命名空间中的名字的作用域仅仅是该编译单元。

在编写代码时，混淆全局命名空间和无名称命名空间的情况并不是很多，因为我们习惯于将全局命名空间中的名称看作不属于任何命名空间，尽管这在理论上是不正确的。但是在单纯讨论代码时，这种混淆则很容易发生。■

**提示：用无名称命名空间代替 static 修饰符**

在旧的 C++ 版本中，常常使用修饰符 static 来将一个名称的定义限制在单个文件之内。这种使用 static 的做法已经逐步被淘汰，我们应该转向使用无名称命名空间来将一个名称限制在某个编译单元之内。需要注意的是，这里 static 的用法与用来修饰类成员的 static（在第 7 章“静态成员”小节中介绍过）没有任何联系。既然 static 有如此多的用法，那么将 static 的一种用法禁用也未尝不是个好主意。■

 **提示：隐藏帮助函数**

这里有两种隐藏帮助函数的好方法：一种方法是帮助函数声明为类的私有成员函数，另一种方法就是将帮助函数放在类实现文件的无名称命名空间中。如果该帮助函数定义中包含对对象的操作，那么就应该将其作为私有成员函数。如果无须操作对象，那么，既可以将其作为类的静态成员函数（如示例 11.1 和示例 11.2 中的 `DigitalTime::readHour` 函数），也可以将其放在类的实现文件的无名称命名空间中（例如，示例 11.8 和示例 11.9 中的 `readHour` 函数）。

如果帮助函数不必操作对象，那么将其放在类的实现文件的无名称命名空间中可以使代码变得更加清晰，这样不但很好地将接口和实现隔离在不同的文件中，而且避免了函数名修饰符的过多使用。比如，在示例 11.9 中，我们可以直接使用函数名 `readHour` 而无须任何修饰，因为该函数是在无名称命名空间中的；然而在示例 11.2 中，却需要使用 `DigitalTime::readHour`。■

有意思的是，无名称命名空间与 C++ 不能在同一个命名空间中定义相同的名字这个规则看似矛盾，但其实每个编译单元都有自己的无名称命名空间，这让不同的编译单元可以方便地重叠。比如，类 a 的实现文件和一个使用类 a 的程序文件中都应该包含这个类的头文件（接口文件），头文件的内容将置于两个无名称命名空间中，只要各个命名空间自身都能理解这一点，就不会出现问题。比如，如果头文件的无名称命名空间中定义了一个名字，那么在实现文件和应用文件的无名称命名空间中都不能再定义这个名字，这样就不会出现重名的冲突了。



## 嵌套命名空间

命名空间的嵌套是合法的。当需要限定一个嵌套命名空间中的名字时，只需要限定该名字两次即可。例如，假设有下面的命名空间。

```
namespace S1
{
    namespace S2
    {
        void sample( )
        {
            .
            .
            .
        }
        .
        .
    } //S2
} //S1
```

**提示：**应该如何指明使用的命名空间

要标明代码中所使用的函数定义 (或者其他名称定义) 是在某个命名空间中定义的，有三种方法。举个例子，假设函数名为 `f`，命名空间为 `theSpace`，那么，第一种方法如下所示：

```
using namespace theSpace;
```

也可以使用下面的方法。

```
using theSpace::f;
```

最后一种方法可以不使用任何 `using` 指令，但必须标明对函数名的限定，即在所有使用 `f` 的地方都使用 `theSpace::f` 来替代。

那么，到底采用哪一种方式最合适呢？这三种方式都可以正常工作，而且权威人士们选择的方式也有很大的差别。然而，为了获取命名空间中的所有值，最好使用这种形式：

```
using namespace theSpace;
```

将这样的一条指令放在文件开头与将所有定义都放在全局命名空间中是不同的，后者是早期 C++ 版本的做法。所以这样做无法体现使用命名空间的优势（如果将上面的 `using` 指令放在一个语句块中，那么该指令只适用于这个语句块。这是一种十分合理的变量用法，许多权威人士都提倡这种方法）。

在大多数情况下，我们更倾向于第二种用法，即将类似下面的语句放到文件的开头部分：

```
using theSpace::f;
```

这种声明方式可以忽略在命名空间中不需要的名称。这样做也可以避免潜在的名字冲突。此外，这种方式不但很好地声明了将要使用的名称，而且也不像第三种方式 `theSpace::f` 那样总是需要使用命名空间名来标示所使用的名称。

如果在文件中不同的位置上要用到不同的命名空间，那么有时应该将 using 指令或者 using 声明放在某个语句块中，比如函数体的定义，而不是放在文件的起始部分。■

为了在命名空间 S1 之外调用 sample 函数，就必须使用下面这种方式。

```
S1::S2::sample();
```

而在命名空间 S1 中，S2 外调用 sample 函数的方式如下：

```
S2::sample();
```

另外，也可以使用一些 using 指令来完成相同的功能。

### 自测练习

10. 假如将示例 11.10 中的 using 声明，如下：

```
using std::cout;
using std::cin;
using std::endl;
using DTimeSavitch::DigitalTime;
```

替换为两个 using 指令：

```
using namespace std;
using namespace DTimeSavitch;
```

示例 11.10 的运行结果会有什么不同？

11. 下面程序的输出是什么？

```
#include <iostream>
using namespace std;
namespace Sally
{
    void message( );
}

namespace
{
    void message( );
}

int main( )
{
    {
        message( );
        using Sally::message;
        message( );
    }
    message( );

    return 0;
}

namespace Sally
```

```

{
    void message( )
    {
        cout << "Hello from Sally.\n";
    }
}

namespace
{
    void message( )
    {
        cout << "Hello from unnamed.\n";
    }
}

```

## 12. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;

namespace Outer
{
    void message( );
    namespace Inner
    {
        void message( );
    }
}

int main( )
{
    Outer::message( );
    Outer::Inner::message( );

    using namespace Outer;
    Inner::message( );
    return 0;
}

namespace Outer
{
    void message( )
    {
        cout << "Outer.\n";
    }
    namespace Inner
    {
        void message( )
        {
            cout << "Inner.\n";
        }
    }
}

```

## 本章小结

- 在定义一个类的时候，可以将类的定义和成员函数的实现放在不同的文件中。这样做可以将类的代码和使用该类的程序代码分开编译，并可以在多个程序中重复使用这个相同的类。
- 命名空间是一个名称定义的集合。这些名称定义包括类的定义及变量的声明等。
- 使用某个命名空间中的名称有三种方式：一是使用 using 指令将命名空间中的所有名称都变为可用；二是使用 using 声明将命名空间中的某一个名字变为可用；三是使用命名空间名和作用域说明符来修饰某个名字。
- 通过将名称定义放在某个命名空间组中的方式，就可以将该名字放入命名空间中。
- 无名称命名空间可以用来将一个名称定义的有效性限制在某个编码单元中。

## 自测练习题答案

1. a、b 和 c 部分应该放在类的接口文件中，d 到 h 部分应该放在实现文件中（所有相关的定义内容都应该放在实现文件中），i 部分（main 函数部分）应该放在应用程序文件中。
2. 接口文件的后缀应该是 .h。
3. 只有实现文件需要重新编译。接口文件不需要编译。
4. 只有实现文件需要重新编译。不过所有文件都需要重新连接。
5. 需要将示例 11.1 中的类接口文件中的两个私有成员变量 hour 和 minute 删除，添加一个 minutes 成员变量，接口文件不需要修改。对于实现文件，我们需要修改所有的构造函数和其他成员函数的定义，包括运算符的重载。（在这种情况下，并不需要修改帮助函数 readHour、readMinute 或 digitToInt，但对其他的类或者对这个类的重新实现可能就行不通了。）例如，重载的运算符 ">>" 的定义可以修改为下面这样：

```
istream& operator >>(istream& ins,
                    DigitalTime& theObject)
{
    int inputHour, inputMinute;
    DigitalTime::readHour(inputHour);
    DigitalTime::readMinute(inputMinute);
    theObject.minutes = inputMinute + 60*inputHour;
    return ins;
}
```

无须修改使用这个类的应用程序文件。不过由于接口文件和实现文件被修改了，所以我们需要将所有的文件重新编译。

6. 不能。如果使用 greeting 代替 bigGreeting，那么 greeting 将被定义在

全局命名空间中。命名空间 `Space1` 和全局命名空间中的名称都是同时有效的，那么函数声明 `void greeting()`；将会有两个不同的定义。

7. 可以。额外增加定义不会产生问题，因为函数是可以重载的。当命名空间 `Space1` 和全局命名空间都有效时，函数名 `greeting` 等于被重载。上个练习题有问题，是因为两个 `greeting` 函数定义具有相同的参数列表。

```
8. Hello
   Good-Bye
   Good-Bye
```

```
9. void wow(speedway::speed s1, indy500::speed s2);
```

10. 程序的运行结果不会发生变化。不过大多数人建议使用 `using` 声明，就像示例 11.10 中那样。不论使用哪种方式，都存在两个不同的名为 `readHour` 函数。在示例 11.10 中定义的 `readHour` 和示例 11.9 中定义在无名命名空间中的函数不同。

```
11. Hello from unnamed.
    Hello from Sally.
    Hello from unnamed.
```

```
12. Outer.
    Inner.
    Inner.
```

## 编程练习

1. 在一个你熟悉的开发环境中练习命名空间和分散编译。在文件 `f.h` 中声明函数 `void f()`，并将其放置在命名空间 `A` 中。在文件 `g.h` 中声明函数 `void g()` 并将其放置在命名空间 `A` 中，然后将这两个函数的定义放在相应的文件 `f.cpp` 和 `g.cpp` 中。这两个函数的功能可以任意编写，但一定要包含如下的语句：

```
cout << "Function_Name called" << endl;
```

上面的 `Function_Name` 应该是实际的函数名。在另外一个文件 `main.cpp` 中编写 `main` 函数，并使用 `#include` 包含访问命名空间 `A` 所需要的头文件。在 `main` 函数中调用 `f` 和 `g`，然后在开发环境中编译代码，连接并执行程序。访问命名空间中的名称时应该使用如下的局部 `using` 声明：

```
using std::cout;
```

也可以使用：

```
using namespace std;
```

将上面的语句放在某个程序块中或者使用命名空间名在局限定使用的名字，不要使用全局的命名空间指令，如下所示：

```
using namespace std;
```

当然除了上面提到的 `std` 命名空间和 `cout` 以外，正确使用命名空间 `A` 和函数 `f` 及 `g` 也是必需的。

完成这些后，整理一份关于在你的环境中如何创建并使用命名空间的总结。

2. 将类 `PFArrayD` 及示例 10.10、示例 10.11 和示例 10.12 中的演示程序修改为使用命名空间和分散编译的程序。将类的定义和其他函数的声明放在一起，类的实现单独放在一个文件中。命名空间的定义应该分布在这两个文件中，演示程序放在第三个文件中。要使用命名空间中的名字，应该使用如下局部的 `using` 声明：

```
using std::cout;
```

或者如下代码：

```
using namespace std;
```

将上面的语句放在某个程序块中或者使用命名空间名在局部限定使用的名字，不要使用全局的命名空间指令，如下所示：

```
using namespace std;
```

3. 扩展第 10 章中的编程练习 1，实现一个二维数组，将类的定义和实现放在一个命名空间中并提供对该命名空间的访问，然后测试你的代码。要使用命名空间中的名字，应该使用如下局部的 `using` 声明：

```
using std::cout;
```

或者如下代码：

```
using namespace std;
```

将上面的语句放在某个程序块中或者使用命名空间名在局部限定使用的名字，不要使用全局的命名空间指令，如下所示：

```
using namespace std;
```

4. 你可能遇到过需要验证你的系统用户的需求。下面的这个类 `Security` 可以用于认证用户和口令（注意这里的简单例子并不安全，实际应用中用户的口令应该进行加密并保存在数据库中）：

```
class Security
{
public:
    static int validate(string username, string password);
};
```

```
// 例子中硬编码了用户信息，这在实践中是不安全的。
// 如果证书无效则返回 0。
// 如果是合法用户则返回 1。
// 如果是合法管理员则返回 2。
```

```
int Security::validate(string username, string password)
{
    if ((username=="abbott") && (password=="monday")) return 1;
    if ((username=="costello") && (password=="tuesday")) return 2;
    return 0;
}
```

将这个类的内容分为两部分，一部分放在头文件 Security.h 中，一部分放在实现文件 Security.cpp 中。

然后再创建两个类 Administrator 和 User。Administrator 包含一个名为 Login 的函数，如果用户的口令合法并且具有管理员权限，此函数返回 true。User 类也包含一个 Login 函数，如果用户的口令合法并且具有普通用户或者管理员权限，则返回 true。

这两个类都应该有单独的头文件和实现文件。

最后编写一个 main 函数调用这两个类的 Login 函数，测试它们是否能正常工作。main 函数应该单独放在一个文件中，确保使用 #ifndef 来保证头文件只被包含过一次。

5. 这个联系中我们将探索一下无名称命名空间是如何工作的。下面是一些用来验证用户名和密码的代码片段。输入并验证用户名的代码和输入并验证密码的代码是分开放在不同的文件中的。

文件 user.cpp：

```
namespace Authenticate
{
    void inputUserName( )
    {
        do
        {
            cout << "Enter your username (8 letters only)"<< endl;
            cin >> username;
        } while (!isValid( ));
    }
    string getUsername( )
    {
        return username;
    }
}
```

在无名称命名空间中定义变量 username 和函数 isValid( ) 并编译。如果用户名恰好包含八个字符，那么函数 isValid( ) 返回 true。为这段代码生成一个合适的头文件。

在文件 password.cpp 中重复上述动作，将变量 password 和函数 isValid( ) 定义在无名称命名空间中，当输入的密码字符超过八个且至少包含一个非字母字符时函数返回 true。

文件 password.cpp :

```
namespace Authenticate
{
    void inputPassword( )
    {
        do
        {
            cout << "Enter your password (at least 8 characters " <<
                "and at least one non-letter)" << endl;
            cin >> password;
        }while (!isValid( ));
    }
    string getPassword( )
    {
        return password;
    }
}
```

现在你已经有了两个名叫 isValid( ) 的函数, 分别定义在不同的命名空间中。  
将如下的 main 函数放在合适的地方, 编译并运行程序。

```
int main( )
{
    inputUserName( );
    inputPassword( );
    cout << "Your username is " << getUsername( ) <<
        " and your password is: " <<
        getPassword( ) << endl;
    return 0;
}
```

用几组用户名和密码测试程序。







# 流和文件 I/O 操作

# 12



## 12.1 I/O流 430

文件I/O 431

陷阱：流变量的使用限制 434

向文件追加输出内容 435

提示：打开文件的另一种方法 436

提示：检查一个文件是否已被成功打开 439

字符I/O 440

文件结束检查 440

## 12.2 I/O流工具 444

使用文件名输入 444

使用流函数进行格式化输出 444

控制符 447

保存设置的标记 448

更多的输出流成员函数 449

示例：整理文件格式 450

示例：编辑文本文件 452

## 12.3 流的继承层次：继承概述 455

流之间的继承 455

示例：另一版本的newLine函数 457

使用类stringstream解析字符串 460

## 12.4 随机文件存取 462

# 第 12 章 流和文件 I/O 操作

鱼说，我们有我们的溪流和池塘。

但，这就是一切了吗？

鲁伯特·布鲁克，《天堂》（1913）

就像溪流中的叶子不知将被带入湖泊还是大海，程序的输出也不知会被流带到屏幕还是文件。

某计算机系卫生间墙上（1995）

## 概述

程序的输入和输出是由叫作流的特殊对象实现的。“流”的概念使得程序不必知道（至少不必清楚地知道）数据的输入源自何方，数据的输出将去向何处。这意味着文件的输入与我们日常使用的键盘输入本质上是相同的，而文件的输出和屏幕上的输出本质上是相同的。

文件的输入/输出将会用到一些会在第 14 章详细介绍的继承的概念。但是我们还是把这章的内容放到了讲述继承的概念之前，这是因为大部分程序员总是迫不及待地想尽早读写文件。所以本章介绍文件读写的过程中将会提到一些继承的概念。

在学习完第 1 ~ 4 章和第 6 ~ 9 章之后，就可以学习本章的内容了。也就是说本章的学习可以安排在第 5 章和第 10、11 章之前。即便没有学完上述所有章节，使用本章中的知识也足以写一些简单的文件读写程序了。在 12.1 节将介绍一些文件输入/输出的基本概念，在学习了第 1 ~ 4 章、第 6 章和第 9 章中 get 和 put 函数的相关知识后就可以学习 12.1 节的内容了。第 9 章中的 get 和 put 函数那部分内容相对独立，不了解第 9 章的其他部分内容并不会影响对 12.1 节的学习。在 12.2 节中除了“使用文件名输入”的部分，其他内容都可以在学习完第 1 ~ 4 章、第 6 章和第 9 章中的 get 和 put 函数后学习。

如果你还没有阅读过第 11 章中关于命名空间的内容，那你需要看一下第 1 章中关于命名空间的内容。

## 12.1 I/O 流

天啊！40 年来我天天都在朗诵散文，自己却没察觉到。

莫里哀，《贵人迷》

流  
输入流  
输出流

流就是字符或者其他格式的数据的流动。假如数据流动的方向是进入你的程序，那么这种流叫作**输入流**。假如数据流动的方向是由程序向外，那么这种流叫作**输出流**。假如输入流的源头是键盘，那么程序将从键盘接收数据；假如输入流的源头是文件，

那么程序将从指定的输入文件接收数据。同样，输出流的目的地可以是屏幕或者文件。

cin 和  
cout 是流

可能我们还没有注意到，其实我们在前面的程序中已经使用过流了。我们之前使用的 cin 就是将程序的输入接在了键盘上，cout 使程序的输出指向了屏幕。只要在程序中包含头文件 `<iostream>` 并且使用 using 指令引入 std 命名空间，就可以在程序中自由使用这两种流。我们还可以自己定义其他的流来从文件获得输入或者将输出写入到文件中。这些流一旦被定义好，就可以在程序中像 cin 和 cout 一样被使用。比如，假设我们在代码中定义了一个 inStream（稍后再介绍定义的具体内容）负责从文件中读取数据，那我们可以用下面这样的语句来为一个 int 型的 theNumber 变量赋值：

```
Int theNumber;
inStream >> theNumber;
```

同样，假如在程序中定义了一个 outStream 流来向另外一个文件写入数据，我们就可以利用它将 theNumber 的值写入到文件中。下面的代码会把字符串“theNumber is”及后面的 theNumber 变量的值都写入流 outStream 关联的文件中去。

```
outStream << "theNumber is " << theNumber << endl;
```

流和目标文件之间建立某种联系之后，程序就可以像实现键盘和屏幕的输入/输出一样，实现文件的输入和输出。

## 文件 I/O

本章中用于输入/输出的文件都是文本文件，它和包含 C++ 源程序的文件是同一种类型的文件。

读取  
写入

我们将程序从文件中获取输入叫作文件的读取；程序向一个文件中输出数据叫作文件的写入。从文件中读取数据有很多方式，但是本小节中讲到的读文件都是指从文件的开始读到文件的末尾（或者程序遇到读操作的终止位置）。使用这种方法的时候程序不允许同时对这个文件做备份或者其他操作。这与程序从键盘上获取输入没什么不同，所以这并没有什么奇怪的（之后我们将会看到程序可以从头重新读取文件，但是这是重新开始，并不是备份）。同样，使用这种方法程序也可以将输出数据顺序地写入到一个文件中，但是不允许备份或者改变已经写入到文件中的内容。这和程序向屏幕上输出数据是一样的：可以向屏幕输出很多内容，但是我们不能备份或者修改屏幕上的输出内容。程序中文件的输入/输出都是通过流来实现的。

要将输出数据写入一个文件，程序必须使用类 ofstream 的一个对象与该文件相关联。程序要从文件中读取数据，必须使用 ifstream 类的一个对象与该文件相关联。ifstream 类和 ofstream 类定义在 `<fstream>` 库中并且包含在 std 命名空间中。所以，假如程序需要使用文件的输入和输出功能，就必须包含下面的语句：

<fstream>

```
#include <fstream>
using namespace std;
```

或者

```
#include <fstream>
using std::ifstream;
```

```
using std::ofstream;
```

### 声明流

声明一个流必须像声明一个变量一样。我们用下面的代码声明一个文件的输入流 `inStream`，以及一个文件的输出流 `outStream`：

```
ifstream inStream;
ofstream outStream;
```

### 将流关联

#### 到文件

#### open

每一个流变量都关联着一个文件，我们刚刚声明的 `inStream` 和 `outStream` 也不例外。流变量的 `open` 成员函数可以对文件进行打开操作。比如，假设要将 `inStream` 和文件 `infile.txt` 相关联，那么在使用 `inStream` 读取文件内容之前代码必须包含下面的语句：

```
inStream.open("infile.txt");
```

#### 路径名

需要指定的文件名也可以是一个路径名（目录或者文件夹）。具体要给出的路径名在不同的操作系统之间会有细微的差别，可以在自己的操作系统上多尝试几次或者向有经验的高手请教。在本书的例子中，我们假设要操作的目标文件和程序文件在同一个目录下，所以我们可以使用简单的文件名。

只要程序中声明了输入流变量并使用 `open` 函数打开了文件，后面的代码中就可以使用提取运算符“>>”从文件的输入流变量中获取输入，就和使用 `cin` 的时候一样。以下代码将从 `inStream` 关联的文件中读取两个数值，并将它们保存在变量 `oneNumber` 和 `anotherNumber` 中。

```
int oneNumber, anotherNumber;
inStream >> oneNumber >> anotherNumber;
```

打开一个输出流（关联了一个文件）的方法和打开输入流的方法相同。比如，下面的代码声明了输出流 `outStream` 并将它与文件 `outfile.txt` 做了关联。

```
ofstream outStream;
outStream.open("outfile.txt");
```

当使用 `ofstream` 类型的流时，假如相关联的文件不存在，成员函数 `open` 将会创建一个新的输出文件。假如文件存在，`open` 函数被调用后，文件的内容将被清空（在后面的内容中我们还会讨论其他的文件打开方式）。

当 `outStream` 流对象通过调用 `open` 函数与一个文件相关联之后，程序通过插入运算符“<<”将数据写入文件。比如，下列代码会将两个字符串和变量 `oneNumber` 与 `anotherNumber` 的值写入流对象 `outStream` 关联的文件中（在当前这个例子中是一个名为 `outfile.txt` 的文件）。

```
outStream << "oneNumber = " << oneNumber
<< " anotherNumber = " << anotherNumber;
```

### 使用重载的“>>”和“<<”运算符

就像在第8章中说过的，假如重载了运算符“>>”和“<<”，那么这种重载应用于文件的输入/输出流和应用于 `cin`、`cout` 的情况是一样的（假如还没有看过第8章的介绍，请忽略本注解，这些内容将在第8章里再次出现）。

外部文件名

值得注意的是，当程序在操作一个文件的时候，文件看起来有了两个名字，一个是操作系统使用的该文件的真实文件名，我们称之为**外部文件名**。在本书的示例代码中，外部文件名分别是 `infile.txt` 和 `outfile.txt`。外部文件名在某种意义上来说是文件的“真实名称”。外部文件名的命名规则对不同操作系统是不同的。本书例子中使用的 `infile.txt` 和 `outfile.txt` 也许并不符合你的操作系统上的文件命名规则，请按照机器上真实安装的操作系统的命名规范来为文件进行命名。尽管外部文件名是文件的真实名称，但是在程序中也只有一次使用。外部文件名作为 `open` 函数的一个参数，当文件被打开后，流对象和与其关联的文件是一一对应的，所以在程序中，流对象的名称也可以看作是文件的第二个名称。

### 一个文件两个名字

程序中出现的**所有输入文件和输出文件都有两个名字**，外部文件名是文件的真实名称，仅仅在调用成员函数 `open` 的时候被使用，用来将文件同流对象相关联。程序在调用 `open` 函数后就可以使用流对象的名称作为文件名了。

示例 12.1 中程序从一个文件中读取了三个数值，然后将它们的和以及一些文本写到了另外一个文件中。

`close`

当程序的输入或者输出结束后，必须关闭打开过的文件。关闭文件将断开流对象与文件的关联。关闭文件通过调用成员函数 `close` 来实现。示例 12.1 中的以下代码描述了如何使用 `close` 函数。

```
inStream.close();
outStream.close();
```

需要注意的是函数 `close` 没有参数。假如程序在结束之前没有关闭打开的文件，那么操作系统将会为你自动关闭打开的文件。但是程序员应该养成在程序中关闭打开过的文件的习惯。这有两个原因：首先，操作系统只会在程序正常退出的情况下才会为你自动关闭文件，假如程序遇到错误异常退出，之前打开的文件不会被自动关闭，从而变成了一个不正常的状态。假如在文件操作完毕后马上就关闭打开的文件，文件被破坏的可能性就很小了。其次，你也许会将程序的输出写入到文件中，然后再将文件的内容读入程序。在这种情况下，程序需要在完成写入文件的操作后关闭文件，然后在需要的时候再使用输入流打开文件（对一个打开的文件既做输入又做输出是可能的，但实现的方法会略有不同）。

输出流有一个并不太常用的成员函数 `flush`，但偶尔它还是有用的。出于效率的考虑，文件的写入过程中都使用了缓存，也就是说在内容真的被写入到磁盘上的文件中之前，是暂时先被保存在某些地方。`flush` 成员函数的作用就是刷新输出流，使得相关的缓存区里的内容确实被写入到磁盘上的文件中。函数 `close` 的调用会自动触发 `flush` 函数的执行，这也是我们极少使用 `flush` 的原因。`flush` 函数的使用方法如下所示：

```
OutStream.flush();
```

## 示例 12.1 简单的文件输入/输出

```

1 // 从文件 infile.txt 中读入三个数字，并将这三个数字做加和，
2 // 将结果写入文件 outfile.txt 中。
3 #include <fstream>
4 using std::ifstream;
5 using std::ofstream;
6 using std::endl;

7 int main( )
8 {
9     ifstream inStream;
10    ofstream outStream;

11    inStream.open("infile.txt");
12    outStream.open("outfile.txt");

13    int first, second, third;
14    inStream >> first >> second >> third;
15    outStream << "The sum of the first 3\n"
16               << "numbers in infile.txt\n"
17               << "is " << (first + second + third)
18               << endl;

19    inStream.close( );
20    outStream.close( );

21    return 0;
22 }

```

示例 12.3 将给出一个此类的升级版。

## 示例运行结果

没有来自键盘的输入或者打印到屏幕上的输出。

**infile.txt**  
(没有被程序改变)

```

1
2
3
4

```

**outfile.txt**  
(代码执行后)

```

The sum of the first 3
numbers in infile.txt
is 6

```



## 陷阱：流变量的使用限制

我们可以使用和一般变量相同的方法声明一个流变量 (ifstream 或者 ofstream)，但是在某些场景中这些变量的使用并不像其他变量一样自由。比如我们对一个流变量赋值的时候不能使用赋值表达式。而且流类型 (ifstream、ofstream 或者其他流类型) 的对象做参数的时候，该参数必须是一个引用参数，而不能是一个传值参数。■

## 向文件追加输出内容

要向文件输出内容，必须先使用成员函数 `open` 打开文件并将它同 `ofstream` 类的一个对象相关联。之前我们使用这种方法（`open` 函数只传入了一个参数）总是产生一个空文件。假如指定文件名的文件已经存在，那么这个文件中的内容会全部消失。其实存在着另外一种打开文件的方式，将使得输出到文件的内容被添加到文件已有内容的末尾。

想向文件名为 `important.txt` 的文件中追加输出内容，可以使用带两个参数的 `open` 函数，代码如下所示：

```
Ofstream outStream;
outStream.open("important.txt", ios::app);
```

假如文件 `important.txt` 不存在，上面的代码将创建一个新的空文件；假如该文件已经存在，那么程序中所有输出到该文件的内容都会追加到文件的末尾，文件中原来的数据并不会丢失，详细内容可参见示例 12.2。

**ios::app**

第二个参数 `ios::app` 是 `ios` 类中定义的常量。`ios` 类定义在库 `<iostream>` 中（某些其他流库中也有）。`ios` 类的定义被放置在 `std` 命名空间中，所以下面的语句可以使 `ios` 类中的内容（比如 `ios::app`）在你的程序中可用：

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

或者

```
#include <iostream>
using std::ios;
```

### 示例 12.2 向文件中追加内容

---

```
1 // 向文件 alldata.txt 的末尾追加内容。
2 #include <fstream>
3 #include <iostream>
4 using std::ofstream;
5 using std::cout;
6 using std::ios;

7 int main( )
8 {
9     cout << "Opening data.txt for appending.\n";
10    ofstream fout;
11    fout.open("data.txt", ios::app);

12    fout << "5 6 pick up sticks.\n"
13         << "7 8 ain't C++ great!\n";

14    fout.close( );
15    cout << "End of appending to file.\n";

16    return 0;
17 }
```



## 示例运行结果

**data.txt**  
(代码执行前)

```
1 2 buckle my shoe.
3 4 shut the door.
```

**data.txt**  
(代码执行后)

```
1 2 buckle my shoe.
3 4 shut the door.
5 6 pick up sticks.
7 8 ain't C++ great!
```

## 屏幕输出

```
Opening data.txt for appending.
End of appending to file
```

## 向文件追加输出

假如想将内容添加到文件已有内容的后面, 可以使用下面的方式打开文件。

## 语法

```
Output_Stream.open(File_Name, ios::app);
```

## 示例

```
ofstream outStream;
outStream.open("important.txt", ios::app);
```

## 提示：打开文件的另一种方法

在类 `ifstream` 和 `ofstream` 中都有带参数的构造函数, 参数是要打开的文件的文件名和打开文件相关的一些其他参数。下面的例子将会帮助大家理解这些构造函数的具体使用方法。

下面的两条语句

```
ifstream inStream;
inStream.open("infile.txt");
```

可以使用如下的单条语句替代：

```
ifstream inStream("infile.txt");
```

这两条语句

```
ofstream outStream;
outStream.open("outfile.txt");
```

可以被下面这一行语句替代：

```
ofstream outStream("outfile.txt");
```

最后一个例子，下面这两行

```
ofstream outStream;
outStream.open( "important.txt" , ios::app);
```

可由如下一行代替：

```
ofstream outStream( "important.txt" , ios::app); ■
```

### 示例 12.3 带打开检查的文件输入/输出

```
1 // 从文件 infile.txt 中读取三个数值并将其加和输出到文件 outfile.txt 中。
2 #include <fstream>
3 #include <iostream>
4 #include <cstdlib> // 为了使用 exit 函数
5 using std::ifstream;
6 using std::ofstream;
7 using std::cout;
8 using std::endl;

9 int main( )
10 {
11     ifstream inStream;
12     ofstream outStream;

13     inStream.open("infile.txt");
14     if (inStream.fail( ))
15     {
16         cout << "Input file opening failed.\n";
17         exit(1);
18     }

19     outStream.open("outfile.txt");
20     if (outStream.fail( ))
21     {
22         cout << "Output file opening failed.\n";
23         exit(1);
24     }

25     int first, second, third;
26     inStream >> first >> second >> third;
27     outStream << "The sum of the first 3\n"
28               << "numbers in infile.txt\n"
29               << "is " << (first + second + third) << endl;

30     inStream.close( );
31     outStream.close( );
32     return 0;
33 }
```

示例运行结果（假如文件infile.txt不存在）

Input file opening failed.

## 文件输入/输出语句总结

假如想将内容添加到文件已有内容的后面, 可以使用下面的方式打开文件。

- 程序包含如下的文件:

```
#include <fstream>  ← 为了文件的输入/输出
#include <iostream> ← 为了使用 cout
#include <cstdlib> ← 为了使用 exit
```

添加下面的语句 (或者具有相同作用的其他语句):

```
using std::ifstream;
using std::ofstream;
using std::cout;
using std::endl; // 如果 endl 被使用
```

- 为输入流起一个变量名并将其声明为 ifstream 类型的变量; 为输出文件选择一个输出流变量名并将其声明为 ofstream 类型的变量,

比如这样:

```
ifstream inStream;
ofstream outStream;
```

- 将外部文件名作为 open 函数的参数, 调用 open 函数使得流文件关联。记得使用成员函数 fail 来检查 open 函数是否执行成功:

```
inStream.open("infile.txt");
if (inStream.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);
}

outStream.open("outfile.txt");
if (outStream.fail( ))
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

- 使用流 inStream 读取输入文件 infile.txt 的内容, 就像使用 cin 从键盘读取输入一样。比如:

```
inStream >> someVariable >> someOtherVariable;
```

- 使用流 outStream 向文件 outfile.txt 写入内容, 就像使用 cout 向屏幕输出内容一样, 比如:

```
outStream << "someVariable = "
<< someVariable << endl;
```

- 使用 close 函数关闭流:

```
inStream.close( );
outStream.close( );
```

### 提示：检查一个文件是否已被成功打开

有时会有一些情况导致 `open` 函数调用失败。比如想打开一个在指定位置并不存在的文件时，`open` 函数就会调用失败。另一个案例是，假如一个文件没有写的权限，以输出流的方式打开该文件也会引起错误。这些情况出现时我们也许看不到错误信息，但代码将不会按照我们希望的那样正常运行。所以 `open` 函数调用后应该检验一下文件的打开是否成功，假如不成功，应该及时终止程序（或者执行某些特定的操作）。

#### fail 成员函数

可以使用 `fail` 成员函数来检查流操作的失败。在类 `ifstream` 和类 `ofstream` 都有 `fail` 成员函数。这个 `fail` 函数不带参数，返回一个布尔类型的值。

应该在每次调用 `open` 函数后及时地调用 `fail` 函数，假如 `open` 函数执行失败，`fail` 函数将会返回 `true`。比如下面的代码中，假如 `open` 函数执行失败了，程序将会输出一个错误信息并及时终止；假如 `open` 函数执行成功了，`fail` 函数返回 `false`，程序会继续执行。

```
inStream.open("stuff.txt");
if (inStream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

示例 12.3 重写了示例 12.1，其中增加了测试文件是否被成功打开的语句。其余的处理文件的过程与示例 12.1 是相同的。假设 `infile.txt` 文件存在并且文件内容与示例 12.1 中的文件内容相同，那么执行示例 12.3 的代码后，将会创建和示例 12.1 结果一样的文件 `outfile.txt`。但是假如程序运行过程中发生了错误，导致其中的一个 `open` 函数没有成功执行，那么示例 12.3 的程序将会被终止并且向屏幕上输出一个错误信息。比如，假如 `infile.txt` 文件并不存在，`inStream.open` 的执行将会失败，程序被终止，屏幕上输出错误信息。使用 `cout` 输出错误信息，是为了将错误信息输出到屏幕上而非文件中。由于程序要使用 `cout` 来向屏幕输出消息（和文件的输出是相同的），所以代码其实部分添加了包含头文件的语句 `#include<iostream>`（实际上代码中包含了 `#include<fstream>` 之后并不需要再使用 `#include<iostream>` 语句，但是额外地添加这个头文件也不会带来任何错误，而且该包含语句的存在将一直提醒我们，程序除了使用文件 I/O 之外，还使用了屏幕输出）。■

### 自测练习題

1. 写一个程序将名为 `fin` 的流与输入文件相关联，用一个名为 `fout` 的流同输出文件相关联。在程序中应该如何声明 `fin` 和 `fout`？假如需要的话，应该在程序文件中包含哪些头文件？
2. 继续上一个练习，假设程序要从文件 `stuff1.txt` 中读取数据并将文件内容输出写入到文件 `stuff2.txt` 中，在程序中应如何将 `fin` 流与文件 `stuff1.txt` 相关联？如何将 `fout` 流与文件 `stuff2.txt` 相关联？记住不要忘记检查 `open` 函数是否执行成功。

3. 继续上两个练习，假设已经不需要再从文件 `stuff1.txt` 中获得输入，也不再向文件 `stuff2.txt` 中写输出，应该怎样关闭这些文件？
4. 修改示例 12.1 的程序，使其输出不是写到文件，而是输出到屏幕（输入仍然来自文件 `infile.txt`）。
5. 程序已经读取了一个文件中一行的一半内容，采用什么方法才能使程序重新读取文件的第一行？

## 字符 I/O

在第 9 章中已经介绍过字符的输入和输出，使用 `cin` 可以从键盘获得输入，使用 `cout` 可以向屏幕输出字符。这些方法也都可以使用在文件的输入和输出中，只需使用与文件相关联的输入流来代替 `cin`，或者使用与文件相关联的输出流来代替 `cout`。当然，`get`、`getline`、`putback`、`peek` 和 `ignore` 这些在键盘输入时可用的函数同样也适用于文件的操作<sup>1</sup>；`put` 函数同样可以将文件输出到屏幕上。

## 文件结束检查

在对一个文件做读操作时，最常用的方法是在一个循环中处理数据，直到遇到文件的结尾。有两种方法可以检测一个文件是否已经到了结尾，最直接的方法是使用 `eof` 成员函数。

**eof 成员函数** 每个输入流都有一个成员函数 `eof` 用来检测文件是否已经到达了结尾。`eof` 函数没有参数，假如输入流定义为 `fin`，那么调用它的 `eof` 函数的语句如下所示：

```
fin.eof();
```

**用 eof 函数结束一个输入循环** 这是一个布尔表达式，可以作为 `while` 循环、`do-while` 循环或者 `if-else` 语句的判断条件。假如文件已经被读到了结尾，那么表达式返回 `true`，否则返回 `false`。

我们经常需要测试读取是否已经到了文件的结尾，在成员函数 `eof` 前面加一个“非”就可以完成这项工作。所以，在调用成员函数 `eof` 时常常在它前面加一个“非”。C++ 中使用符号“!”表示“非”。比如，下面的 `while` 循环可以实现将输入流 `inStream` 相关联的文件的内容输出到屏幕上：

```
inStream.get(next);
while (! inStream.eof())
{
    cout << next;    ←——— 也可以使用 cout.put(next) 来完成相同的功能。
    inStream.get(next);
}
```

上面的 `while` 循环使用 `get` 函数从输入的文件中读取每个字符到 `char` 类型的变量 `next` 中，然后将变量输出到屏幕上。当文件读取遇到文件的末尾时，`inStream.eof()` 的值从 `false` 变成了 `true`。所以，

<sup>1</sup> 如果不了解 `getline`、`putback`、`peek` 和 `ignore` 这些函数，没关系，本章中除了在结尾处简单引用了 `ignore` 函数外，并没有在其他地方使用这些函数。

```
(! inStream.eof() )
```

表达式的值从 true 变成了 false, 循环结束。

表达式 `inStream.eof()` 在即将读取文件末尾的下一个字符时才会返回 true。比如, 假设文件的内容如下 (在字符 c 后面没有“换行符 `<\n>`”):

```
ab
c
```

该文件的实际内容应该包含了下面四个字符:

```
ab<换行符'\n'>c
```

上面这个循环先读取字符 a 并将其输出到屏幕上, 然后读取字符 b 并输出到屏幕上, 再读取换行符 `\n` 并输出到屏幕上, 最终读取到字符 c 并输出到屏幕上。这时程序已经读取完了所有字符, 但是此时 `inStream.eof()` 的值仍然是 false, 这个值要到程序再次尝试从文件中读取一个字符的时候变成 true。while 循环需要多读一个字符来终结自己, 所以上面的程序最后以 `inStream.get(next)` 语句结束。

文件的结尾处都有一个特殊的标记, eof 函数正是读到了这个特殊标记才由 false 变成了 true。这就是上面的 while 循环在已经读取完了所有字符后仍然需要再读一个字符才能结束的原因。但是文件结尾的标识符不是一个普通的字符, 不能像普通的字符那样被处理。我们可以读取到这个字符, 但是我们无法显示它。假如将这个特殊符号输出, 结果是无法预期的。这个结束标识符是操作系统自动添加在文件末尾的。

实际应用中还有一点需要注意, 有些编译器中 eof 返回 true 的时候并不会将文件末尾标示符读取出来。所以, 我们需要阅读自己编译器的相关文档, 或者自己编写一个小程序测试一下。

一种健壮的  
测试文件结  
尾的方法

检查文件是否已经到了结尾还有另外一种方法——使用提取运算符读文件时返回的布尔值。假如读取数据成功, 表达式

```
(inStream >> next)
```

将返回 true; 假如文件已经到达了结尾, 则返回 false。比如, 下面这段代码将会从与输入流 inStream 相关联的文件中读取出所有双精度浮点数并计算这些浮点数的和:

```
double next, sum = 0;
while (inStream >> next )
    sum = sum + next;
cout << "the sum is " << sum << endl;
```

上述循环看起来有些奇怪, `inStream >> next` 会从流 inStream 中读取一个数字并返回一个布尔型的值。一个包含提取运算符“`>>`”的表达式既可以实现一次提取操作, 又可以返回一个布尔值。<sup>2</sup>假如该文件还有其他数字可以读取, 那么这个数字将会被读取并且返回 true, 使得循环继续执行。假如文件中已经没有数字可以读取,

<sup>2</sup> 从技术角度来说, 布尔条件表达式是这样工作的: 如同第8章中解释的, 运算符“`>>`”的返回值是一个输入流的引用, 这个流引用会自动转换成一个bool值。如果流可以提取出数据, 则返回true, 否则返回false。

那么提取运算符不会提取任何东西并且会返回一个 false, 使得循环终止。在这个例子中输入变量 next 的类型是 double, 但实际中这种检测文件末尾的方法对于 int 和 char 类型的数据同样有效。

很多 C++ 程序员更倾向于后面这种方法, 因为这在 C 语言编程中同样经常使用。而且, 由于运行的细节不同, 后面这种方法的执行效率会更高。不论在检测文件结尾时是否采用这种方法, 我们都应该知道这种方法的存在, 以便能够读懂其他程序员写的代码。示例 12.4 给出了 eof 成员函数的使用方法。

### 示例 12.4 文件结尾检测

```

1 // 拷贝 story.txt 到 numstory.txt,
2 // 但是在每一行的开头都添加新的内容。
3 // 假设 story.txt 中没有内容。
4 #include <fstream>
5 #include <iostream>
6 #include <cstdlib>

7 using std::ifstream;
8 using std::ofstream;
9 using std::cout;

10 int main()
11 {
12     ifstream fin;
13     ofstream fout;

14     fin.open("story.txt");
15     if (fin.fail())
16     {
17         cout << "Input file opening failed.\n";
18         exit(1);
19     }

20     fout.open("numstory.txt");
21     if (fout.fail())
22     {
23         cout << "Output file opening failed.\n";
24         exit(1);
25     }

26     char next;
27     int n = 1;
28     fin.get(next);
29     fout << n << " ";
30     while (!fin.eof())
31     {
32         fout << next;
33         if (next == '\n')
34         {
35             n++;
36             fout << n << " ";
37         }
38     }

```

注意循环以 fin.get(next) 结束。成员函数 fin.eof() 会在程序读取到文件中最后一个字符之后才返回 true。

```

39         fin.get(next);
40     }

41     fin.close();
42     fout.close();

43     return 0;
44 }

```

### 示例运行结果

没有输出到屏幕的内容，也没有键盘录入的内容。

**story.txt**

(代码执行前)

```

The little green men had
pointed heads and orange
toes with one long curly
hair on each toe.

```

**numstory.txt**

(代码执行后)

```

1 The little green men had
2 pointed heads and orange
3 toes with one long curly
4 hair on each toe.

```

6. 下面的代码执行后将输出什么？假定文件 list.txt 包含下列数据（假设这些代码被放在一个完整的可运行的程序中，并且包含了所有需要的头文件及声明）：

```

ifstream ins;
ins.open("list.txt");
int count = 0, next;
while (ins >> next)
{
    count++;
    cout << next << endl;
}
ins.close();
cout << count;

```

文件 list.txt 只包含下面的三个数字（没有其他内容）。

```

1 2
3

```

7. 写一个返回值为 void 的函数 toScreen。函数 toScreen 包含一个类型为 ifstream 的 fileStream 的形参。函数的前提条件和运行结果的要求如下：

```

// 前提条件：流 fileStream 已经通过调用 open 函数同一个文件相关联。
// 文件中包含一系列的整数。
// 运行结果：同流 fileStream 相关联的文件中的整数，按照每行一个数的格式被写到屏幕上。
// （该函数的执行没有关闭文件。）

```



## 12.2 I/O 流工具

你将看到的那些写在四开纸面上的文字，像一条淌过青草地的干净溪流。

理查德·布林斯莱·谢立丹，《造谣学校》

### 使用文件名输入

现在，我们已经学习了如何将实际的输入和输出文件名写到程序代码中。下面这个例子是通过将文件名作为 `open` 函数的参数来打开文件的。

```
inStream.open("infile.txt");
```

也可以通过键盘输入文件名来实现，如下列代码所示：

```
char fileName[16];
ifstream inStream;
cout << "Enter file name (maximum of 15 characters):\n";
cin >> fileName;
inStream.open(fileName);
```

作为 C  
字符串

作为  
string  
对象

代码将文件名作为一个 C 字符串读入。成员函数 `open` 将该 C 字符串作为参数。`open` 函数不能使用 `string` 类型的变量作为参数，C++ 中不存在能将 `string` 类型的变量强制转换为 C 字符串的运算符。不过，作为一个可选办法，我们可以将文件名读到一个 `string` 类型的变量中，并调用 `string` 类的成员函数 `c_str()` 得到调用 `open` 函数所需要的 C 字符串。代码如下所示：

```
string fileName;
ifstream inStream;

cout << "Enter file name:\n";
getline(cin, fileName);
inStream.open(fileName.c_str());
```

当使用 `string` 类型的变量存储文件名时，对文件名的字符长度不会有限制。<sup>3</sup>

### 使用流函数进行格式化输出

在程序中可以通过命令控制输出到文本文件或者屏幕上的文本内容的格式，比如项目与项目之间的空格数以及小数点后的位数等。例如在第 1 章中，给出了如下的“魔法公式”来输出钱的数目：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

现在我们来介绍一下这些格式化输出文本的范式。

首先介绍可以在任意输出流中使用的格式化命令。文件关联的输出流有着与 `cout`

<sup>3</sup> `iostream` 库不能兼容 `string` 类型的原因是，`iostream` 是在 `string` 类型之前加入到 C++ 库中的。

中相同的成员函数。假如 `outStream` 是一个文件输出流 (`ofstream` 类型), 就可以用相同的方法来将输出格式化。

```
outStream.setf(ios::fixed);
outStream.setf(ios::showpoint);
outStream.precision(2);
```

为了解释这个奇妙的格式, 我们将逆序一个个地讨论这些成员函数。

每个输出流都有一个名为 `precision` 的成员函数。当程序中出现了一次对 `precision` 函数的调用后, 比如上面 `outStream` 流的调用, 那么从第一次调用后起, 程序中所有带小数点的数字在输出时都将被写成只有两个有效数字或者小数点后带有两位数字, 这取决于所使用的 C++ 编译器。下面是在设置两位有效数字后, 在某一编译器上的可能输出格式:

```
23.      2.2e7      2.2      6.9e-1      0.00069
```

下面这些数字是设置小数点后两位数字之后的某编译器上的可能输出格式:

```
23.56      2.26e7      2.21      0.69      0.69e-4
```

本书假定编译器设置小数点后两位数字。当然, 也可以通过设置与 2 不同的参数来得到不同的精度。

**setf 标记** 每个输出流都含有 `setf` 成员函数用于设置一些标记, 这些标记是包含在 `std` 命名空间中的 `ios` 类的常量。每次调用 `setf` 函数的时候, 这些标记都决定了输出流的行为。以下是输出流成员函数 `setf` 的两次调用:

```
outStream.setf(ios::fixed);
outStream.setf(ios::showpoint);
```

这些标记中的每一个都是对输出格式的一种设置。流的输出将按照设置的格式进行。

**ios::fixed 定点符号** 标记 `ios::fixed` 使流在输出时按照定点符号 (`fixed-point notation`) 的形式输出浮点数。该形式是我们平时书写数字时已经习惯的一种格式。假如设置了标记 `ios::fixed` (通过调用函数 `setf`), 那么所有的浮点数 (例如 `double` 类型的数) 在通过流输出的时候都按照我们习惯的格式输出, 而不是按照科学计数法的格式输出。

**ios::showpoint** 标记 `ios::showpoint` 使流在输出浮点数时总是包含小数点。假如要输出的数是 2.0, 那么在输出时将会输出 2.0 而不是将它简化为 2。也就是说, 即使小数点后的位数都为 0, 在输出时也要包含该小数点。在示例 12.5 中给出了一些常用的标记及其功能的说明。

在程序中可以通过调用一次函数 `setf` 来设置多个标记, 使用 “|” 符号来连接几个不同的标记, 如下所示<sup>4</sup>:

```
outStream.setf(ios::fixed | ios::showpoint | ios::right);
```

<sup>4</sup> “|” 是位运算符, 实际上是通过将标记的设置与一个位掩码逐个进行或运算得到的, 不过这种底层的细节并不需要掌握。

示例 12.5 `setf` 函数的格式化标记

标记	设置该标记后的效果	默认情况
<code>ios::fixed</code>	不按科学计数法格式输出浮点数（设置该标识后将自动地取消 <code>ios::scientific</code> 标记的设置）	未设置
<code>ios::scientific</code>	浮点数按照科学计数法的格式输出（设置该标识后自动地取消 <code>ios::fixed</code> 标记的设置）。假如 <code>ios::fixed</code> 和 <code>ios::scientific</code> 标记都没有设置，则由操作系统来决定输出的每个数的格式	未设置
<code>ios::showpoint</code>	对于浮点数总是显示小数点及其后的 0。假如未设置该标记，则小数点之后是 0 的数在输出时可能只会省略掉部分	未设置
<code>ios::showpos</code>	输出时在正整数前加上“+”号	未设置
<code>ios::right</code>	设置该标记并且通过调用成员函数 <code>width</code> 设置了某些值的显示宽度后，在输出时数值将会显示在设置的宽度值的最右边。即在输出时对于小于宽度的位在左边补足空格（设置该标记后自动取消对 <code>ios::left</code> 的设置）	未设置
<code>ios::left</code>	设置该标记并且通过调用成员函数 <code>width</code> 设置了某些值的显示宽度后，在输出时数值将会显示在设置的宽度值的最左边，即在输出时对于小于宽度的位在右边补足空格（设置该标记后自动取消对 <code>ios::right</code> 的设置）	未设置
<code>ios::dec</code>	显示输出数的进制（八进制数前加 0，十六进制数前加 0x）	未设置
<code>ios::oct</code>	整数按照八进制的格式输出	未设置
<code>ios::hex</code>	整数按照十六进制的格式输出	未设置
<code>ios::uppercase</code>	对于浮点数，在按照科学计数法输出时，用大写的 E 来代替小写的 e。对于十六进制的数将使用大写字母输出	未设置
<code>ios::showbase</code>	显示输出数的进制（八进制数前加 0，十六进制数前加 0x）	未设置

输出流除了 `precision` 和 `setf` 外，还有其他的成员函数。一个经常使用的格式化函数是 `width`。比如，下面用 `cout` 来调用 `width` 函数的代码：

```
cout << "Start Now";
cout.width(4);
cout << 7 << endl;
```

这段代码运行的结果是在屏幕上出现以下内容：

```
Start Now  7
```

这里输出的内容在字母 `w` 和数字 `7` 之间有三个空格。`width` 函数会通知流在输出某一项时使用多少个空格位置。在上述的情况下，数字 `7` 只占用了一位，而设置的显示宽度为四位，所以有三个空位被空格填充了。假如输出需要更多的空格位，可以通过为 `width` 函数指定合适的数值来实现。无论向 `width` 函数传入了什么参数，整个的数值都会被输出。

**ios 类**

在 `ios` 类中定义了一些很重要的常量，比如 `ios::app`（用来表示向一个文件追加内容），还包括在示例 12.5 中列出的一些标记。`ios` 类定义在输出流的库中，例如 `<iostream>` 和 `<fstream>`。使 `ios` 类及类中的常量在程序中可用的一种方法是：

```
#include <iostream> // 或者 #include <fstream>, 或者两个都使用。
using std::ios;
```

**unsetf**

设置的任何标记都可以被取消。可以使用函数 `unsetf` 来取消对一个标记的设置。比如，如下代码将会取消输出到流 `cout` 上的正整数前的“+”号标记：

```
cout.unsetf(ios::showpos);
```

一旦一个标记被设置了，它在取消该设置前一直有效。重新设置 `precision` 之前，上次调用 `precision` 时标记的作用一直有效。注意成员函数 `width` 有些不同，调用一次 `width` 函数仅对将要输出的下一项有效。假如想输出 12 个宽度为 4 的数，那么必须调用 `width` 函数 12 次。假如觉得这么做太麻烦的话，可以使用在下节讲述的控制符函数 `setw` 来实现。

**控制符****控制符**

一个控制符（manipulator）实际上就是一个函数，我们只是没有按照传统的方式叫它而已。控制符通常放置在插入运算符“<<”之后，就像控制符函数是一个要被输出的变量一样。与传统的函数相同，控制符可以有参数，也可以没有参数。我们在前面的章节中早已使用了一个控制符——`endl`。本节将讨论 `setw` 和 `setprecision` 这两个控制符。

**setw**

控制符 `setw` 和成员函数 `width`（以前已经出现过）的作用一样。我们可以通过在插入运算符“<<”之后添加 `setw` 来实现对控制符 `setw` 的调用，就好像它被输出流输出一样，但实际上这么做将调用成员函数 `width`。比如，下列代码将按照指定的宽度显示数字 10、20 和 30：

```
cout << "Start" << setw(4) << 10
    << setw(4) << 20 << setw(6) << 30;
```

上面的代码将产生如下的输出：

```
Start  10  20   30
```

（在 10 之前有两个空格，在 20 之前有两个空格，在 30 之前有四个空格。）

像成员函数 `width` 一样，`setw` 的一次调用仅对下一个要输出的项起作用，但是在输出时包含多项 `setw` 的调用要比多次调用 `width` 函数容易些。

**setprecision**

控制符 `setprecision` 与成员函数 `precision`（以前已经出现过）的作用相同。不过 `setprecision` 也是通过写在插入运算符“<<”之后实现调用的，与控制符 `setw` 的调用相同。比如下面的代码将会按照控制符 `setprecision` 设置的精度位数输出数据：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout << "$" << setprecision(2) << 10.3 << endl
    << "$" << 20.5 << endl;
```

上面代码的输出如下：

```
$10.30
$20.50
```

使用控制符 `setprecision` 设置小数点的位数的效果和调用成员函数 `precision` 的作用相同。该项设置在下次调用 `setprecision` 或者 `precision` 之前将一直有效。

<iomanip>

要想在程序中使用控制符 `setw` 或者 `setprecision`，必须包含如下语句：

```
#include <iomanip>
using namespace std;
```

或者在程序中指定控制符的名字和命名空间，如下所示：

```
#include <iomanip>
using std::setw;
using std::setprecision;
```

## 保存设置的标记

一个函数的调用不应该引发人们不希望出现的副作用。比如，输出一定数量的钱的函数中可能会包含下面的语句：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

函数调用结束后，上述对输入/输出格式的设置仍然有效。假如不希望存在这种副作用，可以保存初始的设置并在需要的时候恢复。

函数 `precision` 已经有一个没有参数的重载版本了，没有参数的 `precision` 函数的返回值就是当前的精度设置，在需要使用该值恢复设置。

使用 `setf` 函数设置的标记是容易保存和恢复的。成员函数 `flags` 被重载以便保存这些标记的位置和在需要的时候恢复。成员函数 `cout.flags()` 返回一个包含所有标记设置的 `long` 类型的长整数值，该长整数值可以作为 `cout.flags` 的参数值来恢复原来的标记值。这种做法对于文件的输出流和 `cout` 同样有效。

比如，一个保存和恢复这些设置的函数体如下所示：

```
void outputStuff(ofstream& outStream)
{
    int precisionSetting = outStream.precision();
    long flagSettings = outStream.flags();
    outStream.setf(ios::fixed);
    outStream.setf(ios::showpoint);
    outStream.precision(2);
    // 在这里随意做些什么。
    outStream.precision(precisionSetting);
    outStream.flags(flagSettings);
}
```

另外一种恢复设置的方法是：

```
cout.setf(0, ios::floatfield);
```

调用参数如上面所示的成员函数 `setf` 将会恢复 `setf` 函数的默认设置。注意恢复的是默认设置，而不是上次改变之前的设置。大家要注意默认设置具体的值取决于操作系统。最后需要强调的一点是，这么做不会改变 `precision` 原来的设置，也不会改变不用 `setf` 函数设置的参数。

### 更多的输出流成员函数

示例 12.6 总结了类 `ostream` 中一些用作格式化的成员函数和运算符。要使用这些运算符，不要忘了在程序中包含下面这些语句（或者其他等效的语句）：

```
#include <iomanip>
using namespace std;
```

示例 12.6 格式化类 `ostream` 的输出格式的工具

功能	描述	对应的运算符
<code>setf(ios_Flag)</code>	设置在示例 12.5 中描述的标记	<code>setiosflags(ios_Flag)</code>
<code>unsetf(ios_Flag)</code>	取消标记的位置	<code>resetiosflags(ios_Flag)</code>
<code>setf(0, ios::floatfield)</code>	恢复标记的默认设置	无
<code>precision(int)</code>	设置浮点数输出的精度	<code>setprecision(int)</code>
<code>precision()</code>	返回当前的精度设置数值	<code>precision()</code>
<code>width(int)</code>	设置输出的数字的宽度，仅对下一次输出有效	<code>setw(int)</code>
<code>fill(char)</code>	当输出域的宽度大于输出所需宽度时，指定在不足位填充的字符，默认为空	<code>setfill(char)</code>

### 自测练习

8. 下面代码的运行结果是什么？

```
cout << " ";
cout.width(5);
cout << 123
    << " " << 123 << " " << endl;
cout << " " << setw(5) << 123
    << " " << 123 << " " << endl;
```

9. 下面代码的运行结果是什么？

```
cout << " " << setw(5) << 123;
cout.setf(ios::left);
cout << " " << setw(5) << 123;
```

```
cout.setf(ios::right);
cout << "*" << setw(5) << 123 << "*" << endl;
```

10. 下面代码的运行结果是什么?

```
cout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
cout.setf(ios::showpos);
cout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
cout.unsetf(ios::showpos);
cout.setf(ios::left);
cout << "*" << setw(5) << 123 << "*"
    << setw(5) << 123 << "*" << endl;
```

11. 下面代码的运行结果是什么?

```
ofstream fout;
fout.open("stuff.txt");
fout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
fout.setf(ios::showpos);
fout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
fout.unsetf(ios::showpos);
fout.setf(ios::left);
fout << "*" << setw(5) << 123 << "*"
    << setw(5) << 123 << "*" << endl;
```

12. 下面代码的运行结果是什么? (假定该行代码包含在一个正确完整的程序中, 并且使用了必要的 include 指令和 using 指令。)

```
cout << "*" << setw(3) << 12345 << "*" << endl;
```

### 示例：整理文件格式

示例 12.7 中程序的功能是从文件 rawdata.txt 中读取输入, 然后将文件内容按照比较整齐的格式输出到屏幕, 同时也将相同格式的内容写到文件 neat.txt 中。程序运行后, 文件 rawdata.txt 中的数字将被格式化指令按照整齐的格式输出到文件 neat.txt 中。写到 neat.txt 文件中的数字格式是每个数字占 12 个字符, 每行一个数字。这意味着在每个数字的前面都有一定数量的空格以保证每个数字均占满 12 个字符。数字是按照一般的格式书写的, 而不是按照科学计数法的格式书写。每个数字在小数点后均有 5 位, 同时都包含 “+”、“-” 标记。程序使用了一个名为 makeNeat 的函数, 该函数包含文件输入流和文件输出流的形参。

## 示例 12.7 格式化输出

```

1  // 读取文件 rawdata.dat 中的所有数字，然后将这些数字按照整齐
2  // 输出到屏幕和文件 neat.txt 文件中。
3  #include <iostream>
4  #include <fstream>
5  #include <cstdlib>
6  #include <iomanip> ← 需要 setw。

7  using std::ifstream;
8  using std::ofstream;
9  using std::cout;
10 using std::endl;
11 using std::ios;
12 using std::setw;

13 void makeNeat(ifstream& messyFile, ofstream& neatFile,
14               int numberAfterDecimalpoint, int fieldWidth);
15 // 前提条件：流 messyFile 和 neatFile 已经和两个不同的文件相关联。
16 // 文件 messyFile 中只包含浮点数。
17 // 运行结果：与流 messyFile 相关联的文件的浮点数已经被写到屏幕上，以及
18 // 与流 neatFile 相关联的文件中。
19 // 浮点数按照一行写一个数，小数点后固定位数的格式（而不是科学计数法的格式）显示，
20 // 小数点后的位数为 numberAfterDecimalpoint；
21 // 在每个浮点数之前都有一个 "+" 或 "-" 符号标记，
22 // 每个数字的宽度都是 fieldWidth（该函数没有关闭文件）。

23 int main( )
24 {
25     ifstream fin;
26     ofstream fout;

27     fin.open("rawdata.txt");
28     if (fin.fail( ))
29     {
30         cout << "Input file opening failed.\n";
31         exit(1);
32     }

33     fout.open("neat.txt");
34     if (fout.fail( ))
35     {
36         cout << "Output file opening failed.\n";
37         exit(1);
38     }

39     makeNeat(fin, fout, 5, 12);

40     fin.close( );
41     fout.close( );
42     cout << "End of program.\n";
43     return 0;
44 }

46 // 使用库 <iostream>、<fstream> 和 <iomanip>。

```

流的参数一定是引用参数。



```
47 void makeNeat(istream& messyFile, ofstream& neatFile,
48               int numberAfterDecimalpoint, int fieldWidth)
49 {
50     neatFile.setf(ios::fixed);
51     neatFile.setf(ios::showpoint);
52     neatFile.setf(ios::showpos);
53     neatFile.precision(numberAfterDecimalpoint);
54
55     cout.setf(ios::fixed);
56     cout.setf(ios::showpoint);
57     cout.setf(ios::showpos);
58     cout.precision(numberAfterDecimalpoint);
59
60     double next;
61     while (messyFile >> next)
62     {
63         cout << setw(fieldWidth) << next << endl;
64         neatFile << setw(fieldWidth) << next << endl;
65     }
66 }
```

运算符 setf 和 precision 对文件输出流和 cout 的作用是一样的。

只要还有下一个数要读，就返回 true。

对文件输出流和 cout 的作用相同。

示例运行结果

rawdata.txt  
(程序不会改变其内容)

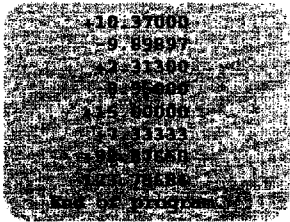
```
10.37      -9.89897
2.313      -8.950  15.0

7.33333    92.8765
-1.237568432e2
```

neat.txt  
(程序执行之后)

```
+10.37000
-9.89897
+2.31300
-8.95000
+15.00000
+7.33333
+92.87650
-123.75684
```

屏幕输出



示例：编辑文本文件

这里将要演示的程序是一个十分简单的编辑文本文件的例子。该程序可以根据现有的讲述 C 语言编程的材料，以一种非常简单的方式自动生成 C++ 编程的文本材料。

程序从一个讲述 C 语言编程优势的文本文件中读取文字，将它改编成讲述采用 C++ 语言编程优势的文字，保存到另外一个文件中。讲述 C 语言编程优势的文件是 cad.txt，将讲述 C++ 语言编程优势的内容保存到文件 cppad.txt 中，如示例 12.8 所示。该程序简单地读取 cad.txt 文件中的每个字符并将这些字符拷贝到文件 cppad.txt 中。拷贝的字符除了在碰到大写的字母 ‘C’ 时将它改写为字符串 “C++” 外，其他的没有变化。程序只要在读取输入文件的过程中发现字符 ‘C’，就认为它指的就是 C 编程语言。我们假设这种改变是合理的。

程序从输入文件读取字符和将字符写到输出文件的过程中，换行符也被读取并处理。在处理过程中，换行符 ‘\n’ 与其他字符被同等对待。从输入文件中读取字符使用的是成员函数 get，写入输出文件使用的是插入运算符 “<<”。由于在这个例子中我们想处理空格，所以必须使用成员 get 函数逐个读取输入文件中的字符（而不是使用提取运算符 “>>”）。

---

### 示例 12.8 编辑文件中的文本

---

```

1 // 该程序将会创建一个被称为 cplusplus.txt 的文件，该文件除了将 cad.txt 文件中的
2 // ‘C’ 换成了 “C++” 外，其他与 cad.txt 文件完全相同。
3 // 该程序假定在 cad.txt 文件中出现的大写字母 ‘C’ 都是表示 C 编程语言。
4 #include <fstream>
5 #include <iostream>
6 #include <cstdlib>
7 using std::ifstream;
8 using std::ofstream;
9 using std::cout;

10 void addPlusPlus(ifstream& inStream, ofstream& outStream);
11 // 前提条件：inStream 已经通过调用 open 同输入文件相关联。
12 // outStream 已经通过调用 open 同输出文件相关联。
13 // 运行结果：与流 inStream 相关联的文件内容已经被拷贝到与流 outStream 相关联的文件中。
14 // 拷贝过程中，字符 ‘C’ 被替换成了 “C++”（该函数中没有关闭文件）。

15 int main()
16 {
17     ifstream fin;
18     ofstream fout;

19     cout << "Begin editing files.\n";

20     fin.open("cad.txt");
21     if (fin.fail())
22     {
23         cout << "Input file opening failed.\n";
24         exit(1);
25     }

26     fout.open("cppad.txt");
27     if (fout.fail())
28     {

```

```

29         cout << "Output file opening failed.\n";
30         exit(1);
31     }

32     addPlusPlus(fin, fout);
33     fin.close();
34     fout.close();

35     cout << "End of editing files.\n";
36     return 0;
37 }
38 void addPlusPlus(istream& inStream, ostream& outStream)
39 {
40     char next;

41     inStream.get(next);
42     while (! inStream.eof())
43     {
44         if (next == 'C')
45             outStream << "C++";
46         else
47             outStream << next;
48         inStream.get(next);
49     }
50 }

```

### 示例运行结果

#### **cad.txt**

(程序不会改变其内容)

```

C is one of the world's most modern
programming languages. There is no
language as versatile as C, and C
is fun to use.

```

#### **cppad.txt**

(程序执行之后)

```

C++ is one of the world's most
modern programming languages. There
is no language as versatile as C++,
and C++ is fun to use.

```

### 屏幕输出

```

Begin editing files.
End editing files.

```

## 12.3 流的继承层次：继承概述

组织类的一个有效方式是通过“派生”关系将类联系起来。当我们说一个类是由另一个类派生而来时，是指通过向其基类中加入新的特性而得到一个新的派生类。比如文件输入流类是通过向描述输入流的基类中加入另外的成员函数如 `open` 和 `close` 等派生得到的。`cin` 是输入流的一种，但不属于文件输入流，因为 `cin` 中没有 `open` 和 `close` 成员函数。本节将简单地介绍 C++ 中预定义的流类是如何通过派生类来组织自己的类库的（第 14 章将详细讲述如何运用派生类的概念来定义你自己的类）。

### 流之间的继承

库中预定义的流 `cin` 和文件输入流都是输入流。所以在某种意义上讲，它们是相同的。比如，对这两种流我们都可以使用提取运算符“>>”。但另一方面，文件输入流能够通过调用成员函数 `open` 同一个文件相关联，可是 `cin` 流中没有 `open` 成员函数。一个文件输入流是与 `cin` 类似但又不完全相同的一种流。文件输入流是 `ifstream` 类的对象，而 `cin` 则是 `istream`（没有“f”）类的对象。类 `ifstream` 和 `istream` 是不同的但又密切关联的两个类，类 `ifstream` 是从类 `istream` 中派生而来的。下面是对于派生的详细解释。

#### 派生类

当我们说类 D 是从类 B 中派生而来的时候，就表示类 D 具有类 B 中的所有特性，同时类 D 也有自己独特的特性。比如，任意的 `istream`（没有“f”）类型的流可以使用提取运算符“>>”。类 `ifstream`（有“f”）是类 `istream` 的派生类，因此 `ifstream` 类型的流也可以使用提取运算符“>>”。类 `ifstream` 的对象具有 `istream` 对象的所有特性。类 `ifstream` 的对象也可以被看作是类 `istream` 的对象。

不过，在 `ifstream` 类中也加入了新的特性，`ifstream` 类的对象有着更多的功能。比如，在 `ifstream` 类中加入了新的成员函数 `open`。`cin` 流是 `istream` 类的对象而不是 `ifstream` 类的对象，`cin` 流不能使用 `open` 函数打开。`ifstream` 类和 `istream` 类的关系是不对称的。`ifstream` 类的对象都是 `istream` 类的对象（一个文件输入流是一个输入流），但是 `istream` 类的对象不一定是 `ifstream` 类的对象（`cin` 是 `istream` 类的对象，但不是 `ifstream` 类的对象）。

派生类的概念比较常见，日常生活中的一个例子可以帮助我们理解这个概念。敞篷车（convertibles）类是汽车类的一个派生类。每辆敞篷车都是一辆汽车，但敞篷车不仅仅是一辆汽车，它还是具有其他汽车所不具有的特殊属性的汽车——可以打开车的顶棚（也可以认为敞篷车加入了新的特性——`open` 函数）。

假如类 D 是从类 B 派生而来，则类 D 的每个对象也都是类 B 的对象。就像敞篷车也是汽车，文件输入流（`ifstream` 类的对象）也是输入流（类 `istream` 的对象）一样。所以假如我们使用 `istream` 而不是将 `ifstream` 作为参数的形参的话，那么在调用该函数时，可以有更多的对象符合条件。考虑下面的两个函数定义，两个函数仅在函数参数的类型上存在不同（函数名也不同）。

```
void twoSumVersion1(istream& sourceFile) // 带一个“f”的 ifstream,
{
    int n1, n2;
    sourceFile >> n1 >> n2;
    cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;
}
```

和

```
void twoSumVersion2(istream& sourceFile) // 不带“f”的 istream.
{
    int n1, n2;
    sourceFile >> n1 >> n2;
    cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;
}
```

对于函数 twoSumVersion1, 参数必须是 ifstream 类型的对象。假如 fileIn 是一个文件关联的文件输入流, 那么语句

```
twoSumVersion1(fileIn);
```

是合法的, 但是语句

```
twoSumVersion1(cin); // 非法
```

是非法的, 因为 cin 不是 ifstream 类型的对象。cin 只是一个流, 它是 istream 类的对象, 而不是一个文件输入流。

函数 twoSumVersion2 更通用一些。下面的表达式均合法:

```
twoSumVersion2(fileIn);
twoSumVersion2(cin);
```

通过比较我们可以看到: 在条件允许的情况下, 使用 istream 而不是 ifstream 作为参数类型。在选择参数类型时, 尽量使用一般意义上的类型。(推广到一般生活中: 你可能喜欢拥有一辆敞篷车, 但是不希望自己的车库中只能放敞篷车。假如从朋友处借来了一辆轿车, 你仍然想将这辆车也放在自己的车库中。)

不能总是用 istream 类型的参数代替 ifstream 类型的参数。假如定义了一个只有一个 istream 类型参数的函数, 那么该参数就只能使用 istream 类的成员函数。而且, 它不能使用 open 和 close 成员函数。假如不能将对成员函数 open 和 close 的调用都放在函数体外的话, 那么只能使用 ifstream 类型的参数。

ostream 和  
ofstream

现在我们已经介绍过两个输入流类了: istream 及派生类 ifstream。输出流的情况与输入流相似。第1章介绍了 ostream 中的两个输出流 cout 和 cerr, 本章介绍了类 ofstream (有“f”)中的文件输出流。类 ostream 是所有输出流类的基类。与 cout 和 cerr 不同, 文件输出流被声明为 ofstream 类的对象。文件输出流类 ofstream 是从类 ostream 中派生而来的。比如, 下面的函数向作为它的参数的输出流中写入单词“Hello”:

```
void sayHello(ostream& anyOutputStream)
{
    anyOutputStream << "Hello";
}
```

在下面的代码中，第一次调用函数向屏幕输出“Hello”；第二次调用函数向文件名为 `afile.txt` 的外部文件中写入“Hello”。

```
ofstream fout;
fout.open("afile.txt");
sayHello(cout);
sayHello(fout);
```

作为类 `ofstream` 对象的文件输出流一定是 `ostream` 的对象。所以上述调用是合法的。

通常使用继承和家庭关系来对比讨论派生类。假如类 `D` 是类 `B` 的派生类，则类 `D` 被称为类 `B` 的**子类或者派生类**，类 `B` 被称为类 `D` 的**父类或者基类**。派生类继承了基类的成员函数。比如，敞篷车继承了汽车有四个轮子的事实，每个文件输入流都从基类输入流中继承了提取运算符“>>”，这也就是派生类通常被称为继承类（或者子类）的原因。

### 更加通用的流参数

假如你想定义一个由输入流作为参数的函数，而且在某些情况下希望输入流参数为 `cin`，而在另外一些情况下希望输入流参数为一个文件输入流时，那么可以使用 `istream`（没有“f”）类型的变量来作为函数的形参。不过，一个文件输入流，即使是作为 `istream` 类型的参数输入，在声明时仍要将它声明为 `ifstream`（有“f”）类型。

同样，假如你想定义一个由输出流作为参数的函数，而且在某些情况下希望输出流参数为 `cout`，而在另外一些情况下希望输出流参数为一个文件输出流时，那么可以使用 `ostream` 类型的变量来作为函数的形参。不过，一个文件输出流即使作为 `ostream` 类型的参数来使用，在声明时仍要把它声明为 `ofstream`（有“f”）类型。你不能打开或者关闭 `istream` 或者 `ostream` 类型的流，可以在函数调用它们之前打开，在函数调用完成之后再关闭它们。

流类 `istream` 和 `ostream` 定义在 `iostream` 库中，而且放置在 `std` 命名空间中。在代码中使它们可用的一种方法是：

```
#include <iostream>
using std::istream;
using std::ostream;
```

### 示例：另一版本的 `newLine` 函数

作为如何使流函数更通用的一个例子，考虑我们在示例 9.2 中定义的函数 `newLine`。该函数只对从键盘的输入有效，即只能处理 `cin` 流的输入。在示例 9.2 中的例子没有参数。下面我们重写 `newLine` 函数，重写的 `newLine` 函数带有一个类型为 `istream` 的形参。

```
// 使用 <iostream>。
```

```

void newLine(istream& inStream)
{
    char symbol;
    do
    {
        inStream.get(symbol);
    } while (symbol != '\n');
}

```

现在，假定程序中包含了上述新版本的 `newLine` 函数。假如程序从一个被称为 `fin` 的输入流（已经同个输入文件相关联）中得到输入，那么下列代码将忽略正在读取的输入文件的当前行的所有字符。

```
newLine(fin);
```

假如程序已经从键盘读取了输入，那么下列代码将忽略通过键盘输入的输入行的所有字符。

```
newLine(cin);
```

假如你的程序只能使用上面所说的重写版本的 `newLine` 函数，其需要输入一个形参，如 `fin` 或者 `cin`，那么在调用该函数时一定要得到流的名称，即使输入流是 `cin`。考虑到 C++ 的函数重载特性，可以将两个版本的 `newLine` 函数（在示例 9.2 中给出的没有参数的函数和我们刚刚定义的带有一个 `istream` 形参的函数）声明放在一个函数体中来实现。那么，下面两个 `newLine` 函数的调用是等价的。

```
newLine(cin);
```

和

```
newLine( );
```

此时，不需要两个版本的 `newLine` 函数。带一个 `istream` 形参的函数完全可以满足要求。不过，由于键盘输入很常见，许多程序员更希望有一个没有参数的函数版本来进行键盘输入。

同时具有两个重载的 `newLine` 函数版本的一个可选方案是使用默认参数（在第 4 章中讲述过）。下面是我们第三次重写的 `newLine` 函数。

```

// 使用 <iostream>。
void newLine(istream& inStream = cin)
{
    char symbol;
    do
    {
        inStream.get(symbol);
    } while (symbol != '\n');
}

```

假如这样调用函数

```
newLine( );
```

则函数的形参将使用默认的参数 `cin`。假如按下列方式调用函数

```
newLine(fin);
```

则函数的形参将使用参数 `fin`。

也可以用我们在第 9 章中学习过的函数 `ignore` 来代替 `newLine` 函数的使用。函数 `ignore` 既是文件输入流的成员，又是 `cin` 的成员函数。

13. 流 `cin` 是什么类型的流？流 `cout` 又是什么类型的流？

14. 定义一个名为 `copyChar` 的函数，该函数是一个输入流的形参。当 `copyChar` 被调用后，将从传入的形参的输入流中读取一个字符，并将读到的字符输出到屏幕上。要求 `copyChar` 函数既可以对文件输入流进行操作，又可以对 `cin` 进行操作（假如参数为一个文件输入流，则在函数调用前，流已经与文件关联，在该函数中没有打开或者关闭文件的操作）。比如，下面的代码中共调用函数 `copyChar` 两次，第一次调用是将字符从文件 `stuff.txt` 中读出并输出到屏幕，第二次调用则是从键盘读取一个字符并输出到屏幕。

```
ifstream fin;
fin.open("stuff.txt");
copyChar(fin);
copyChar(cin);
```

15. 定义一个名为 `copyLine` 的函数，该函数有一个输入流的形参。当 `copyLine` 被调用后，将从传入的形参的输入流中读取一行文本，并将读到的文本输出到屏幕上。要求 `copyLine` 函数既可以对文件输入流进行操作，又可以对 `cin` 进行操作（假如参数为一个文件输入流，则在函数调用前，流已经与文件关联，在该函数中没有打开或关闭文件的操作）。比如，下面的代码中共调用函数 `copyLine` 两次，第一次调用是从文件 `stuff.txt` 中读出一行并输出到屏幕，第二次调用则是从键盘读取一行文本并输出到屏幕。

```
ifstream fin;
fin.open("stuff.txt");
copyLine(fin);
copyLine(cin);
```

16. 定义一个名为 `sendLine` 的函数，该函数有一个输出流的形参。当 `sendLine` 被调用后，将从键盘的输入读取一行文本，并将读到的文本输出到函数传入的形参的输出流中。要求 `sendLine` 函数既可以将文本输出到文件流中，又可以输出到 `cout` 中（假如参数为一个文件输出流，则在函数调用前，流已经与文件关联，在该函数中没有打开或关闭文件的操作）。比如，下面的代码中共调用函数 `sendLine` 两次，第一次调用是将键盘输入的一行文本输出到文件 `morestuff.txt` 中，第二次调用则是将键盘输入的一行文本输出到屏幕。



```

ofstream fout;
fout.open("morestuff.txt");
cout << "Enter 2 lines of input:\n";
sendLine(fout);
sendLine(cout);

```

17. 判断如下说法的正误。假如说法是错误的，请改正。请详细解释各种可能的情况。

使用类 `ifstream` 或 `ofstream` 作为形参声明的函数，在调用时可以分别使用 `istream` 或 `ostream` 类型的变量作为实参来调用。

### 使用类 `stringstream` 解析字符串

接下来我们通过类 `stringstream` 再看一个继承的例子。这个类继承自类 `iostream`，而 `iostream` 顺序继承自 `istream`。`stringstream` 使得运算符 `>>` 可以作用于一个字符串，这个运算符是从 `istream` 类继承过来的。这意味着可以像处理文件、键盘输入或者控制台输出那样操作字符串。`stringstream` 可以用来从其他的数据类型创建新的字符串，或者从各种不同数据类型的数据中读取字符串。为了使用它，必须使用 `include` 语句包含它：

```

#include <sstream>
using std::stringstream;

```

然后我们创建一个 `stringstream` 的对象：

```
stringstream ss;
```

假如想将 `stringstream` 对象初始化为空字符串或者想清空其中的内容，可以使用下面的语句。函数 `clear` 在清除内容的同时也将清除 `stringstream` 对象中保存的错误状态。`str` 函数将 `stringstream` 对象初始化为某字符串。下面的代码将对象初始化为空字符串，你也可以利用相同的方法将对象初始化成其他字符串：

```

ss.clear();
ss.str("");

```

想输出字符串时，可以先使用插入运算符将字符串放入 `stringstream` 对象，然后再将对象输出到 `cout`。与输出到屏幕不同的是，要先将内容输出到 `stringstream` 对象。比如有一个值为 10 的 `int` 型变量 `num` 和值为 'x' 的字符型变量 `c`，下面的代码将 "x 10" 插入到 `ss` 中：

```
ss << c << " " << num;
```

使用函数 `str()` 可以得到 `stringstream` 对象中的字符串值：

```

string s;
s = ss.str(); // 将 s 设置成字符串 "x 10"。

```

想从字符串中析取变量的时候，可以先将字符串放入 `stringstream` 对象，然后利用 `cin` 从 `stringstream` 对象中读取变量。变量的值将不再从键盘读入，而是从字符串读入。举个例子，假设一个 `stringstream` 对象 `ss` 的内容是 "x 10"，我们可以使用下面的代码将变量读取出来：

```
// 假如 ss = "x 10", 那么 c 的值被设置为 'x', num 的值被设置为 10。
ss >> c >> num;
```

一个最简单的应用场景，就是将一个数字值转化为字符串型数值。

示例 12.9 给出了一个使用 `stringstream` 类的例子。假设有一个字符串，内容是一个人的名字和他的分数，比如 Luigi 有三项分数：70、100 和 90，对应的字符串是“Luigi 70 100 90”。使用 `stringstream` 解析这个字符串，将名字放在字符串变量中，将分数放在整型变量中并相加求平均值，然后将名字和平均分插入到 `stringstream` 对象中。这个例子最终的结果是“Name: Luigi Average: 86”。

### 示例 12.9 演示 `stringstream` 类

---

```
1 // 演示 stringstream 类的程序。
2 // 假设有一个包含名字和分数的字符串。
3 // 程序使用 stringstream 解析名字和分数，
4 // 将分数放入整型变量并求平均值，
5 // 然后将名字和平均分放入新的字符串。
6 #include <iostream>
7 #include <string>
8 #include <sstream>

9 using namespace std;

10 int main()
11 {
12     stringstream ss;
13     string scores = "Luigi 70 100 90";

14     // 清空 stringstream。
15     ss.str("");
16     ss.clear();

17     // 将分数放入 stringstream。
18     ss << scores;
19
20     // 解析出名字和分数。
21     string name = "";
22     int total = 0, count = 0, average = 0;
23     int score;
24     ss >> name; // 读取名字。
25     while (ss >> score) // 读取到字符串的末尾。
26     {
27         count++;
28         total += score;
29     }
30     if (count > 0)
31     {
32         average = total / count;
33     }

34     // 清空 stringstream。
```

```

35     ss.clear();
36     ss.str("");
37     // 将名字和平均分放入 stringstream。
38     ss << "Name: " << name << " Average: " << average;

39     // 输出字符串。
40     cout << ss.str() << endl;

41     return 0;
42 }

```

### 示例运行结果

```
Name: Luigi Average: 86
```

### 自测练习

18. 利用 stringstream 将整型变量 num = 10 转化为字符串型变量 s。
19. 利用 stringstream 将字符串 s = "10" 转化为整型变量 num。
20. 下面的代码将计算若干以字符串形式保存的数字的加和。实现方法是先将数字都放入一个 stringstream 对象，然后使用 getline 函数将由逗号分隔的数字字符串都解析出来，另一个 stringstream 对象将数字字符串转化成 double 型。但是最终得出的加和是错误的，请问为什么发生了错误？

```

stringstream ssList, ssNum;
string numbers = "1.1, 1.2, 1.3";

double total = 0;
double num;

ssList.clear();
ssList.str(numbers);

string field;
while (getline(ssList, field, ','))
{
    ssNum.str(field);
    ssNum >> num;
    total += num;
}
cout << total << endl;

```

## 12.4 随机文件存取

任何时间，任何地点。

遭遇挑战的常见反应

本章前面章节中讨论的都是顺序存取文件流，这也是在 C++ 中文件存取经常使用的流。不过，有一些应用程序，在非常大的数据库中需要对记录进行快速存取，类似这样的应用程序需要对文件中的某些特定部分进行随机存取。这种应用程序可能使用特定的数据库软件来开发更好。但是，有可能恰好你就被布置了这样一项任务——用 C++ 来写这样一个包，或者你可能对用 C++ 如何编写这类需求感到好奇。在 C++ 中，确实提供了随机访问文件的功能，你既可以从一个随机的位置读取文件，也可以向一个文件的随机位置写入。本节将对 C++ 中的这一随机文件存取功能进行简单介绍。本节内容不是随机文件存取的一个完整的学习指南，但向读者介绍了将要使用的主要流类及将会遇到的一些重要问题。

在 C++ 中，假如想让一个文件既有读权限又有写权限，那么可以使用在 `<fstream>` 库中定义的 `fstream` 类。类 `fstream` 的定义在 `std` 命名空间中。

打开一个文件并将文件同类 `fstream` 的流相关联的细节，基本上与对类 `ifstream` 和 `ofstream` 讨论过的方法相同，只是 `fstream` 流的 `open` 函数多了一个参数。`fstream` 流的 `open` 函数的第二个参数是用来说明该文件是输入或输出，还是既输入又输出的。比如，对文件 `stuff` 既读又写的程序代码如下所示：

```
#include <fstream>
using namespace std;

int main()
{
    fstream rwStream;
    rwStream.open("stuff", ios::in | ios::out);
```

也可以用下面一行代码来代替上述的最后两行代码：

```
fstream rwStream("stuff", ios::in | ios::out);
```

现在，程序就可以使用流 `fstream` 从文件 `stuff` 中读取，也可以使用相同的流向文件 `stuff` 中写。在由读到写文件或者由写到读文件的转换过程中，不需要关闭再打开该文件，甚至还可以读写文件的任意位置。但是，这带来了另外的问题。

在使用流 `fstream` 对文件进行随机读写时，至少带来两个问题：我们可能使用 `char` 类型的字节或者 `char` 的数组类型来处理文件，此时程序员需要自己进行类型转换；在每次读写文件之前，需要定位文件的指针（指示从哪里开始进行读或写）。

先找到正确的位置然后再用新数据代替旧数据的做法，意味着大部分这样的随机存取输入/输出是通过读写数据记录（以 `struct` 或者 `class` 的形式）来完成的。在每次定位好文件指针后都要读或写一个记录（或者一组记录）。

每个 `fstream` 对象都有一个名为 `seekp` 的成员函数，该函数是用来定位输出内容的指针的，也就是我们希望准备在文件的哪个地方开始写操作。函数 `seekp` 只有一个参数，用来表示开始写的内容的第一个字节的地址。文件中的第一个字节的地址为 0。比如，要想将同 `fstream` 流 `rwStream` 相关的文件的指针定位在第 1000 个字节处，可以用如下的语句：

```
rwStream.seekp(1000);
```

当然，为了准确定位文件的指针，我们需要了解一个记录要占用多少个字节。可以用运算符 `sizeof` 来判断一个类或者一个结构的对象所占用的字节数。实际上，`sizeof` 适用于任何类型、对象或者数值，它返回其参数所占用的字节数。运算符 `sizeof` 是 C++ 编程语言核心的一部分，使用它时不需要另外包含头文件或者库。一些 `sizeof` 运算符的调用如下：

```
sizeof(s) (where s is string s = "Hello");
sizeof(10)
sizeof(double)
sizeof(MyStruct) (where MyStruct is a defined type)
```

上面这些调用都返回一个表示其参数所占字节数的整数值。

在只包含 `MyStruct` 类型记录的文件中，定位写指针到第 100 条 `MyStruct` 类型的记录的方法是：

```
rwStream.seekp(100*sizeof(MyStruct) - 1);
```

成员函数 `seekg` 是用来定位文件的读指针的，标明下一次对文件的读取将会读出什么字节。该函数的用法与 `seekp` 十分相似。

在程序中，我们还可以使用成员函数 `put` 和 `get` 来按照字节写文本到文件 `stuff` 中和从文件 `stuff` 中逐个字节地读文本。也可以使用 `write` 成员函数一次写多个字节到文件中和使用 `read` 成员函数从文件中一次读取多个字节的文本。

理论上，现在我们对随机存取文件输入/输出的操作已经充分了解。但实际上，这只是对该部分的一个概述。本节的目的就是使你随机存取文件有一个概要的了解。假如想要真正地进行随机存取文件这方面的编程，请参考其他更高级和专业的书籍。

## 本章小结

- `ifstream` 类型的流可以通过调用成员函数 `open` 和一个文件进行关联。关联后程序就可以从这个文件中读取输入。
- `ofstream` 类型的流可以通过调用成员函数 `open` 和一个文件进行关联。关联后程序就可以将要输出的内容写到该文件中。
- 应该使用成员函数 `fail` 来检查 `open` 函数的调用是否成功。
- 流成员函数，如 `width`、`setf` 和 `precision` 等，可以用来格式化输出。这些成员函数对于输出到屏幕的流 `cout` 的作用，和它们对文件关联的文件输出流的作用是一样的。
- 函数调用的形参可以是流类型，但是流类型的参数必须是传引用调用。它们不能是传值调用。`ifstream` 类型可以用作文件输入流，`ofstream` 类型可以用作文件输出流（其他类型的可能情况见下个知识点）。

- 假如使用 `istream` (拼写中没有“f”)作为输入流形参的类型,那么与该形参对应的实参可以是 `cin`,也可以是 `ifstream` (拼写中有“f”)类型的文件输入流。假如使用 `ostream` (拼写中没有“f”)作为输出流形参的类型,那么与该形参对应的实参可以是 `cout`、`cerr`,也可以是 `ofstream` (拼写中有“f”)类型的文件输出流。
- 成员函数 `eof` 可以用来检测程序是否将一个输入文件读到了文件尾。
- 利用 `stringstream`,就可以对字符串使用操作文件时使用的输入/输出函数。`stringstream`类为从其他类型的数据创建字符串和从字符串中解析出其他数据类型的值提供了一种简单的方法。

## 自测练习题答案

1. 流 `fin` 和 `fout` 按照如下方法声明:

```
ifstream fin;
ofstream fout;
```

在文件的开始部分还应该包含如下语句:

```
#include <fstream>
```

由于类的声明放在了 `std` 命名空间中,因此还应该在文件中包含如下代码(或者类似的语句):

```
using std::ifstream;
using std::ofstream;
```

或者

```
using namespace std;
```

2. `fin.open("stuff1.txt");`

```
if (fin.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

```
fout.open("stuff2.txt");
```

```
if (fout.fail( ))
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

3. `fin.close( );`  
`fout.close( );`

4. 应该用流 `cout` 来代替流 `outStream`。注意:不需要声明 `cout`,对于 `cout` 无须调用 `open`,也无须关闭 `cout`。

5. 该问题是读文件的一个“起始位置”问题。必须先关闭文件然后再打开文件，重新打开文件后读文件的起始位置在文件的开始。因此可以再重新读文件的第一行。

```
6. 1
   2
   3
   3
```

```
7. void toScreen(istream& fileStream)
{
    int next;
    while (fileStream >> next)
        cout << next << endl;
}
```

```
8. * 123*123*
   * 123*123*
```

上述每个空白处都包含两个空格字符。注意对函数 `width` 和 `setw` 的调用仅对下一次输出有效。

```
9. * 123*123 * 123*
```

每个空白处都包含两个空格字符。

```
10. * 123*123*
    * +123*+123*
    *123 *123 *
```

在第二行的“\*”和“+”之间是一个空格。其他的空白处都是两个空格字符。

11. 写入文件 `stuff.txt` 的内容与自测练习题 10 的答案一样。

```
12. *12345*
```

注意即使当函数 `setw` 指定的位数小于整数的位数时，整个整数位仍会全部输出。

13. `cin` 是 `istream` 类型，而 `cout` 是 `ostream` 类型。

```
14. void copyChar(istream& sourceFile)
{
    char next;
    sourceFile.get(next);
    cout << next;
}
```

```
15. void copyLine(istream& sourceFile)
{
    char next;
    do
    {
        sourceFile.get(next);
        cout << next;
    } while (next != '\n');
}
```

16. void sendLine(ostream& targetStream)

```
{
    char next;
    do
    {
        cin.get(next);
        targetStream << next;
    } while (next != '\n');
}
```

17. 错误。上述情况反过来是正确的。任意 ifstream 类型的流都是 istream 类型，所以 istream 类型的形参可以在函数调用时使用 ifstream 类型的变量作为实参调用，对于 ostream 和 ofstream 的关系与 istream 和 ifstream 的关系类似。

18. int num = 10;  
stringstream ss("");  
ss << num;  
string s = ss.str();

19. int num;  
string s = "10";  
stringstream ss(s);  
ss >> num;

20. 用来将 field 转化为 double 型的 stringstream 类型变量 ssNum，在处理每个字符串之前都要清空自己已经保存的内容。可以循环调用 clear 函数完成这步工作。

```
stringstream ssList, ssNum;
string numbers = "1.1, 1.2, 1.3";

double total = 0;
double num;

ssList.clear();
ssList.str(numbers);

string field;
while (getline(ssList, field, ','))
{
    ssNum.clear();
    ssNum.str(field);
    ssNum >> num;
    total += num;
}

cout << total << endl;
```



## 编程练习

1. 写一个程序，要把一个包含一系列整数的文件中的最大值和最小值输出到屏幕上。读取的文件只包含整数，整数之间用空格或者换行符分隔。
2. 写一个程序，从包含一系列双精度类型的浮点数中获得输入，并将这些数值的平均值输出到屏幕。读取的文件中只包含浮点数，它们之间用空格或者换行符分隔。
3. a. 计算一个数据文件中的所有数据的中值。中值是指这样的—个值，大于该值的数据元素与小于该值的数据元素相同。可以进一步理解中值的概念，假定文件中的数据已经排好序（按照升序排列）。假如文件中的数据元素个数是奇数时，中值是排好序的数据的中间数据；假如数据元素个数是偶数时，中值则是排好序的数据的中间两个值的平均值。你应该先打开文件，计算数据元素的个数，关闭文件并计算文件的中间位置，再打开文件（参考本章前面关于文件的“起始位置”的讨论），计算到你需要的文件输入，然后计算中值。  
b. 对于一个排好序的文件，一个四分位数是如下三个数中的一个：数据中有  $1/4$  的数值小于或等于第一个四分位数， $1/4$  的数值在第一个和第二个四分位数之间（可以等于第二个四分位数）， $1/4$  的数值大于第三个四分位数（可以等于第三个四分位数）。找出 a 部分中使用的数据文件的三个四分位数。注意上述的“ $1/4$ ”并不是恰好就是  $1/4$ ，而只要尽量近似就可以了。

提示：在完成了 a 部分后，你已经完成了本工作的  $1/3$ （已经得到了第二个四分位数），你也已经完成了计算其他两个四分位数的大部分工作。

4. 写一个程序，从包含一些双精度类型的浮点数的文件中读数据，程序将文件中的数据平均值和标准偏差（standard deviation）输出到屏幕。输入文件中只包含 double 类型的数据，数据之间用空格或者换行符分隔开。 $n_1$ 、 $n_2$ 、 $n_3$  等的标准偏差是如下这些数值的平均值的平方根： $(n_1 - a)^2$ ， $(n_2 - a)^2$ ， $(n_3 - a)^2$ ，等等。  
 $a$  是数值  $n_1$ 、 $n_2$ 、 $n_3$  等的平均值。

提示：程序应该首先读取完整的输入文件，计算文件中所有数据的平均值，然后关闭文件，再打开文件计算标准偏差。可以在编程练习 2 的基础上对代码进行修改。

5. 写一个程序，用来输入和显示关于程序编写方面的建议。程序运行时，在屏幕上显示一条建议并要求用户输入不同的建议，然后程序终止。下一个用户在程序运行时，屏幕上显示上一个用户输入的建议。输入的建议被保存在一个文件中，而且文件的内容在每次运行程序后都会发生变化。你可以使用编辑器向文件中输入初始的建议，使第一个运行程序的用户能在屏幕上显示建议。用户输入的建议的长度不受限制（任意行数）。用户通过两次按下回车键来结束建议的输入。程序可以通过判断是否连续两次读入了回车符 ‘\n’ 来决定用户是否输入完毕。

6. 写一个程序，合并两个数据文件中的数据并将数据写到第三个文件中。程序从两个不同的文件中获得输入，并将输出写到第三个文件。每个输入文件都包含一系列 `int` 类型的整数，并且这些数是由小到大排列好的。程序运行后，输出文件中应该包含两个输入文件中的所有数值，并且这些数值仍然是按照由小到大排列好的。应该在程序中定义一个函数，该函数包含由两个文件输入流和一个文件输出流组成的三个形参。

7. 写一个用来生成个人邮寄宣传品的程序。程序的输入来自于输入文件和键盘。输入文件中除了用三个字符“`#N#`”来代替收信人的姓名外，其他是信的正文。程序运行时要求用户输入一个名字，然后生成一封新的信件，在新的信件中用输入的姓名代替输入文件中的“`#N#`”三个字符。在信中“`#N#`”三个字符组成的字符串只出现一次。

提示：你的程序应该直到遇到输入文件中的三个字符“`#N#`”读输入文件才停止，而且要将从输入文件中读到的内容写到输出文件中。当碰到“`#N#`”三个字符时，向屏幕输出提示要求从键盘输入名字。你应该自己想出程序中的其他细节。应该在程序中定义一个函数，函数调用时要求由文件输入流和文件输出流作为参数。假如该练习作为课堂作业的话，由老师给出文件名。

更难的版本：允许字符串“`#N#`”三个字符组成的字符串在文件中任意多次出现。此时，文件将保存在两个字符串变量中。对于该练习而言，假定名字中只有名和姓，而不存在中间名或其他部分。

8. 写一个计算一门课程分数的程序。课程记录保存在一个文件中，该文件作为程序的输入。输入文件的格式如下：每行包含学生的姓，然后依次是一个空格，学生的名，一个空格，之后是 10 次测验的得分情况，全部在一行上记录。10 次测验的得分都是数字并且用空格分隔开。程序的输入来自于该文件并将输出写到一个新文件中。输出文件的数据格式与输入文件基本上相同，只是在每行的最后增加了一个 `double` 类型的浮点数。该浮点数是学生 10 次测验的平均值。要求程序至少使用一个函数，该函数的参数全部或部分为文件流。

9. 修改编程练习 8 的程序，使它能够处理下面的情况。

- 在每行上的测验分数可能包含 10 个或者少于 10 个的测验分数（假如是少于 10 个的测验分数，则说明该学生缺少一个或者多个测验分数）。平均值仍然是测验分数的总和除以 10。对于缺少的测验按照 0 分来计。
- 在输出文件的开始部分加入一行或多行来解释输出的格式。使用输出格式化指令以使输出整齐并且易读。
- 在将正确的输出内容写入到文件后，关闭所有的文件。然后将输出文件的内容拷贝到输入文件中以使输入文件的格式也很整齐。

在本程序中，至少要使用两个参数全部或部分为文件流的函数。

10. 写一个计算文本文件中所有单词平均长度(每个单词包含的平均字符数)的程序。

一个单词是由任意符号组成的字符串,只要该字符串后跟一个空格、一个逗号、一个句点或者行开始符或行结束符时就认为是一个单词。要求程序中定义一个函数,该函数在调用时使用一个文件输入流作为其参数。要求该函数对流 cin 仍然有效,尽管在该程序中函数调用时不会使用 cin 作为其参数。当将该练习作为课堂作业时,由老师给出文件名。

11. 写一个修改 C++ 程序中的错误使用 cin 和 cout 运算符“<<”或者“>>”的程序。在程序中每次出现“cin<<”的地方用正确的“cin>>”来代替。每次出现“cout>>”的地方用正确的“cout<<”来代替。

在早期的程序版本中,可以假定在 cin 和其后出现的“<<”运算符之间有一个空格,同样假定在 cout 和其后出现的“>>”运算符之间有一个空格。在稍难一些的版本中,假定在 cin 和运算符“<<”之间,cout 和运算符“>>”之间可以是任意数目的空格,也可以没有空格;在该版本中,修改后的正确的程序版本在 cin 或 cout 及其后的运算符之间只有一个空格。要被修改的错误程序在一个文件中,修改后的正确的程序版本输出到另一个新文件中。要求程序中定义一个函数,函数的调用参数包含一个输入流和一个输出流(提示:在完成早期程序版本后,即使再做稍难的版本,也会发现这比开始写早期程序版本要容易和迅速。你可以在早期版本程序的基础上进行修改,使它达到稍难版本程序的要求)。

12. 写一个程序,允许用户输入一行文本用来描述一个问题,程序对每个输入的问题进行回答。程序不会真正对问题的内容做出反应,只是简单地读问题行并忽略所读取的内容。程序对于用户输入的问题总是给出如下回答中的一种。

```
Im not sure but I think you will find the answer in Chapter #N.
That's a good question.
If I were you, I would not worry about such things.
That question has puzzled philosophers for centuries.
I don't know. I'm just a machine.
Think about it and the answer will come to you.
I used to know the answer to that question, but I've forgotten it.
The answer can be found in a secret place in the woods.
```

上述答案将被保存在一个文本文件中(每个答案保存一行),程序只是简单地从文件中读取下一个答案并将读出的答案作为问题的回答。程序读完整个文件后,关闭文件再打开文件,再从头开始读答案。

只要程序输出第一个答案时,就要用在 1 ~ 20 (包含 1 和 20) 之间的一个数来代替文本中的“#N”两个符号。为了选择 1 ~ 20 之间的一个数,程序定义一个初始值为 20 的整型变量,并且在每次输出该行时将整型变量的值减 1,使章数从 20 减少到 1。当整型变量的值减少到 0 时,将它的值再改为 20。将 20 这个数值使用 const 修饰符声明为名称为 NUMBER\_OF\_CHAPTERS 的一个全局常量(提示:使用在本章中定义的新Line 函数)。

13. 该练习与编程练习 12 比较类似, 只是该练习的程序将会使用一个更复杂的方法来为输入的问题选择答案。当读入一个问题时, 程序将计算问题中字符的个数并将该数值存储在 `count` 变量中, 然后程序输出 `count%ANSWERS` 数值对应的答案。文件中的第一个答案是答案 0, 依次是 1, 2, 等等。`ANSWERS` 是一个常量声明, 如下所示, 该值等于答案文件中答案的个数。

```
const int ANSWERS = 8;
```

采用这种方法, 你可以很方便地更改答案文件中答案的个数。当答案文件中的答案增加或者减少的时候, 只需将该常量声明的数值改为与答案文件中的答案个数一致即可。假定即使更改了答案文件, 在文件中的第一个答案总是如下。

```
I'm not sure but I think you will find the answer in Chapter #N.
```

此时, 使用 `count%NUMBER_OF_CHAPTERS + 1` 的值来代替上述文本中的“#N”两个字符, `count` 是上面讨论的变量的值, `NUMBER_OF_CHAPTERS` 是一个全局常量, 该常量的值与本书中的章节数目相等。

14. 该程序为一个文本文件的每行内容加上行号。

写一个程序, 读取一个文件中的文本并在每行文本的前面写上每行的行号。输出时行号占三个字符的位置, 而且是右对齐。行号后紧跟一个逗号, 然后是一个空格, 随后是该行的文本。你应该一次读取一个字符, 而且在代码中要忽略每行文本前的空白字符。假定每行的文本足够短, 可以在屏幕上显示一行。否则, 允许默认的打印机或者屏幕输出自动处理超过显示长度的部分, 即截断多余的部分。

完成上述练习后, 有兴趣的读者还可以计算在显示行号区域的空格的数目, 这可通过在处理文件的每行内容之前先计算文件中的行数来完成。要求该版本的程序执行完后, 在文件的最后一个单词后插入一个可以容纳 72 个字符的新行。

15. 该程序用来处理一些文本并针对文本生成一个 KWIX 表 (Key Word In context table, 上下文中的关键字表) 的列表, 然后对每个具体的关键字, 在表中填入关键字、在上下文中出现的行号, 以及关键字出现时的上下文。对于一个给定的关键字, 上下文可能不止一个词。包含关键字的上下文单词的先后顺序与在文本中出现的先后顺序相同。对于填这个表而言, 是由用户决定关键字前的单词数、关键字以及关键字后显示的单词数。

该表的第一列是关键字列, 该列的每行关键字是按照字母排列顺序组织的; 第二列表明关键字出现的行号; 第三列是关键字出现时的上下文内容列。请看下面的例子。请向老师索要相关的文本, 也可以自己找一些测试文本。

提示: 为了得到关键字列表, 应该选择并输入几个段落的文本, 然后略去段落中出现的一些常用词, 如, 动词 “to be”, 名词 I、he、she、her、you、us、them、who、which, 等等。之后, 对关键字列表进行字母排序并去除重复的。该项工作完成得越好, 你所得到的结果就越有用。

例：下面是一段文字以及该段文字的 KWIX 表。

There are at least two complications when reading and writing with random access via an `fstream`: (1) You normally work in bytes using the type `char` or arrays of `char` and need to handle type conversions on your own, and (2) you typically need to position a pointer (indicating where the read or write begins) before each read or write.

KWIX 列表：

关键字	行号	关键字上下文
<code>access</code>	2	with random <i>access</i> via
<code>arrays</code>	3	<i>char</i> or <i>arrays</i> of
<code>bytes</code>	2	work in <i>bytes</i> using
<code>char</code>	3	the type <i>char</i> or
<code>char</code>	3	array of <i>char</i> and
<code>conversions</code>	3	handle type <i>conversions</i> on

实际的表格比上述的示例要长得多。

16. 文本文件 `words.txt`（本书相关的源代码中有这个文件）包含一个按字母顺序排序的英文单词列表。注意这些单词是大小写混用的。

编写一个程序读取这个文件，并找到仅包含一个元音字母（a, e, i, o, u）的最长的单词。输出这个单词（最长的单词可能有好几个，你的程序只需输出其中一个单词即可）。

17. 文本文件 `words.txt` 包含一个按字母顺序排序的英文单词列表。注意这些单词是大小写混用的。

编写一个程序读取这个文件，并找到倒转后是另一个词的最长的单词。比如，`stun` 倒转后变成了单词 `nuts`，但它只有四个字母。找出最长的这样的单词。在编写程序时，你可以使用 `words.txt` 文件包含 45 407 个单词这一信息。

取决于你的计算机的速度及你的实现方案，解本题所需的时间从数分钟到数小时不等。

18. HTML 文件使用放在尖括号中的标记来表示格式化指令。比如，`<B>` 表示粗体，而 `<I>` 表示斜体。假如 Web 浏览器在显示包含 `<` 或 `>` 的 HTML 文档，它会将这些符号误以为是标记。这在 C++ 文件中是很常见的问题，因为这些文件中包含许多 `<` 和 `>`。比如，语句 `#include <iostream>` 会引起浏览器将 `<iostream>` 解释为标记。

为了避免这个问题，HTML 使用特殊的符号来表示 `<` 和 `>`，符号 `<` 用字符串 `&lt;` 来创建，而符号 `>` 用字符串 `&gt;` 来创建。

编写一个程序，读取 C++ 源文件并将所有 < 符号转化为 &lt;，将所有 > 符号转化为 &gt;。同时在文件开始处加上 <PRE> 标记，在文件尾加上 </PRE> 标记。这个标记会保留 HTML 文档中的空格和格式。你的程序应该将 HTML 文件输出到磁盘中。

举个例子，给定如下输入文件：

```
#include <iostream>

int main()
{
    int x=4;
    if (x < 3) x++;
    cout << x << endl;
}
```

程序应当产生具有如下内容的文本文件：

```
<PRE>
#include &lt;iostream&gt;
int main()
{
    int x=4;
    if (x &lt; 3) x++;
    cout &lt;&lt; x &lt;&lt; endl;
}
</PRE>
```

你可以在 Web 浏览器中打开这个输出文件来测试它。显示的内容应当看起来与最初的源文件一样。

19. 编写一个类，跟踪某游戏的五个最高玩家得分。分数应该存储在一个文件中，并包含玩家的姓名（字符串型）和玩家的得分（整型）。得分排行榜的列表最初包含的姓名是 Anonymous，分数是 0。这个类应该支持如下功能：

- 在屏幕上输出最优秀的五位玩家的姓名和得分。得分应该按顺序列出，最高分列在第一位，最低分列在最后。
- 包含一个用来接受新名字和得分的函数。假如该得分高于五高分中的任何一个，就将其添加到文件，并删除文件中的最低得分。否则，五高分列表就保持不变。

有必要的话，加入合适的构造函数和析构函数，以及任何帮助程序功能。编写一个 main 函数，通过模拟一些得分项并输出高分列表来测试这个类。

20. 在本书的源代码中有一个文本文件 words.txt，里面保存了若干按字母顺序排列的未区分大小写单词。

编写程序按行将这些单词读入，找到含有最多连续元音字母的单词。元音字母包括：a, e, i, o, u。

比如单词 `aqueous`，它有四个元音字母。文件中有一个包含五个连续元音字母的单词，找到它。

21. 重新做一遍第9章的编程练习10，不过这次不要再将文本硬编码了，应该从一个名为 `trivia.txt` 的文件中读取问答题。文本文件的格式应该如下：

```
<number of questions, N>
<question 1>
<answer 1>
<dollar amount for 1>
<question 2>
<Answer 2>
<dollar amount for 2>
...
<question N>
<answer N>
<dollar amount for N>
```

比如这里有两个简单的问答题：

```
2
Creator of the C++ programming language?
Bjarne Stroustrup
10.00
The geometric figure most like a lost parrot?
Polygon
20.00
```

22. 第9章的编程练习11曾要求你编写过一个函数用来检查两个字符串是否是字母顺序翻转的。在完成了这个函数的基础上，编写程序将 `words.txt` 中的单词读入到一个 `vector` 中，然后从终端接收输入，将 `vector` 中与输入单词字母顺序翻转的单词输出。
23. 使用逗号分隔的数据或 CSV 文件都是一种保存记录序列的简单方法。逗号经常被用作不同字段的分隔符，这种数据格式经常用来在数据表格或者数据库之间迁移数据。假设有一家商店在卖五种可以简写为 ABCDE 的商品，顾客可以用 1~5 五个数字为商品打分，1 代表不好而 5 代表好。打分情况保存在 CSV 文件中，下面是五种商品打分数据格式的示例：

```
A,B,C,D,E
3,0,5,1,2
1,1,4,2,1
0,0,5,1,3
```

- 第一行是五种商品的缩写，后面每一行代表一个顾客的打分，对应列上的数字是对应商品的分数。创建一个这种格式的文本文件，编写程序计算商品的平均得分。忽略顾客打分的空缺，注意要使用 `vector` 等可以动态调整大小的容器保存数据，编写的程序要能够处理数量变化的商品和顾客打分。

24. 使用 `cin` 直接向一个 `int` 型变量输入数值经常会发生错误，因为很可能输入的值并不是一个 `int` 型的数字。一个简单的解决方法是按照字符串格式输入一个数字，然后验证输入的是否是数字，再将数值转换成 `int` 型变量。编写程序提示用户输入一个整型数字，使用 `getline` 读入用户输入，然后使用 `stringstream` 解析字符串中的数字，假如解析失败则提示用户继续输入。函数应该返回最终输入成功的 `int` 型变量。







递归

13

### 13.1 递归void函数 478

示例：竖直排列的数字 479

跟踪一个递归调用 481

深入理解递归 484

陷阱：无限递归 485

递归调用中的栈 487

陷阱：栈溢出 488

递归与迭代的比较 488

### 13.2 有返回值的递归函数 489

有返回值的递归函数的一般形式 489

示例：另一个幂函数 490

交叉递归 494

### 13.3 按递归方式思考问题 496

递归设计技巧 496

二分查找 497

编码 498

检查递归是否正确 501

效率 502

# 第 13 章 递归

在威廉·詹姆斯的一节宇宙学和太阳系结构的讲座之后，一位小个子的老妇人来找他讨论。

“詹姆斯先生，您有关太阳是太阳系的中心，而地球是环绕它旋转的球体的理论非常令人信服，但是，它是错的。我已经有了一个更好的理论。”

詹姆斯很礼貌地问道：“女士，那是什么呢？”

“我们生存于上的地壳是一只巨龟的壳。”

詹姆斯决定通过让他的对立者正视一些其立场所显示出的不足之处来温和地劝诫她，而非寄希望于运用他所掌握的大量科学论据来驳倒她荒谬的小理论。

他问道：“女士，如果您的理论是正确的，那么巨龟是站在哪里呢？”

“詹姆斯先生，您是个非常聪明的人，这也是个非常好的问题”，小个子老妇人回答道，“但是我确有答案，是这样的：第一只巨龟站在第二只巨龟的背上，那第二只巨龟非常大，就站在第一只巨龟的正下方。”

执着的詹姆斯耐心地询问：“但是第二只巨龟又站在哪儿呢？”

小个子老妇人自鸣得意地说道：“您这样的问题是没用的，詹姆斯先生，总有巨龟站在下面。”

J. R. ROSS, 《语法中的变量约束》

## 概述

在一个函数体内包含对它自己的调用就叫作递归。与大部分编程语言一样，C++ 允许递归地调用函数。只要恰当地使用递归，它将会是一项十分有效的编程技术。本章介绍定义成功递归函数调用的一些基本技术。在本章讲述的内容并不是专门针对 C++ 语言的。如果你对递归已经很熟悉，那么可以略过本章。

本章只用到了第 1 ~ 5 章的内容。13.1 节和 13.2 节并没有用到第 5 章之后的任何内容，因此只要学习了前 4 章的内容，你就可以学习这两节的内容。如果你还没有阅读过第 11 章，那么在遇到命名空间的内容时，参考第 11 章中这部分的内容是很有帮助的。

## 13.1 递归 void 函数

我仍然记得在《一千零一夜》中的某一个夜晚，当谢赫拉沙德（经一个神奇的誊抄失误而命名）开始逐字地讲述《一千零一夜》的故事时，引发了再次重现这一夜的现象，故事中的她也在讲述相同的故事，一直重复，永远循环下去。

乔治·路易斯·博格斯，《小径花园》

当你写了一个函数来完成一项任务时，一个基本的设计原则是将这项任务分为几个子任务，有时可以看到每个子任务都是相同任务的一个小范围的实例。以查找一个给定数值的任务为例，你可能会将该项任务划分为在整个集合中的前半部分查找和后半部分查找两个子任务。在集合的一半空间中查找，相当于原始任务的一个缩略版本。只要分割出来的子任务能看作原来任务的缩略版本。就可以用递归函数来解决该问题。我们用一个简单的例子来说明这项技术。

### 递归

在 C++ 语言中，函数的实现部分可以包含对该函数自身的调用。在这种情况下，该函数就是递归的。

#### 示例：竖直排列的数字

示例 13.1 中的程序演示了一个递归函数 `writeVertical`。递归函数 `writeVertical` 的参数是一个非负的整型变量，该函数的运行结果是将参数数值的每个位上的数字按照一行写一位的格式竖直排列。例如下面的函数调用：

```
writeVertical(1234);
```

将会产生下面的输出：

```
1
2
3
4
```

#### 示例 13.1 一个 void 型递归函数

```
1 // 演示递归函数 writeVertical 的程序。
2 #include <iostream>
3 using std::cout;
4 using std::endl;

5 void writeVertical(int n);
6 // 前提条件：n >= 0。
7 // 运行结果：数字 n 被竖直地输出到屏幕上，
8 // 每行都是一个单独的数字。

9 int main( )
10 {
11     cout << "writeVertical(3):" << endl;
12     writeVertical(3);

13     cout << "writeVertical(12):" << endl;
14     writeVertical(12);

15     cout << "writeVertical(123):" << endl;
```

```

16     writeVertical(123);

17     return 0;
18 }

19 // 使用 iostream。
20 void writeVertical(int n)
21 {
22     if (n < 10)
23     {
24         cout << n << endl;
25     }
26     else //n 必须是两位数以上的值。
27     {
28         writeVertical(n / 10);
29         cout << (n % 10) << endl;
30     }
31 }

```

#### 示例运行结果

```

writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3

```

函数 `writeVertical` 的执行可以分为以下两种情况。

- 简单的情况：如果  $n < 10$ ，那么将数字  $n$  输出到屏幕。  
因为此时数字只有一位，这种情况是最简单的。
- 第二种递归调用的情况是，如果  $n \geq 10$ ，那么程序将分为两个子任务。
  1. 输出除了最后一位的所有其他位上的数字。
  2. 输出最后一位数字。

如果函数的输入参数为 1234，那么第一个子任务将输出：

```

1
2
3

```

第二个子任务将输出数字 4。函数的定义就可以从这种分解子任务的过程中得出。

子任务 1 是原始任务的一个缩略版本，因此我们可以使用递归调用来实现子任务。子任务 2 仅是我们上面列出的简单情况。所以参数为  $n$  的函数 `writeVertical` 的算法逻辑可以用下面的伪代码来表示：

```

if (n < 10)
{
    cout << n << endl;
}

```

```

    }
    else // 数值 n 有两位或者更多位。
    {
        writeVertical (去除数值 n 的最后一位得到的新数值);
        cout << 数值 n 的最后一位数字 << endl;
    }
}

```

递归的子任务。  
↙

只要遵守下面的规则，就可以十分方便地将伪代码转换为完整的 C++ 函数定义：

n/10 所得到的新数值是去除数值 n 的最后一位所得到的数值。  
n%10 是数值 n 的最后一位数字。

例如，1234/10 的结果是 123，而 1234%10 的结果是 4。

完整的函数实现代码如下所示：

```

void writeVertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else // 数值 n 有两位或者更多位。
    {
        writeVertical(n / 10);
        cout << (n % 10) << endl;
    }
}

```

## 跟踪一个递归调用

我们来仔细看一下调用下面的函数时究竟发生了什么（如示例 13.1 所示）。

```
writeVertical(123);
```

当该函数被调用时，计算机对该函数调用的处理就像处理其他的函数调用一样。首先用参数 123 来代替函数的形参 n，然后执行函数体。当用 123 代替 n 后，函数执行流程与下面的代码相同：

```

if (123 < 10)
{
    cout << 123 << endl;
}
else // n 是一个两位或者更多位的数字。
{
    writeVertical(123 / 10); ← 计算过程将在这里挂起，
    cout << (123 % 10) << endl; 直到递归的调用返回。
}

```

既然 123 大于 10，那么函数的 else 部分将被执行。然而函数的 else 部分会继续调用下面这样的函数：

```
writeVertical(n / 10);
```

由于 n 是 123，实际上运行的程序如下所示：

```
writeVertical(123 / 10);
```

上面的函数调用与下面的语句效果是相同的：

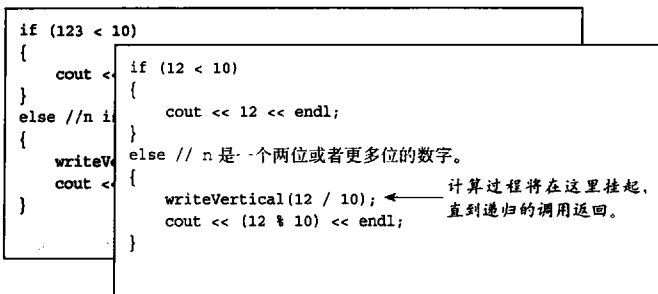
```
writeVertical(12);
```

当程序运行到此处的函数递归调用时，当前的函数执行过程将被挂起，接着执行递归调用的函数。当函数的递归调用结束时，被挂起的执行函数将在被挂起的位置恢复，函数执行过程从恢复的位置开始继续执行。

递归函数调用：

```
writeVertical(12);
```

同其他的函数调用一样。首先用参数 12 来代替函数中的形参 n，然后再执行函数体。用 12 替代参数 n 之后，函数中存在着两个执行过程，一个被挂起，另一个处于运行状态，就如同下面两段代码：



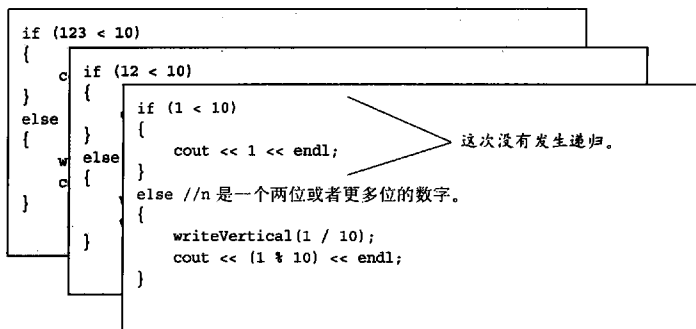
既然 12 大于 10，那么程序将会执行函数体的 else 部分。然而正如刚才所说的一样，else 部分的代码会首先执行一个递归调用。递归调用的参数是 n/10。因此，函数 writeVertical 的第二次执行也会被挂起，函数执行下面的递归调用：

```
writeVertical(12 / 10);
```

实际上就是：

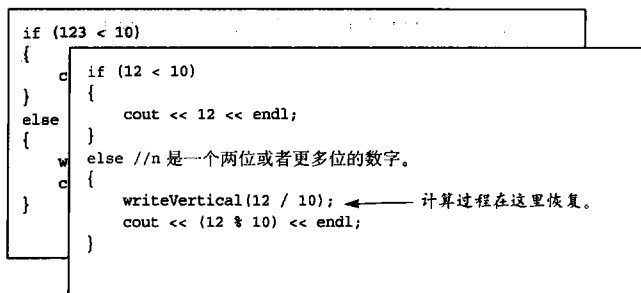
```
writeVertical(1);
```

此时函数中有两个被挂起的执行过程等待着被恢复，计算机开始执行新的函数递归调用，就像以前的递归调用一样。使用实参 1 代替了函数的形参 n 后，函数体开始执行。此时，计算过程如同下面的代码所示：



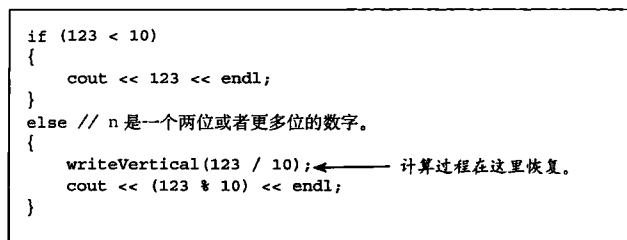
这次执行函数体，与以前执行该函数体时的情况有些不同。既然 1 小于 10，那么在 if-else 语句中的布尔表达式的值为 true，所以程序将会执行 if 部分。if 部分的语句是一个简单的 cout 输出语句，将变量 1 输出到屏幕，所以函数调用 writeVertical(1) 的结果是将 1 写到屏幕上。由于其中不再有函数的递归调用，因此该函数的执行将结束。

当函数调用 writeVertical(1) 结束后，等待被恢复执行的挂起执行过程如下列代码所示：



挂起的执行过程被恢复时，程序执行一个 cout 输出语句输出 12%10 的值 2。现在函数当前的执行过程结束了，但是仍然有一个挂起的执行过程等待被恢复。

当最后一个被挂起的执行过程被恢复后，情况如下列代码所示：





最后一个被挂起的执行过程输出  $123\%10$  的值 3。至此，最开始的函数调用才真正结束。此时，数字 1、2、3 将按照要求的格式在屏幕上每行输出一个数字。

### 深入理解递归

在函数 `writeVertical` 的定义中使用了递归。然而当我们再次回顾递归函数 `writeVertical(123)` 的时候，会发现程序在调用该函数的过程中并没有使用新的未曾介绍过的技术。程序只是把对该函数的调用当作与以前普通的函数调用一样的过程来对待。程序简单地使用实参的值 123 替换函数中的形参 `n`，然后按照函数定义中的代码一步步地执行。当程序执行到递归调用：

```
WriteVertical(123 / 10)
```

时，接下来的过程只是上面的函数调用过程的再一次重复。

递归调用  
的过程

计算机使用下面的方法来跟踪递归调用。当一个函数被调用后，计算机将函数调用的参数数值替换为函数传入的变量的值，然后开始执行函数体代码。如果执行过程中遇到一个递归调用，那么函数的执行过程将在此处挂起，因为函数必须要得到递归调用的结果才能继续向下执行。计算机此时保存了在稍后继续执行所需要的所有信息，然后去执行递归调用子过程。当递归调用完成后，计算机返回到暂停点并继续执行外部计算，直到结束。

递归调用  
的结束

C++ 编程语言没有限制函数的定义中应该如何进行递归调用。但是，为了保证一个递归函数定义是有效的，要求该函数体中必须有不属于递归调用的一部分代码，以使得对递归函数的调用能够自己正常返回。在函数体中可以调用函数本身，而且在递归调用中还可以继续调用函数自身。该递归过程可以是任意次数的重复。不过只有在最后某一次的递归调用中函数执行了不依赖于递归的代码并返回一个值，函数体的递归过程才会正常结束。有效的递归函数体的定义概述如下：

- 在函数完成其任务的过程中，在一种或多种情况下，可以通过一个或多个的函数递归调用来完成其任务的缩略版本。
- 在函数完成其任务的过程中，在一种或者几种情况下将不会再使用递归调用。不再有递归调用的这种情况叫作递归的**基本条件**或**终止条件**。

基本条件  
终止条件

通常情况下程序使用一个 `if-else` 表达式来决定接下来应该执行哪段代码。一个典型的场景是在执行某段程序的时候，一个初始函数的调用中总是包含一个递归调用。该递归调用的执行可能会需要另外一个递归调用。经过一定次数的重复后，每个递归调用都会产生一个另外的递归调用，但是最终的递归调用的结果都应该能理解为一个基本条件或者终止条件的解。递归函数的每次调用，最终都应该有一个终止的条件，否则递归函数的调用将由于无限的循环嵌套而永远不会终止（实际上，无限循环嵌套的递归函数调用通常在把系统资源耗尽时，会发生异常而终止，并不会永远运行下去）。

通常使用下面的方法来保证递归函数调用可以正常结束：在写函数体时，保证函数的每次递归调用中某些数值（正数）减少并将某些“小”数值设定为终止条件。示例 13.1 中的 `writeVertical` 函数就是按照这样的思想来设计的。当函数 `writeVertical` 被调用后，这个条件一直成立，直到函数的参数值小于 10 的时候，

递归调用将停止。当参数小于 10 时, 函数不会再产生新的函数递归调用, 函数的执行过程将一层层地返回到最初始的调用状态, 函数的执行过程将正常终止。

### 定义递归函数的常规形式

有效的递归函数的定义应该具备下面这样的特征:

- 在函数完成自己任务的过程中会有一个或者多个条件判断, 在这些条件判断语句后可以由一个或者多个函数的递归调用来完成初始任务的缩略版本。
- 在函数完成自己任务的过程中, 会有一个或者多个条件判断使得接下来将不会再有函数的递归调用, 这些条件称为递归的基本条件或者终止条件。

### 陷阱: 无限递归

#### 无限递归

在上面讨论的函数 `writeVertical` 中, 函数的递归调用执行的最后一步是对一个不再包含递归的函数的调用 (即到达了终止条件)。如果与上面的情况恰好相反, 每次递归调用都会产生一个新的递归调用, 那么在理论上该函数将会永远执行下去。这种情况叫作无限递归。实际上, 这种无限递归函数的执行将会耗尽系统的所有资源, 然后程序将异常终止。

无限递归的例子有很多。下面的这个函数体就是一个语法正确的 C++ 函数定义, 该函数的目的与 `writeVertical` 的目的相同。

```
void newWriteVertical(int n)
{
    newWriteVertical(n / 10);
    cout << (n % 10) << endl;
}
```

如果我们将上面的函数体的定义放到一个程序中, 编译器会把上面的函数定义转换为机器指令并执行程序。这个函数的定义是有一定的意义的, 其本意是想用函数 `newWriteVertical` 来输出参数, 首先输出除了最后一位的所有位, 然后再输出最后一位。然而, 在实际的函数调用过程中, 该函数将会产生无限的函数递归调用。我们来仔细分析一下上面函数的执行过程。当调用函数 `newWriteVertical(12)`, 而递归调用函数 `newWriteVertical(12/10)` 时 (即调用函数 `newWriteVertical(1)` 时), 执行过程将会被挂起。对于函数 `newWriteVertical(1)` 的递归调用来说, 当递归调用函数 `newWriteVertical(1/10)` 时同样会被挂起。对函数 `newWriteVertical(1/10)` 的调用实际上就是执行函数 `newWriteVertical(0)`。同样, `newWriteVertical(0)` 的调用又会递归调用函数 `newWriteVertical(0/10)`, 该调用仍然是 `newWriteVertical(0)`。这样, 就会反复调用 `newWriteVertical(0)`, 产生无限递归。由于递归函数 `newWriteVertical` 的代码中没有考虑到递归终止的条件, 因此程序将会无限运行下去 (或者直到耗尽系统的资源为止)。■

### 自测练习题

#### 1. 下面程序的输出是什么?

```
#include <iostream>
using std::cout;
void cheers(int n);

int main()
{
    cheers(3);
    return 0;
}

void cheers(int n)
{
    if (n == 1)
    {
        cout << "Hurray\n";
    }
    else
    {
        cout << "Hip ";
        cheers(n - 1);
    }
}
```

2. 实现一个递归函数，该函数的返回类型是 `void`，包含一个输入值为正数的形参，函数的运行结果是向屏幕上输出输入的正数个数的“\*”符号，而且所有的“\*”都在同一行中。
3. 写一个返回值类型为 `void` 的递归函数，该函数的参数是一个正整数。函数将输入的正整数的每一位上的数字逆序输出。也就是说，如果输入的参数为 1234，则该函数向屏幕的输出如下面所示：

4321

4. 写一个返回值类型为 `void` 的递归函数，该函数的输入是一个正整数变量 `n`，运行结果是 1, 2, ..., `n`。
5. 写一个返回值类型为 `void` 的递归函数，该函数的输入是一个正整数变量 `n`，运行结果是 `n`, `n-1`, ..., 3, 2, 1。(提示：可以修改练习 4 的程序来完成该练习，可能只需要修改练习 4 的两行代码就可以实现。)

## 递归调用中的栈

栈

为了实现递归调用（以及其他的功能），大多数计算机系统都使用了叫作“栈”的数据结构。栈是一种特殊类型的存储结构，与一堆纸的存放类似，这种比喻基于一个假设：空白纸的数量总是足够的。当要在栈中记录一些信息的时候，需要将信息记录到某张纸上并把这张纸放到这堆纸的最上面。如果要在栈中放置更多的信息，需要再取出一张干净的纸，在它上面记录这些信息后，再将它放在这堆纸的最上面。使用这种方法，就可以在栈中放置更多的信息。

后进先出

从栈中得到信息也是一个十分简单的过程。首先读取到的一定是这堆纸的最上面的一张纸，读取完后，去掉最上面的一张纸。该方法的一个特点是：只有最上面的这张纸是可以读取的。如果想读取第三张纸，必须要拿掉在它上面的两张纸才可以。由于放入栈中的最后一张纸首先被从栈中取走，因此，栈通常也叫作后进先出的存储结构。

有了栈的概念，计算机可以很容易地实现递归。有函数被调用时，先取出一张新纸，将函数的定义拷贝到这张纸上，用作为实参的变量来替换了函数的形参。然后计算机开始执行函数体，当遇到递归调用时，为了得到递归函数调用的返回值，将在该纸上停止正在进行的计算。但是在执行递归调用之前，计算机将保存足够的信息，以便当递归调用的函数返回后，它可以继续进行被挂起的执行过程，保存的信息被写到一张纸上并放在栈中。一张新纸被取出，专门用于该递归函数的调用过程。计算机再将函数的定义拷贝到刚刚取出的新纸上，并用变量替换函数的形参，继续执行函数体。当再碰到函数的递归调用时，重复保存信息的过程并将纸放置在栈中，再取出新的纸来时我们没有把它叫作“栈”，实际上前面将执行过程放置在其他执行过程之上的图已经解释了栈的具体动作。

在函数的递归调用没有再产生新的递归调用之前，上面的过程将会一直继续。当函数的递归调用没有产生新的递归调用时，计算机将会集中处理位于栈顶的这张纸，它包含了刚才由于递归调用计算而被暂时终止执行的执行过程。因此，在递归调用函数返回后，将会继续执行被挂起的计算。当挂起的执行过程结束后，计算机将从栈中取出这张纸，在该纸下面的那张纸就变为了栈顶元素。计算机然后再集中处理此时被挂起的执行过程，依此类推。当处理完位于栈底的元素后，执行过程正常结束。栈的大小可以随着递归调用的次数以及函数体定义的不同而不同。注意栈中的元素只能以后进先出的方式进行存取，这正是在递归调用跟踪时所需要的。每个等待执行的被挂起的执行过程都在等待位于其上面的执行过程被恢复执行。

活动帧

毫无疑问，在计算机中是没有纸的，上面的过程只是一个比喻。在实际的程序运行过程中，计算机使用的是内存而不是纸。这些内存叫作活动帧，活动帧是按照我们刚才所说的“后进先出”的顺序被使用的（这些活动帧并不保存完整的函数定义体，而是保存了一个函数定义体的引用。但是，一个活动帧包含了足够支持计算机运行的信息，就如同活动帧中包含了函数定义体的一个完整拷贝一样）。

**栈**

栈是一种后进先出的存储结构。栈中第一个被访问或者被删除的元素总是最后一个进入栈的元素。栈可以帮助计算机实现递归调用（或者其他功能）。

**陷阱：栈溢出****栈溢出**

栈的大小是有限制的。如果在函数的递归调用中有一条很长的序列的话，例如函数递归调用它自己，该调用又产生了另一个调用，在另一个调用中又调用了函数本身，等等，依此类推，那么在这个调用序列中的所有调用都会向栈中存入一个活动帧。当调用序列太长的时候，栈可能会超出其大小的极限，这就是所谓的“栈溢出”错误。如果在屏幕上发现了“栈溢出”这样的错误，很有可能是一些函数在递归调用时产生了太长的递归调用序列。无限递归就是栈溢出的一种十分常见的情况。如果一个函数被无限递归调用了，最后会使栈超出栈的空间大小限制，从而导致“栈溢出”错误。■

**递归与迭代的比较****迭代版本**

在程序设计中，递归并不一定是必要的。实际上在有些编程语言中是不允许使用递归的。任何使用递归的程序都可以被转换为不使用递归的程序。例如，示例 13.2 就是示例 13.1 中递归函数的非递归版本。在递归函数的非递归版本中，都使用了一些循环来代替递归。由于这个原因，非递归版本有时也叫作**迭代版本**。如果将示例 13.1 中的 writeVertical 函数用示例 13.2 中的函数体来代替，其实运行结果会是一样的。在这种情况下，我们可以看到，函数的递归版本通常比迭代版本要简洁得多。

**示例 13.2 示例 13.1 中函数的迭代版本**

```

1  // 使用 iostream。
2  void writeVertical(int n)
3  {
4      int nsTens = 1;
5      int leftEndPiece = n;
6      while (leftEndPiece > 9)
7      {
8          leftEndPiece = leftEndPiece / 10;
9          nsTens = nsTens * 10;
10     }
11     //nsTens 是与整数 n 有相同位数的 10 的整数次幂。
12     // 如果 n 是 2345，那么 nsTens 是 1000。

13     for (int powerOf10 = nsTens;
14          powerOf10 > 0; powerOf10 = powerOf10 / 10)
15     {
16         cout << (n / powerOf10) << endl;
17         n = n % powerOf10;
18     }
19 }
```

## 尾递归

大部分编译器会自动将简单的递归函数转换成等价的迭代版本。使用了尾递归的函数在递归调用发生之后不会再进行进一步的计算，而是马上返回。13.2 节会介绍带返回值的递归函数，如果一次递归并没有改变函数返回的值，那它就是尾递归。在这种情况下，递归过程可以很容易地转换成等价的迭代过程。查看你的编译器文档和优化标志，确定这种转换是可用的。

## 效率

一个不是尾递归的递归过程通常要比实现相同功能的迭代过程运行得慢，同时也会占用更多的系统资源。计算机为了实现递归要做大量的工作来管理栈。不过既然计算机系统可以代替程序员自动管理栈，那么使用递归可以简化程序员的工作，而且有时使用递归将会使代码更容易理解。

## 自测练习题

6. 如果程序在运行过程中出现了“栈溢出”错误信息，引起这个错误的原因可能是什么？
7. 为自测练习题 1 中定义的 `cheers` 函数写一个迭代版本。
8. 为自测练习题 2 定义的函数写一个迭代版本。
9. 为自测练习题 3 定义的函数写一个迭代版本。
10. 跟踪在自测练习题 4 中的递归过程。
11. 跟踪在自测练习题 5 中的递归过程。

## 13.2 有返回值的递归函数

“迭代者为人，递归者为神。”

佚名

### 有返回值的递归函数的一般形式

到目前为止，我们所见到的递归函数都是没有返回值（返回值为 `void` 类型）的函数，但是递归调用同样适用于有返回值的函数。递归函数可以返回任意类型的值。有返回值的递归函数的设计与无返回值的递归函数的设计基本上一样。一个成功的有返回值的递归函数体的定义一般如下面所示。

- 如果在函数完成其任务的过程中，可能存在多个任务分支，那么可以通过一个或多个函数递归调用来完成其任务的缩略版本。与没有返回值的递归函数的情况类似，递归调用的参数应该“越来越小”。
- 在函数完成其任务的过程中，存在不需要使用递归调用就可以直接将返回值返回的一种或多种情况。没有递归调用而直接返回值的这些情况叫作递归的基本条件或者终止条件（与没有返回值的递归函数的情况相同）。

这项技术将在下一个编程实例中举例说明。

**示例：另一个幂函数**

在第3章中介绍了用于计算幂的预定义函数 `pow`。例如，`pow(2.0, 3.0)` 返回  $2.0^{3.0}$  的绝对值，因此下面的代码将变量 `result` 的值设为 8.0：

```
double result = pow(2.0, 3.0);
```

函数 `pow` 需要两个双精度浮点类型的变量，并且返回一个 `double` 类型的值。示例 13.3 中包含了与该函数类似的一个函数的递归定义，但是示例 13.3 中的函数适用于 `int` 类型的整型变量而不是 `double` 类型的变量。示例 13.3 中定义的函数的名字是 `power`。例如，下面的代码会将 `result2` 的值设为 8，因为  $2^3 = 8$ ：

```
int result2 = power(2, 3);
```

我们定义函数 `power` 的主要原因是想得到一个递归函数的简单例子，但是函数 `power` 也有优于函数 `pow` 的方面。函数 `pow` 返回一个 `double` 类型的双精度浮点类型的值，该值只是一个近似值。而函数 `power` 返回 `int` 类型的整数，该值是一个精确值。在某些情况下，可能需要使用函数 `power` 来提供额外的精度。

函数 `power` 的定义基于下面的公式：

$$x^n = x^{n-1} * x$$

将上面的公式用 C++ 语言来描述就是，函数 `power(x, n)` 的返回值应该与下面表达式的返回值相等：

```
power(x, n - 1) * x
```

在示例 13.3 中定义的 `power` 函数体对于 `power` 的返回值是：

```
(x, n), 其中 n > 0
```

$n=0$  时的情况是递归的终止情况。如果  $n=0$ ，那么 `power(x, n)` 返回 1（因为  $x^0=1$ ）。

下面我们来看看当函数调用实际的数值时会发生什么情况。首先考虑下面的简单表达式：

```
power(2, 0)
```

当调用该函数时，`x` 的值被设为 2，`n` 的值被设为 0，然后将会继续执行函数体。由于 `n` 的值合法，那么将会执行 `if-else` 表达式判断。由于 `n` 的值不大于 0，将执行 `else` 部分中的 `return` 语句，因此函数返回 1。所以，下面这行代码将 `result3` 的值设为 1。

```
int result3 = power(2, 0);
```

**示例 13.3 递归函数 `power`**

```
1 // 描述递归函数 power 调用的程序。
2 #include <iostream>
3 #include <cstdlib>
4 using std::cout;
5 using std::endl;
```

```

6  int power(int x, int n);
7  // 前提条件: n >= 0。
8  // 返回 x 的 n 次幂。

9  int main( )
10 {
11     for (int n = 0; n < 4; n++)
12         cout << "3 to the power " << n
13             << " is " << power(3, n) << endl;

14     return 0;
15 }

16 // 使用 iostream 和 cstdlib。
17 int power(int x, int n)
18 {
19     if (n < 0)
20     {
21         cout << "Illegal argument to power.\n";
22         exit(1);
23     }
24     if (n > 0)
25         return ( power(x, n - 1) * x );
26     else // n == 0
27         return (1);
28 }

```

### 示例运行结果

```

3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27

```

下面我们再看一个包含递归调用的例子。考虑表达式：

```
power(2, 1)
```

当函数调用后，把  $x$  的值设为 2， $n$  的值设为 1 后，将会执行函数体。由于  $n$  的值大于 0，函数将会返回下面的表达式：

```
return ( power(x, n - 1) * x );
```

此时，上面的表达式实际上是：

```
return ( power(2, 0) * 2 );
```

此时， $\text{power}(2, 1)$  的执行过程将被挂起，该挂起计算将被放置在栈中，然后计算机开始一个新的函数调用来计算  $\text{power}(2, 0)$  的值。正如上面所说的， $\text{power}(2, 0)$  的返回值为 1。在计算出  $\text{power}(2, 0)$  的值后，计算机将使用  $\text{power}(2, 0)$  的值 1 来代替它并恢复被挂起的执行过程。恢复的执行过程继续计算函数  $\text{power}(2, 1)$  应该返回的最终值：



`power(2, 0) * 2` 实际上是  $1 * 2$ , 结果为 2。

因此, 函数 `power(2, 1)` 的最后返回值为 2。所以, 下面的代码将变量 `result4` 的值设为 2。

```
int result4 = power(2, 1);
```

该函数的第二个参数的值越大, 将会产生越长的递归调用串。例如, 考虑下面的表达式:

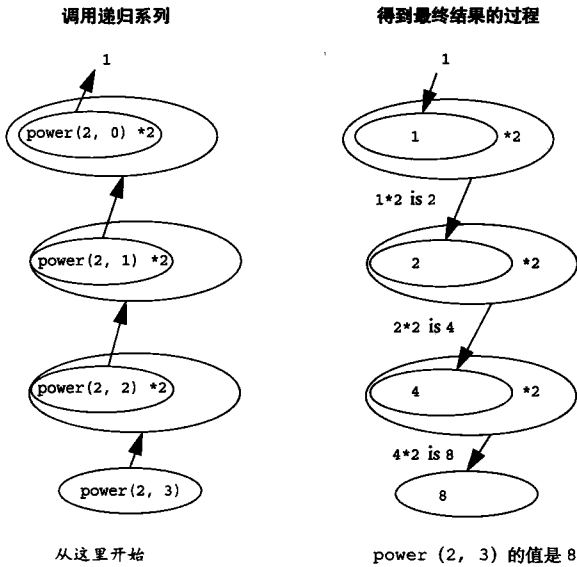
```
cout << power(2, 3);
```

函数 `power(2, 3)` 的值将会按照下面的过程计算:

```
power(2, 3) 的值是 (2, 2)*2
power(2, 2) 的值是 (2, 1)*2
power(2, 1) 的值是 (2, 0)*2
power(2, 0) 的值是 1 (递归终止的情况)
```

当程序运行到终止情况 `power(2, 0)` 时, 有三个被挂起的执行过程。在计算完递归的终止情况的函数返回后, 最后被挂起的执行过程 `power(2, 1)` 将恢复运行。计算机执行完每个被挂起的执行过程后, 都使用函数的返回值来代替上次调用时被挂起的函数中相应的表达式, 直到到达函数的原始调用 `power(2, 3)` 时终止。整个执行过程的详细过程见示例 13.4 的描述。

示例 13.4 递归函数 `power(2, 3)` 的详细过程



## 自测练习

12. 下面程序的输出是什么?

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

int mystery(int n);
// 前提条件: n >= 1。

int main( )
{
    cout << mystery(3) << endl;
    return 0;
}

int mystery(int n)
{
    if (n <= 1)
        return 1;
    else
        return ( mystery(n - 1) + n );
}
```

13. 下面程序的输出是什么? `rose` 函数实现的是什么数学功能?

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

int rose(int n);
// 前提条件: n >= 0。

int main( )
{
    cout << rose(4) << endl;
    return 0;
}

int rose(int n)
{
    if (n <= 0)
        return 1;
    else
        return ( rose(n - 1) * n );
}
```

14. 重新定义函数 `power` 的实现以使它也能对负指数进行处理。为了实现该功能,你也必须要将函数的返回类型改为双精度浮点类型。重新定义的 `power` 函数的声明及其注释如下面所示:

```
double power(int x, int n);
// 前提条件: 如果 n < 0, 则 x 不为 0。
// 返回 x 的 n 次幂。
```

提示： $x^n$  的值是  $1/(x')$ 。

## 交叉递归

迄今为止出现的例子中，递归调用的函数都是直接调用自己，其实，通过调用其他函数间接地调用自身来实现递归也是可能的。当两个或者更多的函数相互调用对方，这种方式叫作交叉递归。

### 交叉递归

这里有一个简单的例子，假设我们有一串连续的 0 和连续的 1，我们想检验这个串中 1 的个数是否是偶数。程序将从左到右扫描这个串，当 1 的计数是偶数时，调用函数 `evenNumberOfOnes`；当 1 的个数是奇数时，调用函数 `oddNumberOfOnes`。从调用 `evenNumberOfOnes` 开始。如果串是一个空串，那么 1 的个数是 0，是个偶数，函数应该返回 `true`。如果串是以 1 开头的，我们要去掉这个 1，然后再调用 `oddNumberOfOnes`，因为一个单独的 1 会让结果固定为奇数。如果串是以 0 开头的，我们要去掉这个 0，然后再调用 `evenNumberOfOnes`，因为 0 不会影响 1 计数的奇偶。

函数 `oddNumberOfOnes` 的逻辑很简单，如果串是空的，函数返回 `false`，表示 1 的个数是偶数。如果串是以一个 1 开头的，那么删除这个 1，然后调用 `evenNumberOfOnes`，因为一个单独的 1 会让结果固定为奇数。如果串是以 0 开头的，我们要去掉这个 0，然后再次调用 `oddNumberOfOnes`，因为 0 不会影响 1 计数的奇偶。

举个例子，对于输入的串“10011”，处理如下：

```
evenNumberOfOnes("10011")
  oddNumberOfOnes("0011")
    oddNumberOfOnes("011")
      oddNumberOfOnes("11")
        evenNumberOfOnes("1")
          oddNumberOfOnes("")
            return false
```

每次扫描到一个 1 的时候，我们在 `evenNumberOfOnes` 和 `oddNumberOfOnes` 之间来回切换。在最后一次调用到了 `oddNumberOfOnes` 之前，我们能看到有一个 `evenNumberOfOnes` 被调用。

示例 13.5 演示了一个交叉递归的例子。函数 `substr` 被用来“移除”字符串中的第一个字母。在指定了开头的 1 的位置并且不给出子字符串的长度的情况下，`substr` 函数将会返回位置 1 到末尾的所有内容，跳过 0 位置的字母。其实这个程序用迭代来实现会更加容易，但是使用交叉递归更容易体现出我们要展示的理念。

### 示例 13.5 使用交叉递归检测一个字符串中是否包含偶数个 1

```
1 // 使用库 iostream 和 string。
2 // 如果递归调用结束于一个空串，
3 // 那么 1 的个数就是偶数。
4 bool evenNumberOfOnes(string s)
```

```

5  {
6    if (s.length() == 0)
7        return true ; // 是偶数。
8    else if (s[0]=='1')
9        return oddNumberOfOnes(s.substr(1));
10   else
11       return evenNumberOfOnes(s.substr(1));
12 }

13 // 如果递归调用结束于一个空串,
14 // 那么1的个数就是奇数。
15 bool oddNumberOfOnes(string s)
16 {
17     if (s.length() == 0)
18         return false; // 不是偶数。
19     else if (s[0]=='1')
20         return evenNumberOfOnes(s.substr(1));
21     else
22         return oddNumberOfOnes(s.substr(1));
23 }

24 int main ()
25 {
26     string s = "10011";
27
28     if (evenNumberOfOnes(s))
29         cout << "Even number of ones." << endl;
30     else
31         cout << "Odd number of ones." << endl;
32     return 0;
33 }

```

### 示例运行结果

```
Odd number of ones.
```

---

### 自测练习题

15. 应该如何跟踪示例 13.5 中的递归程序？
16. 如果输入的数字是偶数，函数 even 返回 true；如果数字是奇数，则返回 false，但是下面的代码并不完整，请补充一个函数 odd，使下面的代码可以正常工作。

```

bool even(int num)
{
    if (num == 0)
        return true;
    else
        return odd(num - 1);
}

```

### 13.3 按递归方式思考问题

“世上有两种人：一种是把世上的人分成两种人的人，另一种是不分的人。”

佚名

#### 递归设计技巧

当定义和使用递归函数时，我们不需要对栈和挂起的执行过程细节十分了解。递归的好处是程序员可以忽略程序运行的细节而让计算机来管理这些细节。这里仍以示例 13.3 中的函数 `power` 为例来说明。函数 `power` 的定义是基于下面的想法考虑的。

```
Power (x, n)
```

返回

```
power(x, n - 1) * x
```

由于  $x^n$  与  $x^{n-1} * x$  是相等的，在函数的执行过程总是会到达一个终止情况并正确地计算终止情况的前提下，函数的返回值是正确的。因此，在确保递归定义的部分是正确的前提下，还要确保递归调用的串总是会到达一个终止情况并且终止情况的计算总是会返回正确的值。

在设计递归函数时，我们不需要对某个函数的具体执行过程进行跟踪，来得到整个递归调用过程。如果函数有返回值，我们所要做的就是检查函数是否满足以下三个属性。

有返回值的  
函数的标准

1. 数中不存在无限递归调用的情况。虽然程序中存在一个递归调用，调用了另一个递归调用，另一个递归调用又产生了一个新的递归调用等，但是每个递归调用的串最后都会达到一个终止情况。
2. 每个终止情况的返回值都是正确的。
3. 对于所有的包含递归调用的情况：如果所有的递归调用都返回正确的值，那么函数最后返回的值是正确的。

例如，以示例 13.3 中的函数 `power` 为例进行说明。

1. 函数中不存在无限递归调用：函数 `power(x, n)` 的第二个参数在每次调用中都减 1，因此任何的递归调用串最后都将到达 `power(x, 0)` 的情况，该情况就是 `power(x, n)` 的终止情况。因此，函数中不存在无限递归调用。
2. 每个终止情况的返回值都是正确的：函数唯一的终止情况是 `power(x, 0)`。函数 `power(x, 0)` 总是返回 1， $x^0$  的正确的值是 1。因此，终止情况返回正确的值。
3. 对于所有包含递归调用的情况，如果所有的递归调用都返回正确的值，那么函数最后返回的值是正确的。只有当  $n > 1$  时才包含递归调用。当  $n > 1$  时，`power(x, n)` 返回 `power(x, n - 1) * x`

为了判断返回值是否正确，注意到如果 `power(x, n - 1)` 返回值正确的话，应该返回  $x^{n-1}$ ，因此 `power(x, n)` 返回  $x^{n-1} * x$ ，该值就是  $x^n$ ，这就是函数 `power(x,`

n) 的正确值。

上面就是为了保证函数 power 的定义正确所需要确认的三个条件（上面的技术叫作数学导论（mathematical induction），你可能在某堂数学课中听到过这个概念。但是，使用本节中的技术不需要你十分熟悉数学导论的概念）。

本节给出了检查有返回值的递归函数正确性的三个评价标准。这三个标准基本上也适用于没有返回值（返回值为 void 的类型）的递归函数。如果返回值为 void 类型的递归函数定义满足下面三个标准，就可以判定该递归函数是正确的。

### 无返回值的 函数的标准

1. 函数中没有无限递归的情况。
2. 每个终止情况都正确执行。
3. 对于包含递归的每个情况：如果所有的递归调用都正确地执行，那么整个函数就正确地执行。

## 二分查找

在本小节中介绍用于查找数组中是否包含某特定值的递归函数的设计过程。例如，数组中可能存储的是无效的信用卡号。商店的职员需要对该数组列表进行查询以决定某个顾客的信用卡是有效还是无效的。

数组 a 的序号是 0 ~ finalIndex。为了简化对数组的查找操作，假定数组已经排好序。即，有如下关系：

$$a[0] \leq a[1] \leq a[2] \leq \dots \leq a[\text{finalIndex}]$$

当对数组进行查找操作时，我们可能想知道两件事情：数组中是否包含要查找的值，如果包含的话，该值存储在数组的哪个单元中。例如，假设我们正在查找一个信用卡号，那么数组的索引序号可能作为一个记录数。在另一个数组中，与之相同的索引序号可能存储的是一个电话号码或者用于报告可疑信用卡号的其他信息。因此，如果查到数组中包含某个顾客的信用卡号后，我们希望函数能告诉我们该值在数组中的存储位置。

现在我们来设计解决该任务的算法。这将帮助我们以十分详尽的方式可视化问题。假定数字的列表十分长并将它们记录在一本书中。这实际上就是在没有计算机的情况下存储无效信用卡号的方法。如果商店的职员正对某一个信用卡号进行处理，就必须检查该信用卡号是否在列表中。此时你怎样处理呢？打开书的中间页并查看该信用卡号是否在该页上。如果不在并且该卡号小于中间页上的信用卡号时，那么就在书的前部分查找。如果信用卡号大于中间页上的信用卡号时，那么就在书的后半部分查找。基于这种思想，设计出了该算法的第一个版本。

### 算法—— 第一个版本

```
found = false ; // 表示到目前为止还没有查到。
mid = approximate midpoint between 0 and finalIndex;
if (key == a[mid])
{
    found = true ;
    location = mid;
}
else if (key < a[mid])
    search a[0] through a[mid - 1];
else if (key > a[mid])
```

```
search a[mid + 1] through a[finalIndex];
```

由于对更短列表的查找是本算法任务的一个缩略版本，那么该算法自然可以用递归来实现。更小的列表可以通过算法自身的递归调用来完成查找。

要将上面的描写算法的伪代码写成 C++ 的代码，还需要确认在递归调用中数组的起点和数组的终点。上面伪代码中的两个递归调用如下：

```
search a[0] through a[mid - 1];
```

以及

```
search a[mid + 1] through a[finalIndex];
```

为了实现这些递归调用，每次至少需要两个参数：一个递归调用指定了要查询数组的一段内容。其中的一种情况是搜索数组中的  $0 \sim \text{mid} - 1$  索引号之间的元素；另一种情况是搜索数组中  $\text{mid} + 1 \sim \text{finalIndex}$  之间的元素。应该为每次搜索过程指定搜索数组的起始和终止索引号，我们分别称之为 *first* 和 *last*。使用上面的参数来代替 0 和 *finalIndex* 来指定查询过程中的低索引号和高索引号，我们可以更精确地使用下面的方式来描述算法，给出的伪代码如下：

算法——  
初步改进

```
To search a[first] through a[last] do the following:
found = false; // 表示到目前为止还没有找到。
mid = approximate midpoint between first and last;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    search a[first] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[last];
```

为了实现对整个数组的查询，算法会将 *first* 设为 0，*last* 设为 *finalIndex* 来执行。在算法的递归调用过程中，会为 *first* 和 *last* 设置其他的值。例如，第一次的递归调用会将 *first* 设为 0，*last* 设为 *mid - 1* 来执行。

终止条件

正如任何递归调用算法一样，我们必须确保所设计的算法一定会结束而不会产生无限递归的情况。如果查询后在列表中找到了数组，那么就不会再产生新的递归调用，查询过程结束，但是如果在列表中不存在要找的数值时，我们必须也能够判断出这种情况来。在每次递归调用的过程中，*first* 值增加而 *last* 的值减少。如果每次递归调用的查询中 *first* 的值大于 *last* 的值时，我们说此时要查找的子数组中已经没有可供继续查找的对象，而且要查找的数组也不在该数组中。当我们把这种判断添加到算法的伪代码中时，我们就得到了算法的完整版本，如示例 13.6 所示。

算法——  
最终版本

## 编码

下面我们来把算法的上述伪代码写成 C++ 代码。写好的 C++ 代码见示例 13.7。函数 *search* 是在示例 13.6 中给出的递归算法的实现，示例 13.8 则用图形说明了该函数对一个实现数组进行查找的过程。

注意函数 `search` 解决了一个比我们的原始任务更一般的问题。我们的目的是设计一个可以查找整个组的函数，然而 `search` 函数可以通过指定数组的上界 `first` 和下界 `last` 来完成对数组任意一个部分的查找。这在设计递归函数时是十分常见的。通常，使算法能够使用递归调用实现，从而能够解决一个更一般意义上的问题。在本文的情况下，只要将 `first` 和 `last` 的值分别设为 0 和 `finalIndex`。然而在递归调用的过程中，函数将会把 `first` 设为与 0、`finalIndex` 不同的其他值。

### 示例 13.6 二分查找算法的伪代码

---

```
int a[ Some_Size_Value ];

    查找数组中 a[first] ~ a[last] 元素的算法

// 前提条件：
//a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]

    定位查找值在数组中的索引号：

if (first > last) // 递归的终止情况
    found = false ;
else
{
    mid = 计算 first 和 last 之间的中点数值；
    if (key == a[mid]) // 递归终止的情况
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] // 需要递归的情况
        在 a[first] 和 a[mid - 1] 之间查找；
    else if key > a[mid] // 需要递归的情况
        在 a[mid + 1] 和 a[last] 之间查找；
}
```

---

### 示例 13.7 二分查找的递归函数

---

```
1 // 演示二分查找的程序。
2 #include <iostream>
3 using std::cin;
4 using std::cout;
5 using std::endl;
6 const int ARRAY_SIZE = 10;

7 void search(const int a[ ], int first, int last,
8             int key,bool& found,int& location);
9 // 前提条件：从 a[first] 到 a[last] 的数组元素是按照升序排列的。
10 // 运行结果：如果 key 不是从 a[first] 到 a[last] 的任意一个值，
11 // 那么 found == false, 否则 a[location] == key 且 found == true.
12 int main( )
13 {
```



```

14     int a[ARRAY_SIZE];
15     const int finalIndex = ARRAY_SIZE - 1;
           这里应该是一段将数组排序的代码，不过其具体的
           实现方法与本例无关。
16     int key, location;
17     bool found;
18     cout << "Enter number to be located: ";
19     cin >> key;
20     search(a, 0, finalIndex, key, found, location);

21     if (found)
22         cout << key << " is in index location "
23             << location << endl;
24     else
25         cout << key << " is not in the array." << endl;

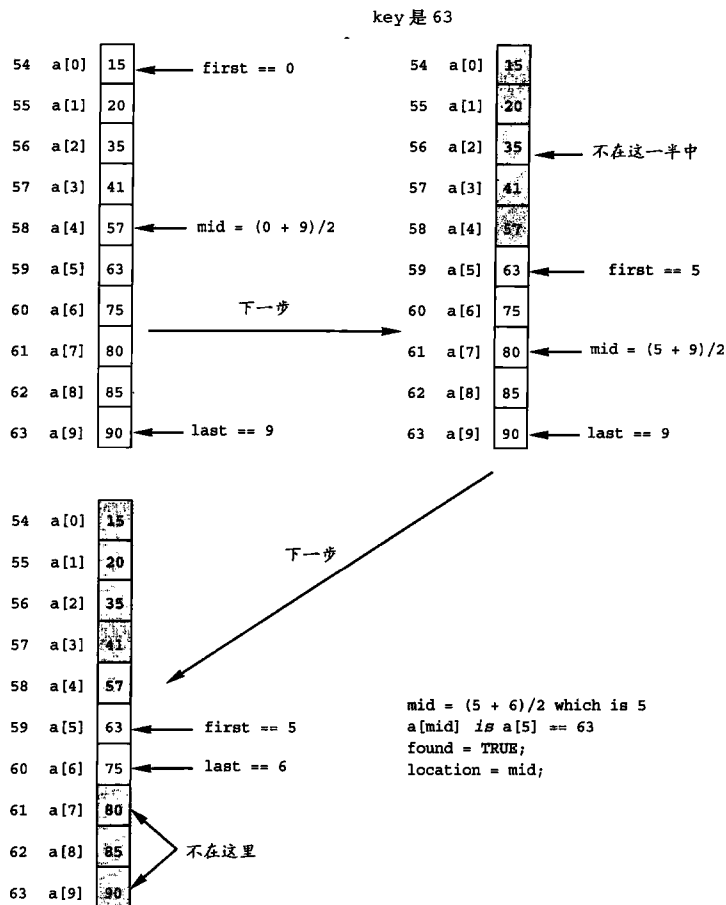
26     return 0;
27 }
28 void search(const int a[ ], int first, int last,
29             int key, bool& found, int& location)
30 {
31     int mid;
32     if (first > last)
33     {
34         found = false ;
35     }
36     else
37     {
38         mid = (first + last)/2;

39         if (key == a[mid])
40         {
41             found = true;
42             location = mid;
43         }
44         else if (key > a[mid])
45         {
46             search(a, first, mid - 1, key, found, location);
47         }
48         else if (key < a[mid])
49         {
50             search(a, mid + 1, last, key, found, location);
51         }
52     }
53 }

```

---

### 示例 13.8 search 函数的执行流程



### 检查递归是否正确

在“递归设计技巧”一节中给出了判断返回值为 void 类型的递归函数的正确性的三个标准。下面我们就用这三个标准来检验一下示例 13.7 中给出的 search 函数的正确性。

1. 函数中没有无限递归的情况：在每次递归调用中，first 的值增加或者 last 的值减小。如果递归调用串在一些其他情况下是不结束时，那么最后函数调用时的参数将会是 first 的值大于 last 的值，这种情况就是递归的一个终止情况。

2. 每个终止情况都正确执行：在函数中有两个终止情况，当  $\text{first} > \text{last}$  和  $\text{key} == \text{a}[\text{mid}]$  时。让我们来分别考虑两个终止情况。

如果  $\text{first} > \text{last}$ ，那么在元素  $\text{a}[\text{first}]$  和  $\text{a}[\text{last}]$  之间就没有数组元素，因此在数组的这段中就不会存在查找的值（此时在查找的数组段中就没有元素）。因此，如果  $\text{first} > \text{last}$ ，函数 `search` 将 `found` 的值设为了 `false`，执行正确。

如果  $\text{key} == \text{a}[\text{mid}]$ ，算法将 `found` 的值设为 `true`，并且将 `location` 的值设为 `mid`。因此，两种终止情况都正确执行。

3. 对于包含递归的每个情况：如果所有的递归调用都正确地执行，那么整个函数就正确执行。存在递归调用的两种情况，当  $\text{key} < \text{a}[\text{mid}]$  和  $\text{key} > \text{a}[\text{mid}]$  时。我们来分别检查这两种递归调用的情况。

首先假定  $\text{key} < \text{a}[\text{mid}]$ 。在这种情况下，由于数组是排好序的，如果要查找的值（假定要查找的值为 `key`）在数组中，那么 `key` 一定是元素  $\text{a}[\text{first}] \sim \text{a}[\text{mid} - 1]$  中的某个值，所以只需在这些元素中查找函数。该项工作在函数中是由语句：

```
search(a, first, mid - 1, key, found, location);
```

来完成的。因此，如果递归调用正确，那么整个函数的执行是正确的。

下面考察  $\text{key} > \text{a}[\text{mid}]$  的情况。在这种情况下，由于数组是已经排好序的，如果要查找的数值 `key` 在数组中，那么 `key` 一定是元素  $\text{a}[\text{mid} + 1] \sim \text{a}[\text{last}]$  之间的某个数值，所以只需在这些元素中查找函数。该项工作在函数中是由语句：

```
search(a, mid + 1, last, key, found, location);
```

来完成的。因此，如果递归调用正确，那么整个函数的执行是正确的。综合以上两种情况，函数总是正确地执行（假定递归调用正确执行）。

通过上面的讨论，函数 `search` 通过了以前讨论的三项正确性检查，因此该函数是一个好的递归函数定义。

## 效率

与简单地数组中的元素按照顺序一个个与要查找的值比较的算法比，二分查找算法是相当快的。在二分查找中，从一开始就大概去除了数组的一半元素，然后又去除了  $1/4$ ，然后是  $1/8$ ，等等。这大大加快了算法的执行速度。在一个具有 100 个元素的数组中，使用二分查找算法与要查找的值 `key` 的比较不超过七次即可完成。而一个简单的序列查询最多时将会进行 100 次的数值比较，平均也要进行 50 次的数组比较。更进一步地说，数组越大，那么使用二分查找所节省的时间就越多。在一个具有 1000 个元素的数组中，使用二分查找进行数组比较的次数不超过 10 次，而使用简单序列查询进行数组比较的次数平均大概要进行 500 次。

示例 13.9 给出了函数 `search` 的迭代版本的实现。在相同的计算机系统中，迭代版本的程序运行的效率要比递归实现的程序效率高，迭代版本的程序实现是从递归版

本的程序实现中导出来的。在迭代版本中的局部变量 `first` 和 `last` 与递归版本中的变量 `first` 和 `last` 相对应，其代表的含义也相同。使用这样的示例是为了演示涉及递归的程序，尽管将来你可能希望将它转换为一个迭代版本的算法程序。

### 示例 13.9 二分查找函数的迭代实现版本

#### 函数声明

```
void search(const int a[ ], int lowEnd, int highEnd,
            int key, bool& found, int& location);
// 前提条件：数组元素 a[lowEnd] ~ a[highEnd] 按照升序排列。
// 运行结果：如果要查找的值 key 不在数组元素 a[lowEnd]
// ~ a[highEnd] 之间，那么 found == false，否则 a[location] == key
// 且 found == true。
```

#### 函数实现

```
void search(const int a[ ], int lowEnd, int highEnd,
            int key, bool& found, int& location)
{
    int first = lowEnd;
    int last = highEnd;
    int mid;

    found = false ; // 到目前为止还没有查到。
    while ((first <= last) && !found)
    {
        mid = (first + last)/2;
        if (key == a[mid])
        {
            found = true ;
            location = mid;
        }
        else if (key < a[mid])
        {
            last = mid - 1;
        }
        else if (key > a[mid])
        {
            first = mid + 1;
        }
    }
}
```

### 自测练习

17. 使用递归实现下面的函数：

```
int squares(int n);
// 前提条件：n >= 1。
// 函数返回 1 ~ n 的平方和。
```

例如，`squares(3)` 返回 14，因为  $1^2 + 2^2 + 3^2 = 14$ 。

## 本章小结

- 如果一个问题可以被划分为相同问题的更小实例，那么这个问题就可以很容易地找到递归的方法并实现。
- 一个函数定义中的递归算法通常包含两种类型的判断条件：一种条件下函数将会产生一次或者多次递归，另一种条件下递归则会终止。
- 当两个或者多个函数彼此调用的时候，会出现交叉递归的现象。
- 写一个递归函数的定义时，总是要检查函数以确保函数不会产生无限递归调用。
- 设计一个递归函数时，使用在“递归设计技巧”小节中的三个标准来检查递归函数的正确性。
- 设计一个用于解决某项任务的递归函数时，解决比给定的任务更具一般性的问题是很必要的。这可能是为了进行合适的递归调用的需要，因为分解的更小问题可能与给定的原始任务不完全相同。例如，在二分查找的问题中，其任务是查找整个数组，但是递归算法的解决方案却是查找数组中的某一小段（所有元素或者部分元素）。

## 自测练习题答案

1. Hip Hip Hurray

```
2. #include <iostream>
using namespace std;
void stars(int n)
{
    cout << '*';
    if (n > 1)
        stars(n - 1);
}
```

下面的代码是正确的，但稍微复杂一些：

```
void stars(int n)
{
    if (n <= 1)
    {
        cout << '*';
    }
    else
    {
        stars(n - 1);
        cout << ' *';
    }
}
```

```
3. #include <iostream>
using namespace std;
void backward (int n)
{
    if (n < 1)
        return;
    cout << n << " ";
    backward(n - 1);
}
```

```

        if (n < 10)
        {
            cout << n;
        }
        else
        {
            cout << (n%10); // 输出最后一位。
            backward(n/10); // 倒序输出其他位。
        }
    }
}

```

4-5. 自测练习题 4 的答案是 writeUp; , 自测练习题 5 的答案是 writeDown; 。

```

#include<iostream>
using std::cout;
using std::endl;
void writeDown(int n)
{
    if (n >= 1)
    {
        cout << n << " "; // 递归调用时正序输出。
        writeDown(n - 1);
    }
}

//5
void writeUp(int n)
{
    if (n >= 1)
    {
        writeUp(n - 1);
        cout << n << " "; // 递归调用时逆序输出。
    }
}

// 自测练习题 4 和自测练习题 5 的测试代码。
int main()
{
    cout << "calling writeUp(" << 10 << ")\n";
    writeUp(10);
    cout << endl;
    cout << "calling writeDown(" << 10 << ")\n";
    writeDown(10);
    cout << endl;
    return 0;
}

/* 测试结果。
calling writeUp(10)
1 2 3 4 5 6 7 8 9 10
calling writeDown(10)
10 9 8 7 6 5 4 3 2 1*/

```

6. “栈溢出”的错误信息是告诉你计算机正试图向栈中放置超出系统允许的更多的活动帧。这种情况的发生很可能是由于程序中出现了无限的递归调用。

```

7. void cheers(int n)
{
    while (n > 1)
    {
        cout << "Hip ";
        n--;
    }
    cout << "Hurray\n";
}

8. void stars(int n)
{
    for (int count = 1; count <= n; count++)
        cout << ' *';
}

9. void backward(int n)
{
    while (n >= 10)
    {
        cout << (n%10); // 输出 n 的最后一位。
        n = n/10; // 忽略 n 的最后一位。
    }
    cout << n;
}

```

10. 对自测练习题 4 的递归函数的追踪。如果  $n = 3$ ，要执行下面的代码：

```

if (3 >= 1)
{
    writeDown(3 - 1);
}

```

下一次递归调用， $n = 2$  时，执行下面的代码：

```

if (2 >= 1)
{
    writeDown(2 - 1);
}

```

下一次递归调用， $n = 1$  时，执行下面的代码：

```

if (1 >= 1)
{
    writeDown(1 - 1);
}

```

最后一次递归调用， $n = 0$  时，true 条件的代码部分不会执行：

```

if (0 >= 1) // 条件表达式为 false。
{
    // 这部分代码被忽略。
}

```

展开递归调用过程，语句 `cout << n << " "`；在栈中的每次递归调用执行这

行代码，分别是  $n = 3$ ，然后是  $n = 2$ ，最后是  $n = 1$ 。输出是 3 2 1。

11. 对自测练习题 5 的递归函数的追踪。如果  $n=3$ ，要执行下面的代码：

```
if (3 >= 1)
{
    cout << 3 << " ";
    writeUp(3 - 1);
}
```

下一次递归调用， $n = 2$  时，执行下面的代码：

```
if (2 >= 1)
{
    cout << 2 << " ";
    writeUp(2 - 1);
}
```

下一次递归调用， $n = 1$  时，执行下面的代码：

```
if (1 >= 1)
{
    cout << 1 << " ";
    writeUp(1 - 1);
}
```

最后一次递归调用， $n = 0$  时，true 条件的代码部分不会执行：

```
if (0 >= 1) // 条件表达式为 false，函数体代码将被忽略执行。
{
    // 这部分代码被忽略。
}
```

展开递归调用过程，输出（根据计算机对栈的操作顺序）是 1 2 3。

12. 6

13. 输出是 24。该函数是计算阶乘的函数，通常写为  $n!$ ，其定义如下：

$n!$  is equal to  $n * (n - 1) * (n - 2) * \dots * 1$

14. // 使用 iostream 和 cstdlib.

```
double power(int x, int n)
{
    if (n < 0 && x == 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }
    if (n < 0)
        return (1/power(x, -n));
    else if (n > 0)
        return (power(x, n - 1)*x);
    else// n == 0
        return (1.0);
}
```

15. 会终止的，因为这些函数都只是简单地递归返回某个数值，并没有其他操作，属于尾递归。



```

16. int odd(int num)
{
    if (num == 0)
        return 1;
    else
        return even(num - 1);
}

17. int squares(int n)
{
    if (n <= 1)
        return 1;
    else
        return (squares(n - 1) + n*n);
}

```

## 编程练习

1. 写一个递归函数的定义，要求该函数有一个 int 类型的形参，返回值为传入参数的斐波那契数列 (Fibonacci) 的值。斐波那契数列的定义如下： $F_0$  是 1,  $F_1$  是 1,  $F_2$  是 2,  $F_3$  是 3,  $F_4$  是 5, 对于任意一个值, 有  $F_{i+2} = F_i + F_{i+1}$ , 其中  $i = 0, 1, 2, \dots$

2. 从  $n$  个事物中去掉不同的  $r$  个事物的组合公式如下：

$$C(n, r) = n! / (r! * (n - r)!)$$

阶乘函数  $n!$  的定义如下,

$$n! = n * (n-1) * (n-2) * \dots * 1$$

为计算组合的公式  $C(n, r)$  设计一个递归实现的函数版本, 来计算某一组数值的组合函数的值。将该函数的实现嵌入到程序中并进行测试。

3. 写一个递归函数的实现。该函数有一个类型为字符数组的参数和两个类型为整数的数组索引的参数。该函数将对位置在两个参数之间的数组元素进行排序。例如, 如果函数的输入字符数组为：

```
a[1] == 'A' a[2] == 'B' a[3] == 'C' a[4] == 'D' a[5] == 'E'
```

两个数组索引为 2 和 5, 那么函数执行后数组元素为：

```
a[1] == 'A' a[2] == 'E' a[3] == 'D' a[4] == 'C' a[5] == 'B'
```

在程序中嵌入该函数并测试它。上面的函数调试成功后, 再定义另外一个函数, 该函数只有一个用于存储字符串值的类型为字符数组的参数。该函数执行后会将参数中的字符串倒序。应该在该函数中包含对上一个函数的调用。在程序中嵌入该函数并测试。

4. 为上一个练习中的递归函数写一个迭代版本的实现。嵌入到程序中并进行测试。

5. 汉诺塔问题。有这样一个关于一个和尚和 64 个石头圆盘的故事：如果这个和尚只借助于第三个塔座，能将这 64 个圆盘从一个塔座上移到另外一个塔座上，要求在转移圆盘的过程中小的圆盘始终在大圆盘之上，并且每次只能移动一个圆盘，当这个和尚完成这个工作时，世界末日就要来临了。世界末日论留给关心它们的人去研究吧，我们关心的将是该问题的递归解决方案。

三个塔座中的一个塔座上插有直径大小各不相同的  $n$  个圆盘，它们按照由小到大的顺序编号为 1, 2, 3, ...,  $n$ ，目的是将插有圆盘的塔座上的圆盘全部转移到另一个空的塔座上，要求每次只能移动一个圆盘。在转移的过程中，你可以借助于第三个塔座，并且在转移圆盘的过程中，要求任意时刻都不能将一个较大的圆盘压在较小的圆盘之上。本练习的要求是写一个递归函数来描述这个问题的实现。由于我们没有图形显示，所以应该输出解决该问题的一个序列索引。

提示：如果你能将第一个塔座上的  $n-1$  个圆盘从第一个塔座借助于第二个塔座顺序转移到第三个塔座上，那么最大的圆盘就可以从第一个塔座转移到第二个塔座上。用同样的技术，你可以将位于第三个塔座上的  $n-1$  个圆盘借助于第一个塔座从第三个塔座顺序转移到第二个塔座上，这样就解决了问题。只要确定非递归的情况是什么，递归情况是什么，以及何时输出移动圆盘的说明即可。

6. (在做本练习之前，应该首先完成编程练习 1) 在本练习中，将比较计算斐波那契数列函数的递归实现和迭代实现两个版本的效率。

- 检查递归实现计算斐波那契数列的值的函数。注意，对于每个斐波那契数列值的计算，都存在很多的重复计算。为了避免这种重复计算，将编程练习 1 用迭代方式（即使用循环）来实现，而不是用递归实现。你应该每次在需要时计算一个斐波那契数列值，当它们不再需要时忽略该数值。
- 以计算  $1^{\text{st}}$ 、 $3^{\text{rd}}$ 、 $5^{\text{th}}$ 、 $7^{\text{th}}$ 、 $9^{\text{th}}$ 、 $11^{\text{th}}$ 、 $13^{\text{th}}$  和  $15^{\text{th}}$  的斐波那契数列值为例，来比较实现练习 1 和本练习 (a) 的执行时间。确定每个函数的执行时间，比较并评价两种算法。

提示：如果你的操作系统为 Linux 的话，可以运用 `Bash time` 工具，它将给出实际时间、用户时间（你的程序占用的 CPU 循环时间）和系统时间（不是运行程序所需要的 CPU 循环时间）。如果操作系统是其他系统时，请阅读系统操作手册，或者向老师请教在你的系统中如何测试程序运行的时间。

7. (在做本练习之前，应该首先完成编程练习 6) 在使用递归定义计算斐波那契数列的函数中，计算每个斐波那契数列值时存在很多重复计算。为了计算  $F_{i+2} = F_i + F_{i+1}$ ，此时要计算  $F_i$  的值，而  $F_i$  的值在计算  $F_{i+1}$  时已经被计算过一次。你可以在计算得到  $F_i$  的值后将它保存在一个数组中，以避免这种重复计算。基于此想法写出计算斐波那契数列值的递归实现的另一个版本。在计算  $N^{\text{th}}$  的斐波那契数列值时，声明一个大小为  $N$  的数组。数组的索引  $i$  用来存放第一次计算  $i^{\text{th}}$  ( $i \leq N$ ) 时的斐波那契数列值，然后在第二次要计算该值时使用已经存储在数组中的值，从而避免了重复计算。像编程练习 6 那样对该算法进行时间估计，并比较该算

法和迭代版本实现算法的效率。

8. 储蓄账户通常使用复利来增加存款。如果你存入 1000 美元，年利率是 10%，则经过一年后你将拥有 1100 美元。如果你将这些钱以 10% 年利率再存上一年，你将得到 1210 美元。经过三年之后你将拥有 1331 美元，等等。

编写一个程序，输入为初始存款额、年利率以及使用复利方式的存款年数。编写一个递归函数，使用上面的输入信息，计算储蓄账户中最终将有的存款额。

为验证你的函数，请注意存款额应当等于  $P(1+i)^n$ ，其中  $P$  是最初的存款， $i$  是年利率， $n$  是年数。

9. 设在某房间中有  $n$  个人， $n$  是大于或等于 1 的整数。每个人都与其他人分别握手一次，握手的总次数  $h(n)$  是多少？编写一个递归函数求解这个问题。从简单的情况开始考虑，如果房间中只有一或两个人，那么

```
handshake(1) = 0
handshake(2) = 1
```

如果第三个人进入房间，他就必须与原有的两个人分别握手。这就增加了两次握手，总的握手次数变成了三次。

如果第四个人进入房间，他就必须与房间中已有的三个人分别握手。这就增加了三次握手，即总的握手次数变成了六次。

如果你可以将这种情况推广到  $n$  次握手，那就应该能写出递归解决方案。

10. 考虑下面的保龄球瓶布局，其中 \* 表示一个球瓶：

```

      *
     **
    ***
   ****
  *****

```

上面共有五行 15 个球瓶。如果我们只有最上面的四行，那就是 10 个球瓶。如果我们只有最上面的三行，那就是六个球瓶。如果我们只有最上面的两行，那就是三个球瓶。如果我们只有最上面的一行，那就是一个球瓶。

编写一个递归函数，输入为行数  $n$ ，输出为在  $n$  行的锥形上的球瓶总数。你的程序应当允许大于 5 的  $n$  值。

11. 编写一个声明如下的函数：

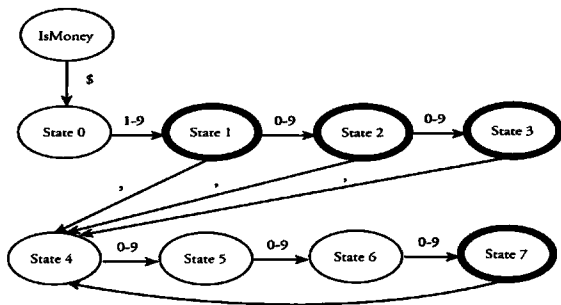
```
bool contains (char *haystack, char *needle)
```

这个函数用来判断 C 字符串 haystack 中是否包含 needle 的内容。打个比方：contains("C++ programming", "ogra") 应该返回 true，而 contains("C++ programming", "grammy") 应该返回 false。

不准使用 string 类的 substr 和 find 成员函数完成查找。

12. 下图是一个决定型有限状态机，或者叫作 DFA。这个 DFA 用来判断一个字

字符串是否符合带逗号的标准货币表示格式。举个例子，“\$1,000”和“\$25”以及“\$551,323,991,391”都是符合这种格式的，但是“1,000”（缺少开头的\$）和“\$1000”（缺少了一个逗号）以及“\$5424,132”（缺少了一个逗号）都是不符合的。



这个 DFA 是有效的，但是初始状态是 IsMoney。如果开头第一个字母是 \$，那么我们移动到第二个字符并将状态迁移到 State 0，否则断定这个字符串不是一个标准的货币字符串。在 State 0 中，如果第二个字符是 1 ~ 9 的数字，那么我们移动到第三个字符并将状态迁移到 State 1，否则断定这个字符串不是一个标准的货币字符串。在 State 1 中，如果接下来已经没有更多的字符了，我们断定这个字符串是一个标准的货币字符串。这个状态叫作结束状态，需要用加粗的椭圆表示。如果第三个字符是 1 ~ 9 的数字，我们移动到第四个字符并将状态迁移到 State 2。如果第三个字符是逗号，我们移动到第四个字符并将状态迁移到 State 4。否则，断定这个字符串不是一个标准的货币字符串。图中剩余的部分和上面描述的逻辑是相同的。

在一个程序中使用递归实现上述的 DFA。每个状态应该对应一个函数，每个函数中都应该调用图中箭头方向指示的下个状态对应的函数。由于存在从 State 7 到 State 4 的循环，所以一定会出现交叉递归。用若干字符串测试你的程序。

这种解决方案需要处理字符串中的每一个字符，但可以采用尾部递归的思想完成函数，这样处理长字符串时可能可以避免耗尽编译器栈空间的问题。

13. 从一个随机生成的方格表中寻找单词是一个流行的文字游戏。单词只能由不成角线的临近的最少三个字母组成，且字母不能被复用。比如下面是一个 4×4 的字母表格。

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

单词“FAB”是合法的（字母都在左上角），单词“KNIFE”也是合法的。单词“BABE”是不合法的但是字母“B”可以被复用。单词“MINE”是不合法的，因为字母“E”与字母“N”不是相邻的。

写一个程序可以用  $4 \times 4$  的二维表格代表游戏图板。程序可以随机地为图板上的空格选择一些字母，选择过程中最好让元音字母出现的概率大于辅音字母。而且，你也可以固定地将“U”安排在“Q”后面来组成一个“QU”并把它当作一个单独的字符。程序应该在文本文件 words.txt 中读取单词，然后使用一个递归的算法判断读取的单词是否可以用游戏图板中的字母组成。程序要输出所有符合条件的单词。

### 14.1 继承基础 514

派生类 514

派生类的构造函数 522

陷阱：使用基类的私有成员变量 524

陷阱：私有成员函数是无法被有效继承的 526

protected限定符 526

成员函数的重定义 529

重定义与重载 530

访问被重定义过的基类函数 531

不可被继承的函数 532

### 14.2 利用继承编程 533

派生类中的赋值运算符和拷贝构造函数 533

派生类的析构函数 534

示例：可备份的部分填充的数组 534

陷阱：赋值运算符两边是同一个对象 541

示例：PFArrayDBak的另一种实现 542

提示：一个类可以访问本类所有对象的私有成员 544

提示：“是一个”和“有一个” 544

保护继承和私有继承 545

多继承 546

# 第14章 继承

有其母必有其女。

俗语

## 概述

面向对象的编程是一种使用广泛并且功能强大的编程技术。姑且不说别的，它提供了一种被称之为继承的抽象机制。这意味着可以定义和编译一个类，用它来表示一种一般情形。之后这个类的属性可以被继承，从而定义出更多的具有特殊情形的类。本章总体描述什么是继承，特别是如何用 C++ 来实现继承。

本章并不会涉及第 12 章(文件 I/O)和第 13 章(递归)的内容,也不会涉及第 7 章 7.3 节关于向量的内容,本章的 14.1 节同样也不会涉及第 10 章(指针和动态数组)的知识。

## 14.1 继承基础

当我们希望孩子在某方面有所改进的时候，我们需要自我检验，这是否也是我们自己无法做到的。

——卡尔·古斯塔夫·荣格，《人格的整合》

### 派生类 基类

继承是一种过程，一个类通过继承自另外一个类而被创建，前者被称为派生类，而后者被称为基类。一个派生类自动拥有了基类的所有成员变量和普通的成员函数，除此之外，派生类还可以拥有自己额外的成员变量和额外的成员函数。

### 派生类

假设我们正在设计一个维护记录的程序，这些记录是关于固定薪水的员工和按小时计费的员工。很自然，这就存在某种层次上的类的划分，这些类囊括了所有具备员工属性的人。

按小时计费的员工是所有员工的一个子集，另外一个子集由那些按月或者按周拿薪水的员工组成。尽管这个程序可能不需要对应全体员工这个大集合的数据类型，但是从更一般的员工的概念来考虑也许会更有用。例如所有的员工都会有姓名和社会保险号，对于按小时计费的员工和固定薪水的员工来说，用于设置改变姓名和社会保险号的成员函数应该是相同的。

在 C++ 中你可以定义一个 Employee 类来包含所有的员工（无论是按小时计费的员工还是固定薪水的员工）。然后，利用这个类来分别定义两个类：时薪员工类和固定薪水员工类。

Employee 类还应该具有操作该类中数据域的成员函数。示例 14.1 和 14.2 给出了类 Employee 的一种可能的定义。

### 示例 14.1 基类 `Employee` 的接口

---

```

1
2 // 以下是头文件 employee.h。
3 // 该文件定义了类 Employee 的接口。
4 // 这个类主要用作基类，从而被不同的员工继承生成派生类。
5 #ifndef EMPLOYEE_H
6 #define EMPLOYEE_H

7 #include <string>
8 using std::string;

9 namespace SavitchEmployees
10 {

11     class Employee
12     {
13     public :
14         Employee( );
15         Employee(const string& theName, const string& theSsn);
16         string getName( ) const ;
17         string getSsn( ) const ;
18         double getNetPay( ) const ;
19         void setName(const string& newName);
20         void setSsn(const string& newSsn);
21         void setNetPay( double newNetPay);
22         void printCheck( ) const ;
23     private :
24         string name;
25         string ssn;
26         double netPay;
27     };

28 } //SavitchEmployees

29 #endif //EMPLOYEE_H

```

---

### 示例 14.2 基类 `Employee` 的实现

---

```

1
2 // 以下是文件 employee.cpp。
3 // 这是类 Employee 的实现文件。
4 // 类 Employee 的接口见头文件 employee.h。
5 #include <string>
6 #include <cstdlib>
7 #include <iostream>
8 #include "employee.h"
9 using std::string;
10 using std::cout;

11 namespace SavitchEmployees

```



```

12 {
13     Employee::Employee( ) : name("No name yet"),
                               ssn("No number yet"), netPay(0)
14 {
15     // 刻意为空。
16 }

17 Employee::Employee(const string& theName, const string& theNumber)
18 : name(theName), ssn(theNumber), netPay(0)
19 {
20     // 刻意为空。
21 }

22 string Employee::getName( ) const
23 {
24     return name;
25 }

26 string Employee::getSsn( ) const
27 {
28     return ssn;
29 }
30
31 double Employee::getNetPay( ) const
32 {
33     return netPay;
34 }

35 void Employee::setName(const string& newName)
36 {
37     name = newName;
38 }

39 void Employee::setSsn(const string& newSsn)
40 {
41     ssn = newSsn;
42 }

43 void Employee::setNetPay (double newNetPay)
44 {
45     netPay = newNetPay;
46 }

47 void Employee::printCheck( ) const
48 {
49     cout << "\nERROR: printCheck FUNCTION CALLED FOR AN \n"
50          << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
51          << "Check with the author of the program about this bug.\n";
52     exit(1);
53 }

54 } //SavitchEmployees

```

---

尽管你可以创建一个能够表示任何员工的 `Employee` 对象，但是我们之所以定义一个 `Employee` 类，目的是为了能够为更多的不同类型的员工创建其派生类。特别是，函数 `printCheck` 在不同的派生类当中会有不同的定义，因此才能为不同类型的员工打印不同的支票。示例 14.2 中，`Employee` 类的 `printCheck` 函数就反映了这一点。而为之之前一个可以表示任何员工的 `Employee` 对象打印一张支票是毫无意义的，因为我们根本无法区分是为哪种员工打印哪种支票。因此，如果我们这样去实现 `Employee` 类的 `printCheck` 方法，那么当一个基类 `Employee` 对象调用它的 `printCheck` 函数时，程序就会停止运行并且提示错误信息。正如你将要看到的那样，派生类需要有足够的信息去重新定义 `printCheck` 函数，从而才能够打印出有意义的员工支票。

从 `Employee` 类派生出的类将自动拥有基类 `Employee` 的全部成员变量 (`name`、`ssn` 和 `netPay`)，同时从 `Employee` 派生出的类也将自动拥有 `Employee` 的全部成员函数，例如 `printCheck`、`getName`、`setName` 以及示例 14.1 所列出的其他成员函数。这通常被称为派生类继承了成员变量和成员函数。

示例 14.3 和示例 14.4 给出了 `Employee` 类的两个派生类的接口定义文件，其中示例 14.3 给出了表示按小时计费员工的 `HourlyEmployee` 派生类，示例 14.4 给出了固定薪水员工的派生类 `SalariedEmployee`。我们将基类 `Employee` 和两个派生类放到了同一个命名空间内。尽管 C++ 不要求必须将它们放到同一个命名空间，但是既然它们是相互关联的类，那么放到同一个命名空间也是有一定道理的。接下来我们将先探讨 `HourlyEmployee` 类，如示例 14.3 所示。

### 示例 14.3 派生类 `HourlyEmployee` 的接口

```

1
2 // 这是头文件 hourlyemployee.h.
3 // 它是类 HourlyEmployee 的接口。
4 #ifndef HOURLYEMPLOYEE_H
5 #define HOURLYEMPLOYEE_H

6 #include <string>
7 #include "employee.h"

8 using std::string;

9 namespace SavitchEmployees
10 {

11     class HourlyEmployee : public Employee
12     {
13     public :
14         HourlyEmployee() ;
15         HourlyEmployee(const string& theName, const string& theSsn,
16                        double theWageRate, double theHours);
17         void setRate(double newWageRate);
18         double getRate() const ;
19         void setHours(double hoursWorked);
20         double getHours() const ;
21         void printCheck();

```

只有要改变基类的函数定义时，我们才有必要在派生类的声明中列出该函数的声明。

```

22     private :
23         double wageRate;
24         double hours;
25     };

26 } //SavitchEmployees
27 #endif //HOURLYEMPLOYEE_H

```

---

注意，一个派生类定义的开头部分和其他普通类的开头部分基本一样，但是区别在于紧接着类名之后要加一个冒号、C++ 保留关键字 `public` 和基类的类名。在示例 14.3 中如下所示：

```

class HourlyEmployee : public Employee
{

```

派生类 `HourlyEmployee` 自动接受了基类 `Employee` 的所有成员变量和成员函数，但是 `HourlyEmployee` 还可以添加自己额外的成员变量和成员函数。

#### 示例 14.4 派生类 `SalariedEmployee` 的接口

---

```

1
2 // 这是头文件 salariedemployee.h.
3 // 这是类 SalariedEmployee 的接口。
4 #ifndef SALARIEDEMPLOYEE_H
5 #define SALARIEDEMPLOYEE_H

6 #include <string>
7 #include "employee.h"

8 using std::string;

9 namespace SavitchEmployees
10 {

11     class SalariedEmployee : public Employee
12     {
13     public :
14         SalariedEmployee( );
15         SalariedEmployee (const string& theName, const string& theSsn,
16                             double theWeeklySalary);
17         double getSalary( ) const ;
18         void setSalary( double newSalary);
19         void printCheck( );
20     private :
21         double salary; // 周薪
22     };

23 } //SavitchEmployees
24 #endif //SALARIEDEMPLOYEE_H

```

---

派生类 `HourlyEmployee` 定义时并没有提到成员变量 `name`、`ssn` 和 `netPay`，

但是任何一个 `HourlyEmployee` 类的对象都有名为 `name`、`ssn` 和 `netPay` 的成员变量，这些成员变量 `name`、`ssn` 和 `netPay` 就继承自基类 `Employee`。另外 `HourlyEmployee` 又声明了两个额外的名为 `wageRate` 和 `hours` 的成员变量。因此任何一个 `HourlyEmployee` 类的对象都有五个成员变量，分别是 `name`、`ssn`、`netPay`、`wageRate` 和 `hours`。注意定义派生类 `HourlyEmployee` 时，只需要列出两个额外的成员变量。在此并没有提到基类中已经声明过的成员变量，这些变量会自动提供给派生类。

正如 `HourlyEmployee` 继承了基类 `Employee` 的成员变量，`HourlyEmployee` 也继承了 `Employee` 的所有成员函数。因此类 `HourlyEmployee` 从基类 `Employee` 继承了成员函数 `getName`、`getSsn`、`getNetPay`、`setName`、`setSsn`、`setNetPay` 和 `printCheck`。

### 继承而来的成员

派生类自动拥有基类的所有成员变量和所有的普通成员函数（本章稍后将会讨论到，有一些特殊的成员函数，例如构造函数，是不可以被自动继承的）。这些来自基类的成员被称为继承而来的。这些继承而来的成员变量和成员函数的特殊之处在于并没有被派生类所提到，但是已经自动成为派生类的成员了。但是有一种例外，正如代码中所描述的那样，当你想要改变一个继承而来的成员函数时，你确实可以在派生类中声明一个已经被基类声明过的成员函数。

除了继承而来的成员变量和成员函数之外，一个派生类还可以添加新的成员变量和成员函数。新的成员变量和成员函数的声明被列在类的定义当中。例如派生类 `HourlyEmployee` 添加了两个新的成员变量 `wageRate` 和 `hours`，并且添加了新的成员函数 `setRate`、`getRate`、`setHours` 和 `getHours`。这正如示例 14.3 所示。注意：你不需要在派生类中给出继承而来的成员函数的声明，除非你想要重新定义这些继承的成员函数（我们稍后将会讨论这一点）。现在，不需要担心派生类的构造函数是如何定义的，我们将在下一节中讨论。

在派生类的实现文件中，例如示例 14.5 所示的 `HourEmployee` 实现，我们将会给出所有额外添加的成员函数的定义。注意，不需要在派生类中为继承而来的成员函数给出定义，除非该派生类中该成员函数确实需要被重新定义，这一点我们还要在后面讨论。

### 父类和子类

当讨论派生类时，人们经常采用源自描述家庭关系时的术语。基类通常也被称为父类，派生类通常也被称为子类。这样便于人们更好理解“继承”所表达的关系。举例来说，我们可以说一个子类从父类那里继承了成员变量和成员函数。这种类比通常可以进一步引申，一个类，如果是另外一个类的父类的父类的父类（或进一步嵌套），则它通常被称为祖先类。如果 A 类是 B 类的祖先，那么 B 类也通常被称为 A 类的子孙。

**重定义** 继承而来的成员函数在派生类中可以被重新定义，从而使该成员函数在派生类中与它在基类当中的含义不同。这被称为重定义继承而来的成员函数。例如，成员函数 `printCheck()` 在派生类 `HourlyEmployee` 的定义当中被重定义了。如何对成员函数进行重定义，简单来说就是在类的定义当中列出它并且给它一个新的函数实现，这与为一个派生类添加一个新的成员函数所要做的工作一样。在示例 14.3 和示例 14.5 中，通过在派生类 `HourlyEmployee` 中重定义 `printCheck()` 进行了阐述。

#### 示例 14.5 派生类 `HourlyEmployee` 的实现

```

1  // 这是 hourlyemployee.cpp 文件。
2  // 它是类 HourlyEmployee 的实现。
3  // 类 HourlyEmployee 的接口
4  // 在头文件 hourlyemployee.h 中。
5  #include <string>
6  #include <iostream>
7  #include "hourlyemployee.h"
8  using std::string;
9  using std::cout;
10 using std::endl;

11 namespace SavitchEmployees
12 {

13     HourlyEmployee::HourlyEmployee() : Employee(), wageRate(0), hours(0)
14     {
15         // 函数体刻意为空。
16     }

17     HourlyEmployee::HourlyEmployee(const string& theName,
18                                     const string& theNumber, double theWageRate, double theHours)
19         : Employee(theName, theNumber), wageRate(theWageRate),
20           hours(theHours)
21     {
22         // 函数体刻意为空。
23     }

24     void HourlyEmployee::setRate(double newWageRate)
25     {
26         wageRate = newWageRate;
27     }

28     double HourlyEmployee::getRate() const
29     {
30         return wageRate;
31     }

32     void HourlyEmployee::setHours(double hoursWorked)
33     {
34         hours = hoursWorked;
35     }

36     double HourlyEmployee::getHours() const
37     {

```

我们选择把设置 `netPay` 的值作为函数 `printCheck` 的一部分，是因为 `netPay` 的值在本函数中被调用到了。不管怎样，这只是编程思想的问题而非编码自身的问题。但是需要注意的是，当我们在派生类 `HourlyEmployee` 中重定义 `printCheck` 函数时，C++ 允许我们省略 `const` 关键字。

```

38         return hours;
39     }

40     void HourlyEmployee::printCheck( )
41     {
42         setNetPay(hours * wageRate);

43         cout << "\n\n";
44         cout << "Pay to the order of " << getName( ) << endl;
45         cout << "The sum of " << getNetPay( ) << " Dollars\n";
46         cout << "\n\n";
47         cout << "Check Stub: NOT NEGOTIABLE\n";
48         cout << "Employee Number: " << getSsn( ) << endl;
49         cout << "Hourly Employee. \nHours worked: " << hours
50             << " Rate: " << wageRate << " Pay: " << getNetPay( ) <<
                endl;
51         cout << "\n\n";
52     }

53 } //SavitchEmployees

```

---

SalariedEmployee 是基类 Employee 的另一个派生类，示例 14.4 给出了类 SalariedEmployee 的声明接口，示例 14.6 给出了它的具体实现。一个 SalariedEmployee 类型的对象除了拥有自身新定义的成员变量和成员函数外，还拥有类 Employee 的全部成员变量和成员函数。这一点是毫无疑问的（即使在类 SalariedEmployee 的定义中没有列出任何一个继承来的成员变量，并且只列出了类 Employee 的一个函数，而这个函数却是因为需要在类 SalariedEmployee 中做重新定义的 printCheck 函数）。虽然如此，类 SalariedEmployee 却拥有了成员变量 name、ssn 和 netPay，以及新增的成员变量 salary。注意，在派生类中我们不需要去为其声明基类 Employee 的成员变量和成员函数，如变量 name 和函数 setName。因为 SalariedEmployee 会自动地从基类那里继承这些成员变量和成员数，而不需要程序员去做任何事情。

需要注意的是，类 Employee 所有的代码对于类 HourlyEmployee 和类 SalariedEmployee 来说都是共有的，这样就可以省去同样的代码写两次的麻烦：一次为类 HourlyEmployee，另一次为 SalariedEmployee。继承允许你重用类 Employee 中的代码。

#### 示例 14.6 派生类 SalariedEmployee 的实现

---

```

1
2 // 这是 salariedemployee.cpp 文件。
3 // 它是类 SalariedEmployee 的实现。
4 // 类 SalariedEmployee 的声明接口在
5 // 头文件 salariedemployee.h 中。
6 #include <iostream>
7 #include <string>
8 #include "salariedemployee.h"

```

```

9  using std::string;
10 using std::cout;
11 using std::endl;

12 namespace SavitchEmployees
13 {
14     SalariedEmployee::SalariedEmployee( ) : Employee( ), salary(0)
15     {
16         // 函数体刻意为空。
17     }

18     SalariedEmployee::SalariedEmployee(const string& theName,
19                                         const string& theNumber,
20                                         double theWeeklyPay)
21     : Employee(theName, theNumber), salary(theWeeklyPay)
22     {
23         // 函数体刻意为空。
24     }

25     double SalariedEmployee::getSalary( ) const
26     {
27         return salary;
28     }

29     void SalariedEmployee::setSalary(double newSalary)
30     {
31         salary = newSalary;
32     }

33     void SalariedEmployee::printCheck( )
34     {
35         setNetPay(salary);
36         cout << "\n_____ \n";
37         cout << "Pay to the order of " << getName( ) << endl;
38         cout << "The sum of " << getNetPay( ) << " Dollars\n";
39         cout << "_____ \n";
40         cout << "Check Stub NOT NEGOTIABLE \n";
41         cout << "Employee Number: " << getSan( ) << endl;
42         cout << "Salaried Employee. Regular Pay: "
43             << salary << endl;
44         cout << "_____ \n";
45     }
46 } //SavitchEmployees

```

### 派生类的构造函数

基类的构造函数是无法被派生类所继承的，但是你可以在派生类的构造函数中调用触发基类的构造函数，这正是我们所期望和想要的。派生类的构造函数通过一种特殊的方式调用基类的构造函数，基类的构造函数用于初始化所有从基类继承而来的成员变量。因此，派生类的构造函数是以调用基类的构造函数为开始的。例 14.5 给出了

派生类 `HourlyEmployee` 构造函数的定义,里面描述了调用基类构造函数的特殊语法。下面是从示例 14.5 中拷贝出来的类 `HourlyEmployee` 的构造函数的定义(仅对代码的换行方面做了少许的改动):

```
HourlyEmployee::HourlyEmployee(const string&theName,
                                const string& theNumber, double theWageRate,
                                double theHours)
    : Employee(theName, theNumber),
      wageRate(theWageRate), hours(theHours)
{
    // 函数体刻意为空。
}
```

冒号后面的部分是构造函数 `HourlyEmployee::HourlyEmployee` 定义中的初始化部分, `Employee (theName, theNumber)` 部分是对基类 `Employee` 中有两个参数的构造函数的调用。应该注意的是,调用基类构造函数的语法与初始化成员变量的语法类似: `wageRate(theWageRate)` 表示将成员变量 `wageRate` 的值赋值为 `theWageRate`; 而 `Employee(theName, theNumber)` 表示用实参 `theName` 和 `theNumber` 来调用基类的构造函数 `Employee`。由于所有的工作都在初始化的时候被完成,所以函数体部分是空的。

在下面的代码中,我们重新定义了从示例 14.5 中所拷贝出来的类 `HourlyEmployee` 的另外一个构造函数:

```
HourlyEmployee::HourlyEmployee( ) : Employee( ), wageRate(0),
                                    hours(0)
{
    // 函数体刻意为空。
}
```

### 派生类的对象有多种类型

根据日常生活的经验,一个按小时计费的员工也属于员工,在 C++ 中也同样成立。既然 `HourlyEmployee` 是类 `Employee` 的一个派生类,那么在任何 `Employee` 类的对象可以使用的地方也同样可以使用 `HourlyEmployee` 类的对象。通常的使用方法是,当一个函数需要 `Employee` 类型的参数时,你可以将一个 `HourlyEmployee` 类型的对象作为参数传递给该函数。你也可以将一个 `HourlyEmployee` 类型的对象赋值给一个 `Employee` 类型的变量(特别注意:不可以把纯 `Employee` 类型的对象赋值给一个 `HourlyEmployee` 类型的变量,毕竟一个员工不一定必须得是按小时计费的员工)。当然,这些注释适用于任何基类以及它的派生类。你可以在任意允许使用基类对象的地方使用派生类对象。

甚至,一个类的对象总是可以使用在它的祖先可以使用的任何地方。如果一个 `Child` 派生类继承自类 `Ancestor`, 而一个子孙类 `GrandChild` 继承自该 `Child` 类,那么这个 `GrandChild` 类的对象既可以使用在 `Child` 类的对象可以使用的任何地方,又可以用于 `Ancestor` 类的对象可以使用的任何地方。



在这个构造函数的定义中，调用了基类的默认构造函数（无参数的）来初始化继承来的成员变量。你应该总是在派生类构造函数的初始化部分调用基类的某个构造函数。

如果派生类的构造函数没有调用基类的构造函数，那么基类的默认构造函数（无参数的）会被自动调用。因此，下面类 `HourlyEmployee` 的默认构造函数的定义（省略了 `Employee()` 与我们刚刚讨论的那个构造函数是等价的）：

```
HourlyEmployee::HourlyEmployee() : wageRate(0), hours(0)
{
    // 函数体刻意为空。
}
```

然而，我们更倾向于显式地调用基类的构造函数（尽管它可以被自动调用）。

派生类的对象拥有基类的全部成员变量，当一个派生类的构造函数被调用时，这些成员变量应该被分配内存并且被初始化。必须在基类的构造函数中为继承来的成员变量分配内存，并且由基类的构造函数来为这些继承来的成员变量初始化更为便捷。这就是当你在定义派生类的构造函数时，总是应该调用某个基类的构造函数的原因。如果没有包含对基类构造函数的调用（在派生类构造函数定义的初始化部分），那么基类默认（无参数的）的构造函数会被自动调用（如果基类不存在默认构造函数，那么将会出错）。

构造函数  
调用顺序

派生类的构造函数首先应该做的就是调用基类的构造函数。因此，如果类 `B` 继承自类 `A`，而类 `C` 继承自类 `B`，那么当类 `C` 的对象被创建时，首先是类 `A` 的构造函数被调用，然后是类 `B` 的构造函数被调用，最后才是调用类 `C` 的构造函数。

### 派生类的构造函数

派生类并不继承其父类的构造函数，然而，在定义派生类的构造函数时，你可以并且也应该包含基类构造函数的调用（在构造函数定义的初始化部分）。

如果你没有包含基类构造函数的调用，那么当调用派生类的构造函数时，基类默认的（无参数的）构造函数将会被自动调用。



### 陷阱：使用基类的私有成员变量

类 `HourlyEmployee`（示例 14.3 和示例 14.5）从类 `Employee`（示例 14.1 和示例 14.2）继承了一个叫作 `name` 的成员变量。例如，下面将要对象 `joe` 的成员变量 `name` 的值赋为“Josephine”（同时把成员变量 `ssn` 赋值为“123-456-789”，将成员变量 `wageRate` 和 `Hours` 都赋值为 0）：

```
HourlyEmployee joe("Josephine", "123-45-6789", 0, 0);
```

如果你想要将 `joe.name` 赋值为“Mighty-Joe”，可以这样做：

```
joe.setName("Mighty-Joe");
```

处理继承而来的成员变量（如 `name`）时，必须多加小心。类 `HourlyEmployee`

的成员变量 `name` 继承自类 `Employee`，但是在类 `Employee` 的定义中成员变量 `name` 是私有的成员变量。这意味着只有在类 `Employee` 自身的成员变量内才可以直接访问到 `name`。在任何其他类的成员函数定义时都不可以按名字直接访问到基类的私有成员变量或者私有成员函数，即使在派生类的成员函数中也依然无法访问到。因此，尽管类 `HourlyEmployee` 中确实有一个成员变量叫作“`name`”（从类 `Employee` 中继承而来），但是在类 `HourlyEmployee` 的任何成员函数的定义中试图直接访问成员变量 `name` 也是非法的。

例如，下面是成员函数 `HourlyEmployee::printCheck`（摘自示例 14.5）函数体的前几行：

```
void HourlyEmployee::printCheck( )
{
    setNetPay(hours * wageRate);
    cout << "\n_____ \n";
    cout << "Pay to the order of " << getName( ) << endl;
    cout << "The sum of " << getNetPay( ) << " Dollars\n";
```

你也许想要知道为什么我们使用成员函数 `setNetPay` 来设置成员变量 `netPay` 的值。你可以尝试着像下面一样重写成员函数定义的开始部分：

```
void HourlyEmployee::printCheck( )
{
    netPay = hours * wageRate; // 非法使用 netPay.
```

正如注释所暗示的那样，这样重写是不行的。成员变量 `netPay` 是类 `Employee` 的私有成员变量，尽管派生类 `HourlyEmployee` 继承了变量 `netPay`，但是无法直接访问它。必须通过某个公有的成员函数才可以访问到成员变量 `netPay`。正确地完成类 `HourlyEmployee` 的成员函数 `printCheck` 定义的方法是我们示例 14.5 中所演示的那样（这部分代码已经在前面给出）。

变量 `name` 和 `netPay` 是继承而来的变量并且在基类当中是私有的，这也解释了为什么我们在函数 `HourlyEmployee::printCheck` 的定义中使用取值函数 `getName` 和 `getNetPay`，而不是简单地使用变量 `name` 和 `netPay`。你不可以直接按名字访问一个基类的私有成员变量，而应该使用基类中定义的公有的取值函数和赋值函数（例如 `getName` 和 `setName`，还记得吗？取值函数允许你访问某个类的成员变量，而赋值函数允许你改变一个类的成员变量的值，这两种函数在第 6 章讨论过）。

基类私有的成员变量不可以在派生类的成员函数定义中被访问到，这一点对于许多人来说看起来好像是错误的。毕竟，如果你是一个按小时计费的员工（小时工）而想要改变你的名字，没有人会说：“`name` 是 `Employee` 的私有成员变量”。如果你是一个小时工，自然而然地也是一名员工，在 `Java` 中这也是成立的。类 `HourlyEmployee` 的一个对象也是类 `Employee` 的一个对象。然而，如果在 `C++` 中允许派生类访问基类的私有成员变量和私有成员函数，那么当你随时想要访问基类的私有成员时，你只需要简单地创建一个派生类，就可以在它的成员函数中访问那个私有成员了。这意味着任何人只需要付出一点努力，就可以访问到所有的私有成员变量。这个例子足以说明问题了，但是更大的问题是这种错误更容易在无意中犯下而非刻意为之。如果一个基

在本章“protected 限定符”小节中，我们将要讨论一种可以避开对基类私有成员变量访问限制的方法。

这应该不是问题。私有成员函数仅应作为帮助函数来使用，因此它们的调用应当仅限于定义它们的类的内部。如果你希望一个成员函数被多个派生类当作帮助函数来使用，那么它就不仅仅只是一个帮助函数，你就更应该将这个函数定义为公有函数。

正如你所看到的，你不能在一个派生类的定义或者实现过程中访问基类的私有成员变量或者私有成员函数。这里有一种关于成员变量和成员函数的声明，可以使基类的成员变量和成员函数能够按名字被派生类所访问，但是不可以非派生的类中访问到。如果你在某个类的成员变量和成员函数的前面使用了限定符 `protected`，而不是 `private` 或者 `public`，那么对于非派生类的类或者函数来说，它的作用与在成员变量前面加 `private` 的作用是一样的，但是在派生类中可以按名字访问这些加了 `protected` 的变量。

**protected**

```
void HourlyEmployee::printCheck()
// 仅当类 Employee 的成员变量被标注为
//protected 而非 private 时，这个函数才能正常运行。
{
    netPay = hours * wageRate;
    cout << "\n\n";
    cout << "Pay to the order of " << name << endl;
    cout << "The sum of " << netPay << " Dollars\n";
    cout << "\n\n";
    cout << "Check Stub: NOT NEGOTIABLE\n";
    cout << "Employee Number: " << ssn << endl;
```

```
cout << "Hourly Employee. \nHours worked: " << hours
      << " Rate: " << wageRate << " Pay: " << netPay << endl;
cout << "_____ \n";
}
```

在派生类 `HourlyEmployee` 中, 继承而来的成员变量 `name`、`netPay` 和 `ssn` 可以通过它们的变量名访问, 只要它们在基类 `Employee` 中是以 `protected` (这一点与 `private` 相反) 限定的。然而, 对于不是从类 `Employee` 派生出来的其他任何类而言, 那些成员变量的访问权限就跟它们使用 `private` 限定的一样。

如果基类的成员变量被限定为 `protected` 的, 那么在任何一个派生类中这些成员变量也相当于被限定为 `protected` 的了。例如, 假设你定义了一个派生类 `Part-TimeHourlyEmployee` 继承自类 `HourlyEmployee`, 类 `PartTimeHourlyEmployee` 继承了类 `HourlyEmployee` 所有的成员变量, 包括了类 `HourlyEmployee` 从类 `Employee` 继承来的成员变量。因此, 类 `PartTimeHourlyEmployee` 拥有成员变量 `netPay`、`name` 和 `ssn`。如果这些成员变量在 `Employee` 中被限定为 `protected` 的, 那么在类 `PartTimeHourlyEmployee` 中就可以通过变量名来使用这些成员变量。

除了派生类 (和派生类的派生类, 等等) 外, 对于 `protected` 限定的成员变量的使用限制就跟它们使用 `private` 限定的一样。

我们之所以讨论保护型成员变量, 主要是因为后面将要介绍它们的使用, 需要提前熟悉它们。许多编程权威, 尽管不是全部, 认为使用保护型成员变量是一种不好的编程风格, 因为这样就破坏了隐藏类的内部具体实现的原则。他们认为所有的成员变量都应当被限定为 `private` 的。如果所有的成员变量都被标为 `private` 的, 那么继承来的成员变量就无法通过变量名被派生类的函数定义所访问。但是, 这并不像听起来那样糟糕, 继承来的私有成员变量仍然可以通过调用继承来的函数所访问, 这些函数来读取或者修改继承的私有变量。既然权威们在是否应该使用保护型成员上有分歧, 你只能自己决定是否采用它们。

### 保护型成员

如果你在某个类的成员变量前面使用了 `protected` 限定符 (而不是 `private` 或者 `public`), 那么除非对于派生类, 否则对于其他任何类或者函数, 它的应用效果都跟这个成员变量被使用了 `private` 限定一样。然而, 在这个类的派生类的成员函数定义当中可以按变量名来访问该成员变量。与此类似, 如果你在某个类的成员函数前面使用了 `protected` 限定符, 那么除非是派生类, 否则对于其他任何类或者函数, 它的应用效果都跟这个成员函数被使用了 `private` 限定一样。然而, 在这个类的派生类的成员函数定义中该保护型函数可以被调用。

保护型成员被派生类继承后就如同它们在派生类中已经被标示为 `protected` 的一样。换句话说, 如果某个基类的成员被标示为 `protected` 的, 那么在它的所有子孙类中都可以通过名字来访问该成员, 而不必仅仅局限于直接从这个基类派生出来的那些类。

## 自测练习题

1. 下面的程序是否是合法的（假设已经适当地添加了 `#include` 和 `using` 指令）？

```
void showEmployeeData(const Employee object);

int main( )
{
    HourlyEmployee joe("Mighty Joe",
                       "123-45-6789", 20.50, 40);
    SalariedEmployee boss("Mr. Big Shot",
                          "987-65-4321", 10500.50);

    showEmployeeData(joe);
    showEmployeeData(boss);

    return 0;
}

void showEmployeeData(const Employee object)
{
    cout << "Name: " << object.getName( ) << endl;
    cout << "Social Security Number: "
         << object.getSsn( ) << endl;
}
```

2. 已经给出基类 `Smart` 的定义如下代码所示，请给出它的派生类 `SmartBut` 的定义。不需要考虑 `#include` 指令或者命名空间的细节。

```
class Smart
{
public:
    Smart( );
    void printAnswer( ) const ;
protected:
    int a;
    int b;
};
```

这个派生类应当有一个新增的数据成员 `crazy`，数据类型为 `bool`，一个新增的成员函数，无参数，返回值类型为 `bool`，以及一个合适的构造函数。新增的成员函数名为 `isCrazy`。不必给出具体实现，只需要给出类的定义即可。

3. 下面的函数定义是否可以看作自测练习题2中所讨论的派生类 `SmartBut` 的成员函数 `isCrazy` 的一个合法的定义？请解释你的答案。（问题是该函数定义是否合法，而不是它是否合情合理）。

```
bool SmartBut::isCrazy( ) const
{
    if (a > b)
        return crazy;
    else
        return true ;
}
```

## 成员函数的重定义

在派生类 `HourlyEmployee` 的定义（示例 14.3）中，我们给出了新增成员函数 `setRate`、`getRate`、`setHours` 和 `getHours` 的声明，也同样给出了唯一一个从基类 `Employee` 继承来的函数的声明。如果在派生类中没有给出基类的某些继承而来的函数的声明（例如 `setName` 和 `setSsn`），那么这些函数就维持基类当中的定义不变。这样的函数在派生类 `HourlyEmployee` 和基类 `Employee` 中的定义都是一样的。当你定义一个像 `HourlyEmployee` 那样的派生类的时候，你只需要列出这样的成员函数的声明，它们将在派生类中被改变从而与基类中的定义不同。如果你看一下类 `HourlyEmployee`（示例 14.5）的实现，就会发现继承而来的成员函数 `printCheck` 已经被重新定义了。类 `SalariedEmployee` 也给出了成员函数 `printCheck` 的新定义，如同示例 14.6 所示的那样。而且，这两个派生类所给出的函数定义互不相同。派生类都对函数 `printCheck` 进行了重定义。

示例 14.7 的示例程序展示了派生类 `HourlyEmployee` 和 `SalariedEmployee` 的使用。

### 示例 14.7 派生类的使用

```

1  #include <iostream>
2  #include "hourlyemployee.h"
3  #include "salariedemployee.h"
4  using std::cout;
5  using std::endl;
6  using SavitchEmployees::HourlyEmployee;
7  using SavitchEmployees::SalariedEmployee;

8  int main( )
9  {
10     HourlyEmployee joe;
11     joe.setName("Mighty Joe");
12     joe.setSsn("123-45-6789");
13     joe.setRate(20.50);
14     joe.setHours(40);
15     cout << "Check for " << joe.getName( )
16         << " for " << joe.getHours( ) << " hours.\n";
17     joe.printCheck( );
18     cout << endl;

19     SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
20     cout << "Check for " << boss.getName( ) << endl;
21     boss.printCheck( );

22     return 0;
23 }
```

函数 `setName`、`setSsn`、`setRate`、`setHours` 和 `getName` 是从类 `Employee` 继承而来的。没有发生任何改变。函数 `printCheck` 是被重新定义的，函数 `getHours` 是派生类 `HourlyEmployee` 新增的函数。

#### 示例运行结果

Check for Mighty Joe for 40 hours.

---

Pay to the order of Mighty Joe

The sum of 820 Dollars

---

Check Stub: NOT NEGOTIABLE

Employee Number: 123-45-6789

Hourly Employee.

Hours worked: 40 Rate: 20.5 Pay: 820

---

Check for Mr. Big Shot

---

Pay to the order of Mr. Big Shot

The sum of 10500.5 Dollars

---

Check Stub NOT NEGOTIABLE

Employee Number: 987-65-4321

Salaried Employee. Regular Pay: 10500.5

---

### 重定义继承函数

派生类继承了基类所有的成员函数和成员变量。然而，如果派生类需要某个继承来的成员函数有不同的实现，那么这个函数就可能需要在派生类中被重新定义。当要重定义某个成员函数时，我们必须将它的声明列在派生类的定义当中（尽管这个声明与基类中的完全相同）。如果你不希望重新定义某个从基类继承而来的成员函数，那么就一定不要将它放在派生类的定义中列出。

### 重定义与重载

请不要把在派生类中重新定义一个函数定义与重载一个函数名字相混淆。当对函数定义进行重新定义的时候，派生类中所给出的新的函数与基类的函数拥有相同数目和类型的参数。当对函数进行重载时，派生类中的新函数与基类中函数的参数数目不同或者参数的类型不同，而且派生类同时拥有这两个函数。例如，假设我们将下列函数声明的成员函数加到类 `HourlyEmployee` 的定义中：

```
void setName(string firstName, string lastName);
```

类 `HourlyEmployee` 将要拥有这个带两个参数的函数 `setName`，同时又将从基类中继承得到一个带一个参数的函数 `setName`：

```
void setName(string newName);
```

类 `HourlyEmployee` 将拥有两个名为 `setName` 的函数，这就是对函数 `setName` 的重载。

从另一个方面来说，类 `Employee` 和类 `HourlyEmployee` 都定义了一个函数声明如下的函数。

```
void printCheck();
```

在这种情况下，类 `HourlyEmployee` 只有一个名为 `printCheck` 的函数，但是类 `HourlyEmployee` 的函数 `printCheck` 的定义与它在基类 `Employee` 中的定义不同，这时函数 `printCheck` 就是被重新定义了。

如果你对重定义和重载仍然感到迷惑，那么至少你还可以有一些值得欣慰的：它们都是合法的。因此，学会如何使用它们远比学会如何区分它们重要。不管怎么样，你还是应该了解它们的不同之处。

### 标识

一个函数的标识是指该函数的名字加上参数列表中的类型序列，而不包括 `const` 关键字及“&”（与）符号。根据这种标识的定义，当要重载一个函数名时，该函数名的两个函数定义必须拥有不同的函数表示（一些权威专家也将 `const` 关键字或 `&` 作为标识的一部分，但是这里关于标识的定义只用于解释重载）。如果一个函数在基类和派生类中的函数名相同，但是拥有不同的表示，那么这是重载，而不是重定义。

如我们在第4章所提到的，一些编译器事实上允许你在 `const` 和非 `const` 的基础上进行重载，但是你不能指望编译器，因为 C++ 标准是不允许这样做的。

### 访问被重定义过的基类函数

假设你重新定义了某个函数，从而使得该函数在派生类类中拥有与基类不同的函数定义，那么对于派生类来说，基类中的函数定义并没有完全丢失。然而，如果你想要通过派生类的对象来访问该函数在基类中的定义版本，通常需要采用某种方式来说“调用该函数在基类中定义的版本（尽管我是一个派生类对象）”。这种方式就是使用带有基类类名的作用域标识符，我们用一个例子来说明其中的细节。

考虑基类 `Employee`（示例 14.1）和派生类 `HourlyEmployee`（示例 14.3），函数 `printCheck` 在两个类中都被定义了。现在假设每个类都有一个对象，如下面的代码所示：

```
Employee JaneE;
HourlyEmployee SallyH;
```

那么

```
JaneE.printCheck();
```

调用了类 `Employee` 所给出的 `printCheck` 函数定义，而

```
SallyH.printCheck();
```

调用了类 `HourlyEmployee` 给出的 `printCheck` 函数定义。

但是如果你想要通过派生类的对象 `SallyH` 来调用基类 `Employee` 所给出的 `printCheck` 函数定义的版本，那么就应该像下面这样做：

```
SallyH.Employee::printCheck();
```



当然，你不太可能想要刻意调用 `Employee` 所定义的 `printCheck` 函数，但是对于其他类和函数而言，偶尔又有可能发生为派生类的对象调用某个重定义过的函数在基类中的定义版本。在自测练习题 6 中就有这样一个例子。

### 不可被继承的函数

一般而言，如果 `Derived` 是基类 `Base` 的一个派生类，那么基类 `Base` 中所有“普通”的函数都有可能被类 `Derived` 继承而成为它的成员。但是，由于特殊原因，基类中也有一些特殊的函数是不可以被继承的。我们已经知道构造函数是不可以被继承的，私有成员函数也是不可以被继承的，析构函数（将在 14.2 节中讨论）也同样是不可被继承的。

拷贝构造函数是不可以被继承的，但是如果你不在派生类（或其他任何类）中定义一个拷贝构造函数，那么 C++ 就会自动为它生成一个拷贝构造函数。然而默认的拷贝构造函数只是简单地拷贝成员变量的内容，对于成员变量中含有指针或动态数据的类来说，它是无法正确工作的。因此，如果你的类成员变量中含有指针、动态数组或者其他动态数据，那么你就应该为这个类定义一个拷贝构造函数。无论这个类是否是派生类，这都适用。

赋值运算符“=”也是不可以被继承的。如果基类 `Base` 定义了赋值运算符，但是派生类 `Derived` 没有定义，那么派生类 `Derived` 将会有有一个赋值运算符，但是这个赋值运算符只是 C++ 为你创建的默认的赋值运算符（前提是你没有在派生类 `Derived` 中显式定义赋值运算符），它和定义在基类 `Base` 中的赋值运算符没有任何关系。如何定义赋值运算符将在 14.2 节的“派生类中的赋值运算符和拷贝构造函数”部分进行讨论。

构造函数、析构函数和赋值运算符不可被继承是很正常的。为了正确执行它们的任务，它们需要基类所没有涉及的信息，换句话说，它们需要知道派生类中所引入的新的成员变量。

### 自测练习题

4. 类 `SalariedEmployee` 从基类 `Employee` 继承了函数 `getName` 和 `printCheck`（还有其他成员），但是在派生类 `SalariedEmployee` 的定义中只给出了函数 `printCheck` 的函数声明，为什么在类 `SalariedEmployee` 的定义中不给出函数 `getName` 的函数声明？
5. 请给出示例 14.4 所给出的基类 `SalariedEmployee` 的派生类 `TitledEmployee` 的定义。类 `TitledEmployee` 有一个新增的成员变量 `title`，其数据类型为 `string`，并且拥有两个新增的成员函数：`getTitle`，没有参数但是返回值类型为 `string`；`setTitle`，函数返回值类型为 `void`，并且带有一个 `string` 类型的参数。该类重定义了成员函数 `setName`。你只需要给出类的定义即可，而不需要任何实现。但是，需要给出所有必需的 `#include` 指令和所有 `using namespace` 指令，将类 `TitledEmployee` 放在命名空间 `SavitchEmployees` 中。

6. 请给出自测练习题 5 的答案中所给出的类 `TitledEmployee` 的构造函数定义，并且给出成员函数 `setName` 的重定义，函数 `setName` 应该在实现中将 `title` 插入到 `name` 中。不需要给出 `#include` 指令和命名空间的具体细节。
7. 如文中所述，重载的赋值运算符和拷贝构造函数是无法被继承的。这是否意味着，如果你不为派生类重载一个赋值运算符或拷贝构造函数，那么这个派生类就没有赋值运算符和拷贝构造函数？

## 14.2 利用继承编程

失之毫厘，差之千里。

俗语

本节将给出关于继承的一些更为巧妙的细节，并且给出了另外一个完整的例子。除此之外，还给出了一些关于继承和编程技巧的讨论。本节使用到了动态数组（第 10 章），并且绝大多数内容都与使用了动态数组、指针或者其他动态数据的类相关。

### 派生类中的赋值运算符和拷贝构造函数

重载的赋值运算符和构造函数是不可以被继承的，但是在为派生类定义重载赋值运算符和拷贝构造函数时，它们是可以被使用的，并且在绝大多数情况下也必须被使用。

当在派生类中重载赋值运算符时，通常都要使用基类重载的赋值运算符。为了便于理解下面将要给出的代码，请记住重载赋值运算符必须被定义为类的成员函数。

如果类 `Derived` 继承自类 `Base`，那么类 `Derived` 的重载赋值运算符定义的开始部分通常会像下面这样：

```
Derived& Derived::operator =(const Derived& rightSide)
{
    Base::operator=(rightSide);
```

函数体的第一行是对类 `Base` 的赋值运算符的调用，这涉及了继承而来的成员变量及这些变量的数据。重载的赋值运算符将继续设置类 `Derived` 所引入的新增成员变量。包含该技术的完整例子请参考本章后面的编程示例“可备份的部分填充的数组”。

在派生类中定义拷贝构造函数也是相类似的情形。如果 `Derived` 是 `Base` 的一个派生类，那么类 `Derived` 的拷贝构造函数的定义通常都使用类 `Base` 的拷贝构造函数来设置继承来的成员变量和它们的数据。代码的开始部分通常如下所示：

```
Derived::Derived(const Derived& Object)
    : Base(Object),          < 可能有更多的初始化 >
{
```

基类拷贝构造函数 `Base(Object)` 的调用，设置了正在创建的 `Derived` 对象中那些继承而来的成员变量。注意，既然 `Object` 是 `Derived` 类型的，那么它也是 `Base` 类型的。因此，`Object` 对于基类的拷贝构造函数来说也是一个合法的参数。包

含基类的拷贝构造函数的完整例子请参考本章后面的编程示例“可备份的部分填充的数组”。

当然，这些方法还无法正常运行，除非在基类中有一个正确的、可用的赋值运算符和一个正确的、可用的拷贝构造函数。这就意味着在基类的定义中必须包含一个拷贝构造函数，并且在基类中自动创建的赋值运算符能正常工作或者基类中有一个重载过的赋值运算符的定义。

## 派生类的析构函数

如果一个基类拥有一个正常的能运行的析构函数，那么在派生类中定义一个正常的可运行的析构函数就会相对容易一些。当派生类的析构函数被调用时，基类的析构函数就会被自动触发，因此不需要显式地写出调用基类的析构函数（它通常会自动触发）。派生类的析构函数只需要关心派生类新增的成员变量（和它们所指向的数据）是否需要使用 `delete`。至于继承来的成员变量是否需要调用 `delete`，则是基类的析构函数的任务了。

如果类 B 继承自类 A，而类 C 继承自类 B，那么当类 C 的对象超出作用域时，首先是类 C 的析构函数被调用，然后是类 B 的析构函数被调用，最后才是类 A 的析构函数被调用。注意，析构函数的调用顺序恰好与构造函数的调用顺序相反。在编程示例“可备份的部分填充的数组”中，我们给出了一个例子来说明如何编写派生类的析构函数。

---

## 示例：可备份的部分填充的数组

这个例子给出了第 10 章（示例 10.10）中所给出的部分填充数组类 `PFArrayD` 的一个派生类。为了便于参考，我们在示例 14.8 中再次给出了基类 `PFArrayD` 的头文件。在示例 14.9 中，我们将尽可能多地给出要讨论到的基类 `PFArrayD` 的实现。注意，我们对第 10 章所给出的定义做出一些重要的更改，把私有的成员变量变为保护型的成员变量。这将允许派生类的成员函数按照名字访问成员变量。

我们将以 `PFArrayD` 为基类定义一个名字为 `PFArrayDBak` 的派生类。一个派生类 `PFArrayDBak` 的对象将拥有基类 `PFArrayD` 的所有成员函数，并且可以当作基类 `PFArrayD` 的一个对象来使用。但是一个 `PFArrayDBak` 类的对象将拥有下列新增的特性：有一个名字为 `backup` 的成员函数，这个函数可以备份该部分填充的数组的所有数据，之后，程序员可以随时随地调用成员函数 `restore` 来恢复该部分填充的数组从而使之达到这样一种状态：能够恢复到最后一次调用 `backup` 函数之前的数据。如果你对这个新增的成员函数的意思不是很明白，那么可以参考示例 14.12 中的示例程序。

派生类 `PFArrayDBak` 的头文件在示例 14.10 中给出，类 `PFArrayDBak` 新增了两个成员变量用于维护部分填充的数组的备份：一个是类型为 `double*` 的变量 `b`，它指向一个用于备份当前（继承而来的）工作数组的动态数组；另外一个为 `int` 类型的成员变量 `usedB`，它用来表示备份的数组 `b` 填充了多少数据。既然没有办法去改变

PFFArrayD (或者 PFFArrayDBak) 的数组大小, 那么就没有必要来备份数组大小这个数据。所有用于操作一个部分填充的数组的基本功能函数均继承自基类 PFFArrayD 而并没有加以更改, 这些继承来的函数操作继承来的数组 a 和继承来的 int 类型变量 used, 就像它们在基类 PFFArrayD 中所做的一样。

类 PFFArrayDBak 新增的成员函数的实现在示例 14.11 中给出。派生类 PFFArrayDBak 的构造函数依赖于基类的构造函数去设置常规的部分填充的数组 (继承来的变量 a、used 和 capacity)。每一个构造函数也同样创建了一个与数组 a 同等大小的动态数组。第二个数组是数组 b, 用于备份数组 a 的数据。

成员函数 backup 从部分填充的数组 (a 和 used) 拷贝数据到位置 b 和 usedB, 如下列代码所示:

```
usedB = used;
for (int i = 0; i < usedB; i++)
    b[i] = a[i];
```

你应该注意到成员变量 a 和 used 在基类中是保护型的而不是私有型的, 否则上面的代码将是非法的, 因为上面的代码直接按照名字访问了继承来的成员变量。

成员函数 restore 只是反过来, 从 b 和 usedB 中拷贝数据到 a 和 used。

派生类 PFFArrayDBak 中重载的赋值运算符的定义, 开始部分调用了基类 PFFArrayD 中定义的赋值运算符, 它把赋值运算符右边的对象 (右操作数) 的成员变量 a、used 和 capacity 的值拷贝到赋值运算符左边的对象 (调用对象)。我们依赖于这样的事实: 这些都会在基类重载的赋值运算符的定义中正确地完成。也就是说, 当赋值运算符的两边是同一对象 (对于继承而来的部分) 时, 我们依赖于基类的赋值运算符也能够正常地运行, 并且我们假设赋值行为是从数组 a 中拷贝出来的一个独立的备份。派生类 PFFArrayDBak 中重载的赋值运算符, 其函数体的代码中也应该创建一个类似的数组 b 的拷贝。

既然赋值运算符左边和右边的对象可能大小不同, 那么我们必须创建一个新的数组 b (大多数情况下), 如下列代码所示:

```
if (oldCapacity != rightSide.capacity)
{
    delete [] b;
    b = new double [rightSide.capacity];
}
```

注意, if 语句中的布尔型表达式确保当赋值运算符两边的对象大小相等时数组 b 不会被删除。特别是, 当赋值运算符两边是同一个对象时, 数组 b 不会被删除。此后, 数组的拷贝可以继续。

派生类 PFFArrayDBak 的析构函数没有显式地提到继承而来的成员变量 a, 它只是简单地将数组 b 的内存空间释放以供再次使用。如下列代码所示:

```
delete [] b;
```

即使你在派生类 PFFArrayDBak 的析构函数中没有看到数组 a 被提到, 这个继承而来的数组 a 的内存空间仍然得到释放。当派生类 PFFArrayDBak 的析构函数被调用时,

调用的最后将自动调用基类 PFArrayD 的析构函数。基类析构函数包含数组 a 的释放代码，如下所示。

```
delete [] a;
```

示例 14.12 给出了类 PFArrayDBak 的示例程序。

## 示例 14.8 基类 PFArrayD 的接口

```
1 // 这是头文件 pfarrayd.h。它是类 PFArrayD 的接口文件。
2 // 该类的对象是部分填充 double 类型数据的数组。
3 #ifndef PFARRAYD_H
4 #define PFARRAYD_H
5
6     class PFArrayD
7     {
8     public:
9         PFArrayD( );
10        // 初始的容量为 50。
11
12        PFArrayD(int capacityValue);
13
14        PFArrayD(const PFArrayD& pfaObject);
15
16        void addElement(double element);
17        // 前提条件：数组没有满。
18        // 运行结果：元素被添加。
19
20        bool full( ) const ;
21        // 如果数组已满则返回 true，否则返回 false。
22
23        int getCapacity( ) const ;
24
25        int getNumberUsed( ) const ;
26        void emptyArray( );
27        // 将已用数量重置为 0，清空数组。
28
29        double& operator [](int index);
30        // 访问 0 号元素到 numberUsed - 1 号元素。
31        PFArrayD& operator =(const PFArrayD& rightSide);
32
33        ~PFArrayD( );
34
35    protected :
36        double *a; // 为 double 型数组。
37        int capacity; // 为数组的大小。
38        int used; // 当前使用的数组位置的个数。
39    };
40
41 #endif //PFARRAYD_H
```

### 示例 14.9 基类 PFArrayD 的实现

```

1  #include <iostream>
2  using std::cout;
3  #include "pfarrayd.h"

4  PFArrayD::PFArrayD( ) : capacity( 50), used( 0)
5  {
6      a = new double [capacity];
7  }

8  PFArrayD::PFArrayD(int size) : capacity(size), used( 0)
9  {
10     a = new double [capacity];
11 }

12 PFArrayD::PFArrayD(const PFArrayD& pfaObject)
13     :capacity(pfaObject.getCapacity( )), used(pfaObject.getNumberUsed( ))
14 {
15     a = new double [capacity];
16     for (int i =0; i < used; i++)
17         a[i] = pfaObject.a[i];
18 }
19 double & PFArrayD:: operator [](int index)
20 {
21     if (index >= used)
22     {
23         cout << "Illegal index in PFArrayD.\n";
24         exit(0);
25     }
26     return a[index];
27 }
28

29 PFArrayD& PFArrayD:: operator =(const PFArrayD& rightSide)
30 {
31     if (capacity != rightSide.capacity)
32     {
33         delete [] a;
34         a = new double [rightSide.capacity];
35     }

36     capacity = rightSide.capacity;
37     used = rightSide.used;
38     for (int i = 0; i < used; i++)
39         a[i] = rightSide.a[i];
40     return *this ;
41 }

42 PFArrayD::~PFArrayD( )
43 {
44     delete [] a;
45 }

```

这是部分实现文件 pfarrayd.cpp。  
完整的实现在示例 10.11 中给出，这里  
只是给出你在这章所需要的部分。

**示例 14.10 派生类 PFArrayDBak 的接口**


---

```

1 // 这是头文件 pfarrraydbak.h。它是类 PFArrayDBak 的接口文件。
2 // 这个类的对象是 double 类型部分填充的数组。
3 // 这个版本允许程序员对部分填充的数组最近存储的内容进行备份和恢复。
4 #ifndef PFARRAYDBAK_H
5 #define PFARRAYDBAK_H
6 #include "pfarrayd.h"

7 class PFArrayDBak : public PFArrayD
8 {
9 public :
10     PFArrayDBak( );
11     // 数组大小初始化为 50。

12     PFArrayDBak(int capacityValue);

13     PFArrayDBak(const PFArrayDBak& Object);

14     void backup( );
15     // 对部分填充数组进行备份,

16     void restore( );
17     // 将部分填充数组恢复到上一次保存的状态。
18     // 如果先前没有进行过备份。
19     // 则将部分填充数组置空。
20     PFArrayDBak& operator =(const PFArrayDBak& rightSide);

21     ~PFArrayDBak( );
22 private :
23     double *b; // 用于备份的数组。
24     int usedB; // 备份数组中已经使用的部分。
25 };

26 #endif //PFARRAYDBAK_H

```

---

**示例 14.11 派生类 PFArrayDBak 的实现**


---

```

1 // 这是文件 pfarrraydbak.cpp。
2 // 这是 PFArrayDBak 类的实现。
3 // 类 PFArrayDBak 的接口在文件 pfarrraydbak.h 中。
4 #include "pfarraydbak.h"
5 #include <iostream>
6 using std::cout;

7 PFArrayDBak::PFArrayDBak( ) : PFArrayD( ), usedB( 0)
8 {
9     b = new double [capacity];
10 }
11 PFArrayDBak::PFArrayDBak(int capacityValue) :
12     PFArrayD(capacityValue), usedB(0)
13 {
14     b = new double [capacity];

```

```

15 }

16 PFArryDBak::PFArryDBak(const PFArryDBak& oldObject)
17     : PFArryD(oldObject), usedB(0)
18 {
19     b = new double [capacity];
20     usedB = oldObject.usedB;
21     for (int i = 0; i < usedB; i++)
22         b[i] = oldObject.b[i];
23 }

24 void PFArryDBak::backup( )
25 {
26     usedB = used;
27     for (int i = 0; i < usedB; i++)
28         b[i] = a[i];
29 }
30
31 void PFArryDBak::restore( )
32 {
33     used = usedB;
34     for (int i = 0; i < used; i++)
35         a[i] = b[i];
36 }

37 PFArryDBak& PFArryDBak::operator =(const PFArryDBak& rightSide)
38 {
39     int oldCapacity = capacity;
40     PFArryD::operator =(rightSide);
41     if (oldCapacity != rightSide.capacity)
42     {
43         delete [] b;
44         b = new double [rightSide.capacity];
45     }

46     usedB = rightSide.usedB;
47     for (int i = 0; i < usedB; i++)
48         b[i] = rightSide.b[i];

49     return *this;
50 }

51 PFArryDBak::~PFArryDBak( )
52 {
53     delete [] b;
54 }

```

注意，b 是数组 a 的拷贝。我们不能用 b=a。

通过调用基类的赋值运算符来为基类的成员变量赋值。

基类 PFArryD 的析构函数会被自动调用，并且会执行 delete []a;

### 示例 14.12 类 PFArryDBak 的示例程序

```

1 // 演示类 PFArryDBak 的程序。
2 #include <iostream>
3 #include "pfarraydbak.h"

```



```

4  using std::cin;
5  using std::cout;
6  using std::endl;

7  void testPFArrayDBak( );
8  // 测试一下类 PFArrayDBak。

9  int main( )
10 {
11     cout << "This program tests the class PFArrayDBak.\n";
12     char ans;
13     do
14     {
15         testPFArrayDBak( );
16         cout << "Test again? (y/n) ";
17         cin >> ans;
18     } while ((ans == 'y') || (ans == 'Y'));

19     return 0;
20 }

21 void testPFArrayDBak( )
22 {
23     int cap;
24     cout << "Enter capacity of this super array: ";
25     cin >> cap;
26     PFArrayDBak a(cap);

27     cout << "Enter up to " << cap << " nonnegative numbers.\n";
28     cout << "Place a negative number at the end.\n";

29     double next;

30     cin >> next;
31     while ((next >= 0) && (!a.full( )))
32     {
33         a.addElement(next);
34         cin >> next;
35     }
36     if (next >= 0)
37     {
38         cout << "Could not read all numbers.\n";
39         // 清除没有读取到的输入。
40         while (next >= 0)
41             cin >> next;
42     }

43     int count = a.getNumberUsed( );
44     cout << "The following " << count
45         << " numbers read and stored:\n";
46     int index;
47     for (index = 0; index < count; index++)
48         cout << a[index] << " ";
49     cout << endl;

50     cout << "Backing up array.\n";

```

```

51     a.backup();
52     cout << "Emptying array.\n";
53     a.emptyArray();
54     cout << a.getNumberUsed()
55         << " numbers are now stored in the array.\n";

56     cout << "Restoring array.\n";
57     a.restore();
58     count = a.getNumberUsed();
59     cout << "The following " << count
60         << " numbers are now stored:\n";
61     for (index = 0; index < count; index++)
62         cout << a[index] << " ";
63     cout << endl;
64 }

```

### 示例运行结果

```

This program tests the class PFArrayDBak.
Enter capacity of this super array: 10
Enter up to 10 nonnegative numbers.
Place a negative number at the end.
1 2 3 4 5 -1
The following 5 numbers read and stored:
1 2 3 4 5
Backing up array.
Emptying array.
0 numbers are now stored in the array.
Restoring array.
The following 5 numbers are now stored:
1 2 3 4 5
Test again? (y/n) y
Enter capacity of this super array: 5
Enter up to 5 nonnegative numbers.
Place a negative number at the end.
1 2 3 4 5 6 7 -1
Could not read all numbers.
The following 5 numbers read and stored:
1 2 3 4 5
Backing up array.
Emptying array.
0 numbers are now stored in the array.
Restoring array.
The following 5 numbers are now stored:
1 2 3 4 5
Test again? (y/n) n

```



### 陷阱：赋值运算符两边是同一个对象

无论什么时候重载一个赋值运算符，都应当确保当赋值运算符两边是同一个对象时，定义也可以正常运行。在大多数情况下，你需要通过它自身的一些代码使之成为一种特例。在之前的示例程序“可备份的部分填充的数组”中已经给出这样一个例子。■

### 自测练习题

- 假设类 Child 继承自类 Parent, 而类 Grandchild 继承自类 Child。这个问题是关于这三个类 Parent、Child 和 Grandchild 的构造函数和析构函数的。当类 Grandchild 的构造函数被调用触发了, 那么将有哪些构造函数以什么样的顺序被调用? 当类 Grandchild 的析构函数被调用触发了, 那么将有哪些析构函数以什么样的顺序被调用?
- 下面对类 PFArrayDBak (参见示例 14.10 和示例 14.11) 的默认构造函数的定义是否合法? (省略了对基类构造函数的调用) 并且请解释你的答案。

```
PFArrayDBak::PFArrayDBak( ) : usedB(0)
{
    b = new double [capacity];
}
```

### 示例：PFArrayDBak 的另一种实现

从最开始来看, 为了给出派生类 PFArrayDBak 的成员变量的定义, 我们似乎需要把基类 PFArrayD 的成员变量定义成保护型的。毕竟, 许多成员函数要操作继承而来的成员变量, 例如 a、used 和 capacity。示例 14.11 所给出的实现确实按名字调用到了变量 a、used 和 capacity, 并且因此这些特殊的定义确实依赖于基类将它们定义为保护型的 (相对于私有型)。但是, 请多考虑一下, 如果我们在基类中有足够多的赋值函数和取值函数, 那么即使所有的成员变量在基类中都被定义成私有的 (而非保护型的), 我们仍然可以重写派生类 PFArrayDBak 的实现。

示例 14.13 给出了派生类 PFArrayDBak 的另外一种实现, 并且在基类的所有成员变量都被定义为私有型而非保护型的情况下仍然能够正确运行。与我们之前的实现不同的部分被阴影化了。绝大多数的变动都很明显, 但是也有一些地方值得注意。

考虑一下成员函数 backup, 在我们之前的实现 (见示例 14.11) 当中, 我们从 a 拷贝数组到 b。既然 a 现在是私有的, 我们就无法按名字访问它, 但是由于我们已经重载了方括号运算符 (operator[]), 因此它可用于 PFArrayD 类型的对象, 并且该运算符在类 PFArrayDBak 中被继承了。我们可以简单地通过调用对象来使用 operator[], 最终的效果就是将数组 a 拷贝到数组 b, 但是我们再也没有按名字来访问私有数组 a, 代码如下:

```
usedB = getNumberUsed( );
for (int i = 0; i < usedB; i++)
    b[i] = operator[](i);
```

请一定注意调用一个正在被定义的类的运算符的语法。如果 superArray 是类 PFArrayDBak 的一个对象, 那么在 superArray.backup( ) 的调用中, 记号 operator[ ](i) 的意思是 superArray[i]。

我们本来可以在成员函数 restore 的定义中使用符号 operator[ ](i), 并且

这样做也是很简单的：先调用继承而来的成员函数 `emptyArray` 来清空数组 `a`，然后调用成员函数 `addElement` 来添加已经备份的元素。通过这种方法，我们还在过程中设置了私有成员变量 `used`。

通过这种实现方式实现的类 `PFArrayDBak` 的使用方法与之前的那种实现方式一样。特别是，无论采用哪种实现方式，示例 14.12 中程序的运行效果都是一样的。

### 示例 14.13 类 `PFArrayDBak` 的另一种实现方式

```

1 // 这是文件 pfarraydbak.cpp.
2 // 它是类 PFArrayDBak 的实现,
3 // PFArrayDBak 类的接口文件在 pfarraydbak.h 文件中.
4 #include "pfarraydbak.h"
5 #include <iostream>
6 using std::cout;

7 PFArrayDBak::PFArrayDBak( ) : PFArrayD( ), usedB(0)
8 {
9     b = new double [getCapacity( )];
10 }
11 PFArrayDBak::PFArrayDBak(int capacityValue) : PFArrayD(capacityValue),
                                                usedB(0)
12 {
13     b = new double [getCapacity( )];
14 }
15 PFArrayDBak::PFArrayDBak(const PFArrayDBak& oldObject)
16     : PFArrayD(oldObject), usedB(0)
17 {
18     b = new double [getCapacity( )];
19     usedB = oldObject.usedB;
20     for (int i = 0; i < usedB; i++)
21         b[i] = oldObject.b[i];
22 }
23 void PFArrayDBak::backup( )
24 {
25     usedB = getNumberUsed( );
26     for (int i = 0; i < usedB; i++)
27         b[i] = operator[](i);
28 }
29
30 void PFArrayDBak::restore( )
31 {
32     emptyArray( );
33
34     for (int i = 0; i < usedB; i++)
35         addElement(b[i]);
36 }
37
38 PFArrayDBak& PFArrayDBak::operator =(const PFArrayDBak& rightSide)
39 {
40     int oldCapacity = getCapacity();

```

即使在基类的成员变量都被设置为私有型（而非保护型）的情况下，这种实现方式仍然能够正常工作。

当前调用对象对方括号运算符的调用。

```

39     PFArrayD::operator =(rightSide);
40     if (oldCapacity != rightSide.getCapacity( ))
41     {
42         delete [] b;
43         b = new double[rightSide.getCapacity( )];
44     }

45     usedB = rightSide.usedB;
46     for (int i = 0; i < usedB; i++)
47         b[i] = rightSide.b[i];

48     return *this ;
49 }

50 PFArrayDBak::~PFArrayDBak( )
51 {
52     delete [] b;
53 }

```

 提示：一个类可以访问本类所有对象的私有成员

考虑下面几行从示例 14.13 重载赋值运算符的实现中拷贝出来的代码：

```

usedB = rightSide.usedB;
for (int i = 0; i < usedB; i++)
    b[i] = rightSide.b[i];

```

你也许会认为 `rightSide.usedB` 和 `rightSide.b[i]` 是非法的，因为 `usedB` 和 `b` 是其他对象而非当前调用对象的私有成员变量。一般而言，这种看法是正确的。但是，因为对象 `rightSide` 与正在被定义的类是同一个类型的，所以这是合法的。

在类的定义中，你可以访问任何一个该类的对象的私有成员，而不仅仅局限于当前调用对象的私有成员。■

 提示：“是一个”和“有一个”

在本章的最开始部分，我们定义了一个派生类 `HourlyEmployee`，并且使用类 `Employee` 作为基类。在这个例子中，一个派生类 `HourlyEmployee` 的对象也是 `Employee` 类型的。简单来说，一个 `HourlyEmployee` 也是一个 `Employee`。这就是类与类之间“是一个”的关系。通过这种方法，可以在一个简单类的基础之上创建一个更为复杂的类。

另外一个在简单类的基础之上创建更为复杂类的方式是被称为“有一个”的关系。例如，如果你有一个类 `Date` 来记录日期，那么你可以通过在类 `Employee` 中添加一个 `Date` 类型的成员变量来为一个 `Employee` 类添加雇佣日期。在这种情况下，我们称一个员工“有一个”日期。再举一个例子，如果我们有一个被恰当命名的类来模拟喷气发动机，并且我们定义了一个类来模拟客机，那么我们给了 `PassengerAirPlane` 类一个或者多个 `JetEngine` 类型的成员变量。在这种情况下，我们称一个 `PassengerAirPlane` “有一个” `JetEngine`。

“是一个”  
的关系

“有一个”  
的关系

在大多数情况下，你既可以用“是一个”关系也可以用“有一个”关系来编写你自己的代码。如果把类 `PassengerAirPlane` 类作为 `JetEngine` 类的一个派生类，这看起来很愚蠢（并且确实很愚蠢）。幸运的是，最好的编程技巧就是简单地遵循英语中听起来很自然的方式。描述“一架客机有一个喷气发动机”要比“一架客机是一个喷气发动机”更合理。因此，把 `JetEngine` 作为类 `PassengerAirPlane` 的一个成员变量更符合编程逻辑，而将类 `PassengerAirPlane` 作为 `JetEngine` 的一个派生类是没有任何意义的。■

### 自测练习题

10. 假设你利用一个 `PFFArrayD` 类型的参数定义了一个函数，那么你是否能够以一个 `PFFArrayDBak` 类型的对象作为该函数的参数传入？
11. 将下面定义的成员函数加到类 `Employee`（示例 14.1）中是否合法？（记住，问题是它是否合法，而不是它是否有意义）。

```
void Employee::doStuff( )
{
    Employee object("Joe", "123-45-6789");
    cout << "Hello " << object.name << endl;
}
```

### 保护继承和私有继承

到目前为止，我们定义过的所有派生类在其定义的开始部分都包含了 `public` 关键字，如下列代码所示：

```
class SalariedEmployee : public Employee
{
```

这也许会误导你，使你猜想关键字 `public` 可以被 `protect` 或 `private` 所替换，从而可以获得一种不同的继承。在这个例子中，你的猜想是正确的，但是保护型和私有型继承很少被用到。为了完整起见，我们对它们进行概要介绍。

保护继承和私有继承的语法如下列代码所示：

```
class SalariedEmployee : protected Employee
{
```

如果你在继承中使用关键字 `protected`，那么基类中的公有成员被派生类继承后就成为保护型的。如果你在继承的过程中使用了 `private` 关键字，那么基类中的所有成员（无论是 `public`、`protected` 还是 `private`）在派生类中都是无法访问的。换句话说，所有继承而来的成员都像它们在基类中被标示为 `private` 一样。

此外，对于保护型继承和私有型继承而言，派生类的对象不可以当成参数传递给以基类为参数类型的函数。如果 `Derived` 类继承自 `Base` 类，并且使用了 `protected` 或者 `private`（而不是 `public`），那么一个 `Derived` 类型的对象就不是 `Base` 类型的了，因此“是一个”的关系无法适用于保护型继承和私有型继承。原因在于对于保护型继承和私有继承而言，基类只是一个用来定义派生类的工具。尽管

出于某些特殊的目的，保护型继承和私有型继承可以被使用，但是，正如你所猜测的那样，它们很少被用到。而且任何它们所能拥有的优点总可以通过其他方式获得。

关于保护型继承和私有型继承的细节将在示例 14.14 中列出。

示例 14.14 公有、保护和私有继承

基类中的访问限定符	继承类型（在派生类的定义中类名之后的限定符）		
	public	protected	private
public	public	protected	private (这能在成员函数或友元的定义中按名字访问)
protected	protected	protected	protected (这能在成员函数或友元的定义中按名字访问)
private	在派生类中无法按名字访问	在派生类中无法按名字访问	在派生类中无法按名字访问

示例 14.14 显示了派生类是如何处理继承而来的成员的。

注意保护型继承和私有型继承并不是通常我们描述公有型继承那种意义上的继承。对于保护型继承和私有继承而言，基类只不过是派生类要使用的一种工具。

多继承

对于派生类而言，有可能有多个基类。语法很简单：把所有的基类都列出来，用逗号分隔。但是模棱两可的可能性很多。如果两个基类有一个名字相同、传递的参数类型相同，结果会怎么样？哪一个会被派生类所继承？如果两个基类都有一个名字相同的成员变量会怎么样？尽管所有的问题都可以被回答，但是这些和其他的问题会使多继承成为一种危险的行为。有一些权威认为既然多继承有这么多风险，那么就完全不要使用多继承。这或许是一种很极端的观点，但是有一点是正确的：那就是除非你是一个经验特别丰富的 C++ 程序员，否则不要刻意地钻多继承的“牛角尖”。在这个问题上，你应该意识到几乎总是可以通过一些风险较小的技巧来避免使用多继承。我们准备在本书中过多地讨论多继承，而是把它留给更高级的参考书目。

## 本章小结

- 继承提供了一种代码重用的工具，它通过一个类从另外一个类派生类，并且在派生类中添加新特性。
- 派生类的对象继承了基类的所有成员，也可以添加新的成员。
- 如果基类的一个成员变量是私有的，那么在派生类中就无法通过名字访问到该私有成员变量。
- 私有成员函数不可以被继承。
- 一个成员函数可以在派生类中被重新定义，因此它能够表现出与在基类中表现不同的行为。被重新定义的函数声明即使与基类中的声明完全相同，也必须在派生类的定义中重新给出该函数的声明。
- 在公有派生类的成员函数定义中，可以通过名字访问基类中的保护成员。
- 重载的赋值运算符是不可以被继承的，但是在派生类重载的赋值运算符的定义中可以调用基类的赋值运算符。
- 构造函数是不可以被继承的，但是，基类的构造函数可以在派生类的构造函数的定义中被调用。

## 自测练习答案

1. 是的。你可以把一个派生类的对象传递给一个基类类型的参数。一个 `HourlyEmployee` 是一个 `Employee`，一个 `SalariedEmployee` 也是一个 `Employee`。
2. 

```
class SmartBut :public Smart
{
public :
    SmartBut( );
    SmartBut(int newA, int newB, bool newCrazy);
    bool isCrazy( ) const ;
private :
    bool crazy;
};
```
3. 这是合法的，因为 `a` 和 `b` 在基类 `Smart` 中被标注为 `protected` 的，所以它们在派生类中可以通过名字被访问到。相反，如果 `a` 和 `b` 被标注为 `private` 的，那么就是非法的。
4. 函数 `getName` 的声明并没有在 `SalariedEmployee` 的定义中给出，因为它在 `SalariedEmployee` 类中没有被重新定义。它是原封不动地从基类 `Employee` 中继承而来的。
5. 

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



```

using namespace std;
namespace SavitchEmployees
{
    class TitledEmployee : public SalariedEmployee
    {
    public:
        TitledEmployee( );
        TitledEmployee(string theName, string theTitle,
                        string theSsn, double theSalary);
        string getTitle( ) const ;
        void setTitle(string theTitle);
        void setName(string theName);
    private:
        string title;
    };
} //SavitchEmployees

```

#### 6. namespace SavitchEmployees

```

{
    TitledEmployee::TitledEmployee( )
        : SalariedEmployee( ), title("No title yet")
    {
        // 函数体刻意为空。
    }

    TitledEmployee::TitledEmployee(string theName,
                                    string theTitle,
                                    string theSsn, double theSalary)
        : SalariedEmployee(theName, theSsn, theSalary),
          title(theTitle)
    {
        // 函数体刻意为空。
    }

    void TitledEmployee::setName(string theName)
    {
        Employee::setName(title + theName);
    }
} //SavitchEmployees

```

7. 否。如果你不为派生类重新定义一个重载的赋值运算符或者拷贝构造函数，那么一个默认的赋值运算符或者一个默认的拷贝构造函数将会被派生类自动创建。但是，如果这个类包含指针、动态数组或者其他动态数据，那么几乎可以确定的是，既不会有默认的赋值运算符也不会有默认的拷贝构造函数像你期望的那样运行。
8. 构造函数将会如下列顺序被调用：首先是 Parent 的，然后是 Child 的，最后是 GrandChild 的。析构造函数将会以相反的顺序被调用：首先是 GrandChild 的，然后是 Child 的，最后是 Parent 的。
9. 是的，这是合法的，并且与示例 14.11 所给出的定义含义相同。如果没有基类的构造函数被调用，那么基类默认的构造函数将会被自动调用。
10. 是的。一个派生类的对象也是这个派生类的基类的对象。一个 PFArrayDBak

是一个 PFArrayD。

11. 是的，这是合法的。你认为它是合法的原因之一是 name 是一个私有的成员变量。但是，object 在类 Employee 中，而这个类是当前正在被定义的类，所以我们访问类 Employee 的所有对象的所有成员变量。

## 编程练习

1. 用示例 14.4 所给出的类 SalariedEmployee 写一段程序。这段程序需要定义一个叫作 Administrator 的派生类，这个派生类将继承自类 SalariedEmployee。你可以将基类中的限定符由 private 改为 protected。你将要提供下列新增的数据和函数成员：
  - 一个 string 类型的成员变量，这个变量将用于记录管理者的称谓，如董事、副经理。
  - 一个类型为 string 的成员变量，它用于表示公司的职权范围，如生产、会计和人事。
  - 一个类型为 string 的成员变量，它用于该管理者的直接上司的名字。
  - 一个保护型的类型为 double 的成员变量用于表示管理者的年薪。如果你将基类中的年薪变量由 private 的变为 protected 的，你也可以使用这个已有的变量。
  - 一个叫作 setSupervisor 的成员函数，用于设置上司的名字。
  - 一个成员函数用于读取从键盘输入的管理者数据。
  - 一个叫作 print 的函数，用于将对象的数据输出到屏幕。
  - 最后，重载成员函数 printCheck，使之在支票上有适当的标识。
2. 给示例 14.1、示例 14.3 和示例 14.4 所体现的继承关系添加新的临时雇员、管理者雇员、永久雇员和其他雇员分类。实现并且测试这些新的分类。测试所有的成员函数。在测试程序中，如果有带有菜单的用户界面，将会收到良好的效果。
3. 给出一个名字为 Doctor 的类的定义，这个类的对象用于记录诊所的医生。这个类将是示例 14.4 中给出的 SalariedEmployee 类的派生类。一个 Doctor 记录包含医生的专业（如“儿科医师”、“产科医师”、“全科医师”等，因此类型为 string）和出诊费（类型为 double）。确保你的类有合理实现的构造函数和访问方法、重载的赋值运算符和拷贝构造函数。编写一个驱动程序来测试所有的函数。
4. 创建一个叫作 Vehicle 的基类，这个类拥有制造者的名字（类型为 string）、发动机的气缸数目（类型为 int）以及它的所有者（类型为 Person，其定义在下列代码中给出）。然后创建一个叫作 Truck 的类，它继承于 Vehicle 类，

并且拥有新增的属性，以吨为单位的载重容量（既然它可能有小数部分，那么类型为 double）、以磅为单位的牵引力（类型为 int）。确保你的类有合理实现的构造函数和访问方法、重载的赋值运算符和拷贝构造函数。编写一个驱动程序来测试所有的函数。

类 Person 的定义如下，关于这个类的实现是本题编程的一部分。

```
class Person
{
public:
    Person( );
    Person(string theName);
    Person(const Person& theObject);
    string getName( ) const;
    Person& operator =(const Person& rtSide);
    friend istream& operator >>(istream& inStream,
                                Person& personObject);
    friend ostream& operator <<(ostream& outStream,
                                const Person& personObject);

private:
    string name;
};
```

5. 请给出 Patient 和 Billing 两个类的定义，这两个类的对象是关于一个诊所的记录。Patient 继承自编程练习 4 给出的 Person 类。一个 Patient 记录拥有病人的名字（继承自类 Person）、主治医师（是 Doctor 类型的，Doctor 类在编程练习 3 中给出）。一个 Billing 对象将包含一个 Patient 对象和一个 Doctor 对象，还有一个类型为 double 的费用总额。确保你的类有合理实现的构造函数和访问方法、重载的赋值运算符和拷贝构造函数。首先编写一个驱动程序来测试所有的函数，然后写一个测试程序用于创建至少两个病人、至少两个医生和至少两个账单，最后打印出账单的全部收入。
6. 定义一个 Payment 类，它包含了一个 float 类型的成员变量用于保存付款额，并且包含了适当的取值函数和赋值函数。同时创建了一个函数名为 paymentDetails 的成员函数，用于输出用英文描述的付款额。

接下来定义一个名为 CashPayment 的类，它继承自 Payment 类。这个类应当重定义函数 paymentDetails，表明是现金付款。同时创建适当的构造函数。

定义一个叫作 CreditCardPayment 的类，它继承自 Payment 类。这个类应该包含持卡人的姓名、卡的有效期限及信用卡号相关的成员变量。包含适当的构造函数。最后，重新定义 paymentDetails 函数，从而在打印输出中包含所有信用卡信息。

创建 main 函数，在此创建至少两个 CashPayment 对象和两个 CreditCardPayment 对象，这些对象拥有不同的值，并且为每个对象调用 paymentDetails 函数。

7. 定义一个名为 `Document` 的类，这个类包含一个 `string` 类型的成员变量，叫作 `text`，用于存储文档的所有文本内容。创建一个名为 `getText` 的函数，用于返回文本域，再创建一个赋值函数和一个重载的赋值运算符。

接下来定义一个 `Email` 类，它继承自类 `Document`，并且拥有如下成员变量：`sender`、`recipient` 和电子邮件信息的 `title`。实现适当的取值函数和赋值函数。电子邮件信息的文本内容应当存储在继承来的变量 `text` 中，并且包含一个重载的赋值运算符。

与此类似，定义一个名为 `File` 的类，它也继承自 `Document` 类。包含一个表示文件路径的成员变量，为这个变量实现适当的取值函数、赋值函数和重载的赋值运算符。

最后，在你的 `main` 函数中创建几个类型为 `Email` 和 `File` 的样例对象，将这些对象传递给如下子程序进行测试，如果对象包含文本属性中指定的关键字，子程序返回 `true`。

```
bool ContainsKeyword(const Document& docObject, string keyword)
{
    if (docObject.getText().find(keyword) != string::npos)
        return true ;
    return false ;
}
```

例如，通过调用 `ContainsKeyword(emailObj, "c++")`，你可以测试一个电子邮件信息是否包含文本“c++”。

8. 创建一个简单的博客类，博主应当具有以下权限：(a) 发送一条消息，(b) 按序号排列显示所有的消息，(c) 选择一个指定的消息并且删除它。

博客的读者应该只能按序号排列并显示所有的已发出消息。

创建一个 `Viewer` 类和一个 `Owner` 类，利用继承来帮助实现博客的功能。可以使用任意你喜欢的格式来存储消息数据（最简单的是使用一个 `string` 类型的向量）。应当为每一个类实现一个菜单函数，用于输出不同类型的用户都进行哪些合法的操作。为了测试你的类，程序的 `main` 函数应该允许用户从 `Viewer` 和 `Owner` 对象激活菜单。

9. 假设你正在创建一个幻想型角色扮演游戏。在这个游戏中，我们有四个不同种族的生物：人类、数码恶魔、蝙蝠恶魔和精灵。为了表示这些生物中的一种，我们需要像下面这样定义一个 `Creature` 类。

```
class Creature
{
private :
    int type; // 0 人类, 1 数码恶魔, 2 蝙蝠恶魔, 3 精灵生物
    int strength; // 我们能够发挥出多大的打击力
    int hitpoints; // 我们能够承受多大的打击力
    string getSpecies(); // 返回种族的类型
}
```

```

public :
    Creature( );
    // 初始化人类, 10 点力量, 10 点打击

    Creature(int newType, int newStrength, int newHit);
    // 初始化生物的种族、力量、打击点数

    // 同时为 type、strength 和 hitpoints 创建适当的取值函数和赋值函数

    int getDamage();
    // 返回这个生物在一场战斗中所发挥出的打击伤害
};

```

这里有一个 `getSpecies()` 函数的实现。

```

string Creature::getSpecies()
{
    switch (type)
    {
        case 0: return "Human";
        case 1: return "Cyberdemon";
        case 2: return "Balrog";
        case 3: return "Elf";
    }
    return "Unknown";
}

```

`getDamage()` 函数输出并且返回了这个生物在一场战斗中所能造成的伤害。计算伤害值的规则如下：

- 每一个生物所造成的伤害值为一个随机数  $r$ ,  $0 < r \leq \text{strength}$ 。
- 恶魔有 5% 的概率发出恶魔打击, 魔法打击将会造成 50 点的额外伤害。数码恶魔和蝙蝠恶魔都属于恶魔。
- 精灵有 10% 的概率发出魔法打击, 魔法打击将会造成双倍的普通伤害。
- 蝙蝠恶魔非常敏捷, 所以它们有机会发出两次打击。

`getDamage()` 的一个实现如下：

```

int Creature::getDamage()
{
    int damage;

    // 所有生物造成的伤害都是一个小于它们力量值的随机数
    damage = (rand() % strength) + 1;
    cout << getSpecies() << " attacks for " <<
        damage << " points!" << endl;

    // 恶魔拥有 5% 的概率造成 50 点的额外伤害
    if ((type == 2) || (type == 1))
        if ((rand() % 100) < 5)
        {
            damage = damage + 50;
            cout << "Demonic attack inflicts 50 "

```

```

        << " additional damage points!" << endl;
    }

    // 精灵拥有 10% 的概率造成双倍的伤害
    if (type == 3)
    {
        if ((rand() % 10) == 0)
        {
            cout << "Magical attack inflicts " << damage <<
                " additional damage points!" << endl;
            damage = damage * 2;
        }
    }

    // 蝙蝠恶魔速度很快, 所以它们有机会发出两次打击
    if (type == 2)
    {
        int damage2 = (rand() % strength) + 1;
        cout << "Balrog speed attack inflicts " << damage2 <<
            " additional damage points!" << endl;
        damage = damage + damage2;
    }
    return damage;
}

```

这样实现的一个问题是不便于添加新的生物。重写这个类从而能够使用继承, 去掉变量 `type`。重写之后, `Creature` 类是基类, 类 `Demon`、`Elf` 和 `Human` 应当继承自 `Creature` 类。类 `Cyberdemon` 和 `Balrog` 应当继承自 `Demon` 类。你需要重写函数 `getSpecies()` 和函数 `getDamage()`, 从而使它们能够适应每个类。

例如, 每个类的 `getDamage()` 函数应当只计算该类所对应的对象的伤害数。总的伤害应当通过综合每一级继承层次的 `getDamage()` 的结果来计算。例如, 触发一个 `Balrog` 对象的 `getDamage()` 函数就应当也触发一个 `Demon` 对象的 `getDamage()` 函数, 而触发了 `Demon` 对象的 `getDamage()` 函数也会触发 `Creature` 对象的 `getDamage()` 函数。这将计算所有生物所造成的基本伤害, 外加恶魔造成的 5% 的随机伤害, 以及蝙蝠恶魔造成的双倍普通伤害。为私有变量创建适当的取值函数和赋值函数。写一个 `main` 函数来测试你所写的类。应当为每一种生物创建一个对象并且重复地输出 `getDamage()` 的结果。

10. 定义一个 `Pet` 类用于保存宠物的名字、年龄和体重, 添加适当的构造函数、取值函数和赋值函数。并且定义一个叫作 `getLifespan` 的函数用于返回一个字符串 “unknown lifespan”。

接下来定义一个 `Dog` 类继承自 `Pet` 类。 `Dog` 类应当有一个私有的成员变量 `breed` 用于保存狗的品种。为成员变量 `breed` 添加适当的取值函数和赋值函数, 并且添加适当的构造函数。重新定义 `getLifespan` 函数用于返回狗的寿命, 如果狗的体重超过 100 磅, 则返回 “大约七年”; 如果狗的体重低于 100 磅, 则

返回“大约 13 年”。

然后定义一个 `Rock` 类也继承自 `Pet` 类。重定义 `getLifespan` 函数用于返回“成千上万年”。

最后，写一个测试程序，分别为宠物石头和宠物狗创建一个实例来练习继承函数和重写函数。

## 15.1 虚函数基础 556

延迟绑定 556

C++中的虚函数 557

提示：virtual属性会被继承 563

提示：什么时候应该使用虚函数 563

陷阱：没有对虚成员函数进行定义 564

抽象类与纯虚函数 564

示例：抽象类 565

## 15.2 指针和虚函数 567

虚函数与扩展类型兼容性 567

陷阱：切片问题 571

提示：使析构函数成为虚函数 572

向下类型转换和向上类型转换 573

C++如何实现虚函数 574



# 第 15 章 多态与虚函数

我走我的路。

弗兰克·辛纳屈

## 概述

多态是指通过一种被称为虚函数或者延迟绑定的特殊机制将多个实现方式关联到一个函数名的能力。多态是面向对象编程思想的基本特征 (mechanism) 之一。本章将对这一概念做详细介绍。

15.1 节的学习不需要第 10 章 (指针和动态数组)、第 12 章 (文件 I/O) 及第 13 章 (递归) 的相关知识,但需要第 14 章 (继承) 的相关知识。15.2 节不需要第 12 章 (文件 I/O) 及第 13 章 (递归) 的相关知识,但会用到第 10 章 (指针和动态数组) 的相关知识。

## 15.1 虚函数基础

**virtual** 是形容词,指虽然没有实际的事实、形式或名义,但在实际上或效果上是存在的或产生的。

《美国传统英语字典 (第三版)》

虚函数之所以被称为虚函数,是因为我们可能会在定义它之前使用它。通过本章的学习,读者将会对虚函数有更清楚的理解。此外,虚函数也是软件重用的另一个工具。

### 延迟绑定

我们通过一个具体的例子来对虚函数进行说明。假设你要设计一个图形包软件,此软件包含一些表示各种图形的类,诸如矩形、圆、椭圆等。每一个图形可能表示的是一个类的具体对象。比如,Rectangle 这个类具有高、宽、中心点这些成员变量,而 Circle 这个类可能包含中心点和半径两个成员变量。在一个设计良好的编程项目中,所有这些类可能有同一个父类,比如说上面提到的这些类的父类可能是一个被称为 Figure 的类,这些类都是 Figure 类的派生类。假设需要一个函数在屏幕上画一个图,如果希望画的是圆,那么它的实现指令必然和矩形的实现指令不一样。因此,每一个派生类需要一个不同的函数来画对应的图。但是,由于这些函数分别属于它们各自的类,因此,可以把它们都命名为 draw。如果 r 是类 Rectangle 的对象, c 是类 Circle 的对象,那么函数 r.draw() 和 c.draw() 的实现代码是不同的。上述这些内容对我们来说并不是新知识。接下来,我们将要学习一些新的知识:在父类 Figure 中定义虚函数。

我们可能会在父类 Figure 中定义一些函数,这些函数可以对各种图进行一些操

作。比如，可能会在 `Figure` 中定义一个 `center` 函数，该函数会先把原有的图形擦除掉，然后在屏幕的中央重新绘制一个和原图一样的图，以这种方式来把图移动到屏幕的中央。函数 `Figure::center` 可能会调用函数 `draw` 来在屏幕中心重新绘图。当我们考虑使用继承于类 `Rectangle` 和类 `Circle` 中的函数 `center` 时，就会开始发现问题变得有些复杂了。

为了把这个问题解释清楚，我们让问题暴露得更明显一些。通过具体例子来看这个问题。假设我们已经写好了 `Figure` 类，并且已经在使用这个类了。随后我们为一个全新的图形增加了一个新的类，比如说，这个新增的类是 `Triangle`。现在，`Triangle` 可以作为类 `Figure` 的派生类，与此同时，函数 `center` 也将从类 `Figure` 继承而来。为此，函数 `center` 应当适用于（能正确执行）所有的 `Triangle` 对象。但是这里存在一个问题。函数 `center` 调用了函数 `draw`，对于不同的图形，函数 `draw` 的实现方式是不同的。如果未做任何特殊处理，从 `Figure` 类中继承下来的函数 `center` 将调用类 `Figure` 中定义的函数 `draw`，但这个 `draw` 函数并不能为 `Triangle` 对象正确地绘制出三角形。我们期望被继承下来的成员函数 `center` 调用函数 `Triangle::draw` 而不是调用函数 `Figure::draw`。但是，当我们在编写和编译类 `Figure` 中定义的函数 `center` 时，我们甚至还没开始编写类 `Triangle` 及其函数 `Triangle::draw`。因此，函数 `center` 怎么可能正确地被 `Triangle` 对象所使用？当函数 `center` 被编译时，编译器对 `Triangle::draw` 函数一无所知！对于这一问题，解决的办法是，将函数 `draw` 声明为虚函数。

虚函数  
动态绑定  
延迟绑定

把一个函数声明为虚函数，就是告诉编译器“我不知道这个函数是如何实现的。等到这个函数在程序中被使用时，才从对象实例中获取它的具体实现”。这种等到运行时才决定具体实现方式的技术就是通常所称的延迟绑定或动态绑定。虚函数是 C++ 提供延迟绑定的一种技术手段。我们已经对此做了详尽的介绍，除此之外，我们还需要给出一个例子来更好地理解虚函数，并教会大家如何在程序中使用虚函数。为了把 C++ 中虚函数的细节讲解得更为清楚，我们将不再使用画图这个例子，而是使用一个来自于某一应用领域的简单例子进行说明。

## C++中的虚函数

假设你正在为一个汽车零件商店设计一个记录维护的程序。你希望这个程序具有很好的通用性，但你不确定你能考虑到所有可能出现的情况。比如，你想跟踪销售情况，但却无法预见各种销售类型。最初，只有常规销售，面向那些去店里买某个零件的零售客户。但是，后来你可能想增加打折销售或者需要付运费的邮购销售方式。所有这些销售方式都是面向某一特定商品的，该商品有一个基本的价格，而且该商品最终产生了一些账单。对于一种简单销售类型来说，账单仅仅是一个基本价格，但是如果后面增加了打折，那么这些账单类型也将依赖于打折的程度而定。所编写的程序需要计算每日的毛收入，直观地看，毛收入应该正好就是所有销售账单之和。也许你还想计算出每天的最大、最小销售额及平均销售额。所有这些都可以通过对各个账单的统计分析而得到。但是，在程序设计之初，并不会把这些计算账单的函数写到程序中

去,直到已经决定了采取哪种销售类型之后才把它们添加到程序中。为了应对这种情况,我们把所有计算账单的函数都定义为虚函数。(为了简单起见,在第一个例子中我们假设每一次销售只是面向某一个商品。事实上,通过使用派生类和虚函数,我们也可以对多个商品的销售问题进行说明,但是在这里暂不考虑多个商品销售的情况。)

示例 15.1 是一个表示销售的基类 Sale 的 C++ 接口程序,示例 15.2 是基类 Sale 的 C++ 实现程序。所有销售类型都是基类 Sale 的派生类。基类 Sale 表示的是一个商品销售的简单类,它没有打折及运费等项。请注意,在示例 15.1 中的成员函数 bill 前的 C++ 保留字 virtual,以及在示例 15.2 中的成员函数 savings 和重载运算符“<”,它们都使用了 bill 函数。由于 bill 被声明的是虚函数,因此,我们才能在之后定义类 Sale 的派生类及各派生类自己的 bill 函数。各派生类在继承基类 Sale 后,它们也继承了基类中的成员函数 savings 及重载符“<”,派生类中的成员函数 savings 以及重载运算符“<”在调用成员函数 bill 时,它们调用的是派生类中所定义的 bill 函数。

### 示例 15.1 基类 Sale 的接口

---

```

1
2 // 这是头文件 sale.h.
3 // 这是类 Sale 的接口。
4 //Sale 表示的是一个简单的销售类。

5 #ifndef SALE_H
6 #define SALE_H

7 namespace SavitchSale
8 {
9     class Sale
10     {
11     public :
12         Sale( );
13         Sale(double thePrice);
14         double getPrice( ) const ;
15         void setPrice( double newPrice);
16         virtual double bill( ) const ;
17         double savings(const Sale& other) const ;
18         // 如果买其他商品比买这件商品更省钱, 返回节省的余额。
19     private :
20         double price;
21     };

22     bool operator < (const Sale& first, const Sale& second);
23     // 比较两个销售类, 看看哪个销售额更大。
24 } //SavitchSale

25 #endif // SALE_H

```

---

## 示例 15.2 基类 Sale 的实现

---

```

1
2 // 这是文件 sale.cpp.
3 // 这是类 Sale 的实现。
4 // 类 Sale 的接口在文件 sale.h 中。

5 #include <iostream>
6 #include "sale.h"
7 using std::cout;

8 namespace SavitchSale
9 {

10     Sale::Sale( ) : price(0)
11     {
12         // 特意设计为空。
13     }

14     Sale::Sale(double thePrice)
15     {
16         if (thePrice >= 0)
17             price = thePrice;
18         else
19         {
20             cout << "Error: Cannot have a negative price!\n";
21             exit(1);
22         }
23     }

24     double Sale::bill( ) const
25     {
26         return price;
27     }

28     double Sale::getPrice( ) const
29     {
30         return price;
31     }

32     void Sale::setPrice(double newPrice)
33     {
34         {
35             if (newPrice >= 0)
36                 price = newPrice;
37             else
38             {
39                 cout << "Error: Cannot have a negative price!\n";
40                 exit(1);
41             }
42         }

43     double Sale::savings(const Sale&other) const
44     {
45         return (bill( ) - other.bill( ));
46     }

```

```

47     bool operator < (const Sale& first, const Sale& second)
48     {
49         return (first.bill() < second.bill());
50     }
51 } //SavitchSale

```

---

例如，示例 15.3 和示例 15.4 分别定义了派生类 DiscountSale 的接口和实现。注意，类 DiscountSale 需要定义一个不同于基类的 bill 函数，它需要为自己定义一个特定的 bill 函数。当类 DiscountSale 的对象调用成员函数 savings 和重载运算符“<”时，成员函数 savings 和重载运算符“<”将会调用 DiscountSale 所定义的 bill 函数。这的确是 C++ 所实现的一个非常奇特的技巧。我们再来看一下类 DiscountSale 的两个对象 d1 和 d2 的函数调用 d1.savings(d2)。基类中的函数 savings，甚至包括派生类 DiscountSale 的对象所使用的 savings 函数都是在基类 Sale 的实现文件中定义的，这个文件可能在我们能够考虑到需要设计类 DiscountSale 之前就已经被编译好了。但是，通过使用虚函数，在函数调用 d1.savings(d2) 中，调用 bill 函数的这条语句有足够的信息知道应该去调用派生类 DiscountSale 中定义的 bill 函数。

这种机制是如何工作的呢？如果我们只想写 C++ 程序而不关注其背后的机理，我们可以只把它看作一种很神奇的机制，但实际上，在本章开始的概述中我们已经向读者解释了这种机制。即：当我们把一个函数标记为 virtual 时，我们相当于告诉了 C++ 语言“等到函数被程序调用时，再去获得对应于调用对象的该函数的具体实现”。

示例 15.5 是一个完整的样例程序，向我们展示了虚函数 bill 及调用 bill 的那些函数是如何工作的。

### 示例 15.3 派生类 DiscountSale 的接口

---

```

1
2 // 这是文件 discountsale.h。
3 // 这是类 DiscountSale 的接口。

4 #ifndef DISCOUNTSALE_H
5 #define DISCOUNTSALE_H
6 #include "sale.h"

7 namespace SavitchSale
8 {

9     class DiscountSale : public Sale
10    {

11    public :
12        DiscountSale();
13        DiscountSale(double thePrice, double theDiscount);
14        // 折扣是用价格的百分比来表示的。
15        // 负的折扣表示的是价格的提升。
16        double getDiscount() const;
17        void setDiscount(double newDiscount);

```

```

18     double bill( ) const;
19     private :
20         double discount;
21     };
22 } // SavitchSale
23 #endif // DISCOUNTSALE_H

```

由于在基类中，我们把 `bill` 声明为虚函数，因此，在派生类中它也自动表示一个虚函数。在派生类中，可以通过标识符 `virtual` 来说明它是虚函数，也可以不加标识符 `virtual`。不管哪种方式，派生类 `DiscountSale` 中的函数 `bill` 都表示的是一个虚函数。（虽然在某些情况下可以不加标识符 `virtual`，但我们还是建议在所有虚函数的声明前加上标识符 `virtual`。在此处我们没加 `virtual`，是因为我们想告诉读者不加是可以的。）

#### 示例 15.4 派生类 `DiscountSale` 的实现

```

1
2 // 这是类 DiscountSale 的实现。
3 // 这是文件 discountsale.cpp。
4 // 类 DiscountSale 的接口在
5 // 头文件 discountsale.h 中。
6 #include "discountsale.h"

7 namespace SavitchSale
8 {
9     DiscountSale::DiscountSale( ) : Sale( ), discount(0)
10    {
11        // 特意设计为空。
12    }

13    DiscountSale::DiscountSale(double thePrice, double theDiscount)
14        : Sale(thePrice), discount(theDiscount)
15    {
16        // 特意设计为空。
17    }

18    double DiscountSale::getDiscount( ) const
19    {
20        return discount;
21    }

22    void DiscountSale::setDiscount(double newDiscount)
23    {
24        discount = newDiscount;
25    }

26    double DiscountSale::bill( ) const
27    {
28        double fraction = discount / 100;
29        return (1 - fraction) * getPrice( );
30    }
31 } // SavitchSale

```

在这个函数的定义部分，不需要重复出现限定符 `virtual`。

## 示例 15.5 虚函数的用法

```

1  // 演示虚函数 bill 的性能。
2  #include <iostream>
3  #include "sale.h" // 实际上不需要，但由于有 ifndef，引入后会更安全。
4  #include "discountsale.h"
5  using std::cout;
6  using std::endl;
7  using std::ios;
8  using namespace SavitchSale;

10 int main( )
11 {
12     Sale simple(10.00); // 每件商品售价 $10.00。
13     DiscountSale discount(11.00, 10);
        // 售价 $11.00 的商品，可以有 10% 的折扣。

14     cout.setf(ios::fixed);
15     cout.setf(ios::showpoint);
16     cout.precision(2);

17     if (discount < simple)
18     {
19         cout << "Discounted item is cheaper.\n";
20         cout << "Savings is $" << simple.savings(discount) << endl;
21     }
22     else
23         cout << "Discounted item is not cheaper.\n";
24     return 0;
25 }

```

对象 discount 和对象 simple 的成员函数 bill 的实现代码是不同的。同样，它们的成员函数 savings 的实现代码也是不同的。

## 示例运行结果

```

Discounted item is cheaper.
Savings is $0.10

```

**虚函数**

我们通过在成员函数前加上修饰符 virtual 来将成员函数标识为虚函数，虚函数是在类的定义中给出的。

如果一个函数是虚函数，并且在其派生类中对该函数进行了新的定义，那么该派生类的任何对象将总会调用该派生类中所定义的虚函数，即使另外一个间接调用该函数的继承函数也会去调用派生类中所定义的虚函数。这种可以选择使用哪个虚函数的方式被称为延迟。

## 多态

多态是指通过所谓的延迟绑定机制将多个实现方式关联到一个函数名的能力。因此，多态、延迟以及虚函数实际上涉及的都是同一主题。

## 覆盖

当在一个派生类中改变了基类中虚函数的定义，程序员们称此虚函数的定义被覆盖了。在C++文献中，术语重定义和覆盖是有区别的。两者都是指在派生类中改变了函数的定义。如果这个函数是虚函数，则改变函数的这种行为被称为覆盖。如果该函数不是虚函数，则这种行为被称为重定义。对于程序员来说，这种区别似乎显得有点愚蠢，因为两种情况下做的事都是一样的；但对于编译器来说，两种情况的处理方式是不同的。

提示：virtual 属性会被继承

虚函数的 virtual 属性是会被继承的。比如，由于在基类 Sale（示例 15.1）中声明 bill 为虚函数，在派生类中函数 bill 自动继承为虚函数（示例 15.3），因此，在派生类 DiscountSale 中以下两个关于成员函数 bill 的定义是等价的：

```
double bill() const;
virtual double bill() const;
```

因此，如果类 SuperDiscountSale 是类 DiscountSale 的派生类，而类 DiscountSale 继承了基类中的函数 savings，并且如果在类 SuperDiscountSale 中对 bill 函数进行了重新定义，那么类 SuperDiscountSale 的所有对象都将使用在该类中所定义的 bill 函数。任何时候，只要是类 SuperDiscountSale 中的调用对象，即使是该类中调用 bill 函数的继承函数 savings，都会去调用 SuperDiscountSale 中所定义的 bill 函数。■

提示：什么时候应该使用虚函数

使用虚函数的好处显而易见，但目前为止，我们还没看到使用虚函数有什么明显的弊端。因此，我们何不把所有函数都声明为虚函数？更有甚者，为何不像其他一些编程语言，比如 Java 那样，把 C++ 编译器设计为可以自动将所有函数转为虚函数的编译器？之所以没有这么做的原因是，把函数定义为虚函数是有开销的。这样做不仅会占用更多的存储资源；同时，相对于没有定义函数为虚函数的情况，程序的运行会更慢。这就是 C++ 语言的设计者要将是否定义函数为虚函数的权利交给程序员自己决定的原因。如果你希望利用把成员函数定义为虚函数的好处，那便将成员函数尽可能定义为虚函数。如果你不希望利用虚函数的一些优势，不把成员函数定义为虚函数，那么，你的程序将会运行得更加高效。■



### 目测练习题

1. 请解释术语虚函数、延迟绑定和多态三者间的区别。
2. 假设我们对示例 15.1 中的类 `Sale` 进行了修改，删除了保留字 `virtual`。请问对于示例 15.5 的输出结果将会产生什么影响？

### 陷阱：没有对虚成员函数进行定义

增量式开发是一种明智的开发模式。它的特点是，先编写一部分代码，然后对所编写的这部分代码进行测试。接着再编写一部分代码，然后继续测试这部分代码，按这种方式继续下去，直到开发结束。但是，如果在编译一个包含虚成员函数的类时，还没有实现每一个成员函数，那么，我们将会碰到编译器所抛出的一些非常难理解的错误信息，即使我们并没有调用这些未实现的成员函数！

如果某一个虚成员函数在编译前没有被实现，那么编译器将无法成功编译，并抛出类似于下面的错误信息：

```
Undefined reference to Class_Name virtual table.
```

即使基类并没有派生类，它只有一个虚成员函数，但没有实现这个成员函数，上述错误信息依然会被抛出。

如果没有对声明为 `virtual` 的函数进行定义，会让编译器抛出难以理解的错误。它甚至会进一步抛出奇怪的错误消息，提示我们引用了未定义的默认构造函数，但实际上我们已经定义了这些构造函数。

当然，在定义一个“真正”的虚函数前，可以先对虚函数做一些简单的定义，以避免编译问题。

对于纯虚函数，我们就不需要这么小心了。我们将在下一节讨论纯虚函数的知识。届时我们将会看到对于纯虚函数而言，我们不应该对其进行定义。■

### 抽象类与纯虚函数

在实际编程中，我们可能会遇到这种情形，需要一个类作为其他一些类的基类，但并不想对该类的某个或某些成员函数做任何有意义的定义。在介绍虚函数时，我们讨论过这种情况。现在我们来回顾一下。

假设我们正在设计一个图形软件包，这个软件包中包含几个表示图形的类，比如矩形、圆形和椭圆形等。每一个图都可能是某个类（比如 `Rectangle` 类和 `Circle` 类）的对象。在一个设计良好的编程项目中，所有这些类可能都是一个父类的子类，比如我们称这个父类为 `Figure`。现在，假设我们想编写一个在屏幕上画图的函数。那么，我们画一个圆所需要的代码和想画一个矩形所需要的代码是完全不同的。因此，每一种类都需要一个各自不同的函数来画出它所对应的图形。如果 `r` 是一个 `Rectangle` 对象，`c` 是一个 `Circle` 对象，那么函数 `r.draw()` 和函数 `c.draw()` 的实现代码是不同的。

父类 `Figure` 可能包含一个叫作 `center` 的函数，它的功能是把一个图移动

到屏幕的中心位置。它会先删除原图，然后在屏幕中心位置处重新绘制原图。函数 `Figure::center` 可能会使用函数 `draw` 来在屏幕中心位置处重新绘制图形。通过把成员函数 `draw` 声明为虚函数，我们就可以为类 `Figure` 编写成员函数 `Figure::center` 的代码，并可以知道派生类，比如 `Circle`，什么时候调用这个函数——类 `Circle` 中 `draw` 函数的定义将是 `Circle` 对象真正使用到的函数定义。我们可能从未想过去创建一个 `Figure` 类型的对象。我们只想创建派生类对象，比如类 `Circle` 和 `Rectangle` 的对象。因此，我们在 `Figure` 类中定义的函数 `Figure::draw` 绝不会被其他函数所调用。但是，基于我们目前所学到的内容，必须给出 `Figure::draw` 的定义（尽管它没有什么实际意义）。

## 纯虚函数

如果你把成员函数 `Figure::draw` 设计成一个纯虚函数，那么你就不需要对该成员函数进行任何定义了。让成员函数成为纯虚函数的方法是把它标记为 `virtual`，并且在成员函数的声明之后添加 “= 0” 的符号，如下例所示：

```
virtual void draw() = 0;
```

任何成员函数都可以成为纯虚函数，但并不要求一定要像上面的例子所示的那样：返回类型为 `void` 且不带输入参数。

## 抽象类

具有一个或多个纯虚函数的类被称为抽象类。抽象类仅仅被用来作为基类来派生出其他类。不可以为抽象类创建对象，因为它不是一个完整定义的类。抽象类是一个部分定义的类，因为它包含其他非纯虚函数的成员函数。抽象类也是一种数据类型，因此你可以编写以抽象类作为参数类型的代码，抽象类参数类型可以应用于所有那些由抽象类的派生类所创建的对象。

如果你从抽象类派生了一个类，派生出来的类本身也将是一个抽象类（除非你对派生类中所有继承而来的纯虚函数进行了定义，同时也没有引入任何新的纯虚函数）。如果你所创建的继承类中，确实为所有继承而来的纯虚函数进行了定义，同时也没有引入任何新的纯虚函数，那么该继承类不是一个抽象类，你可以创建该类的对象。

---

## 示例：抽象类

在示例 15.6 中，我们对示例 14.1 中的类 `Employee` 进行了少许改动，重写了该类。在该例中我们将类 `Employee` 设计成了一个抽象类。示例 15.6 中突出强调的那一行是与前面在示例 14.1 中所定义类 `Employee` 唯一有区别的地方：

```
virtual void printCheck() const = 0;
```

在成员函数声明中的保留字 `virtual` 及符号 “= 0” 告诉编译器这是一个纯虚函数，因此类 `Employee` 是一个抽象类。类 `Employee` 在本例的实现中，没有包含任何 `Employee::printCheck` 的定义。否则，本例的类 `Employee` 的实现就和示例 14.2 完全一样了。

对成员函数 `Employee::printCheck` 不做任何定义是合理的，因为你不知道所要开的支票针对哪种类型的员工，所以你不知道该开哪种类型的支票。在

最初关于类 `Employee` 的定义中（参见示例 14.1 和示例 14.2），我们不得不为 `Employee::printCheck` 进行定义，因此我们给出了一个“该函数不能被调用”的错误消息输出。现在，我们的解决方案显得更为优雅。通过将函数 `Employee::printCheck` 定义为纯虚函数，编译器禁止了对该函数的调用。

### 示例 15.6 抽象类 `Employee` 的接口

```

1
2 // 这是头文件 employee.h。
3 // 这是抽象类 Employee 的接口。

4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H                                这是示例 14.1 所给出的类 Employee 的
                                                    一个改进版。

6 #include <string>
7 using std::string;

8 namespace SavitchEmployees
9 {
10     class Employee
11     {
12     public:
13         Employee();
14         Employee(const string& theName, const string& theSsn);
15         string getName() const;
16         string getSsn() const;
17         double getNetPay() const;
18         void setName(const string& newName);
19         void setSsn(const string& newSsn);
20         void setNetPay(double newNetPay);
21         virtual void printCheck() const = 0; ← 这是一个纯虚函数。

22     private:
23         string name;
24         string ssn;
25         double netPay;

26     };

27 } // SavitchEmployees

28 #endif //EMPLOYEE_H

```

### 自测练习

3. 让一个抽象类的所有成员函数都为纯虚函数是否合法？
4. 关于示例 15.6 给出的类 `Employee`，下面哪些是合法的？

a. `Employee joe;`  
`joe = Employee();`

```

b. class HourlyEmployee : public Employee
{
public:
    HourlyEmployee( );
    <一些更为合法的成员函数的定义，它们都不是纯虚函数。>
private:
    double wageRate;
    double hours;
};

int main( )
{
    Employee joe;
    joe = HourlyEmployee( );

c. bool isBossOf(const Employee& e1, const Employee& e2);

```

## 15.2 指针和虚函数

切莫追逐幻影而失去已有之物。

伊索，《狗和自己的影子》

在这一部分，我们将探索虚函数的一些更加精妙的东西。理解这部分的知识需要掌握第 10 章指针的知识。

### 虚函数与扩展类型兼容性

如果类 `Derived` 是基类 `Base` 的一个派生类，那么可以把一个类型为 `Derived` 的对象赋值给一个类型为 `Base` 的变量（或参数），但是把类型为 `Base` 的对象赋值给类型为 `Derived` 的变量却不行。通过一个具体的例子来看待和认识这个问题是明智之举。例如，`DiscountSale` 是类 `Sale` 的派生类（见示例 15.1 和示例 15.3）。那么可以把一个类 `DiscountSale` 的对象分配给一个类型为 `Sale` 的变量，这是因为一个 `DiscountSale` 对象也是一个 `Sale` 对象。但是，不能反过来赋值，因为一个 `Sale` 对象不一定是个 `DiscountSale` 对象。把一个派生类对象赋值给它对应的基类变量（或参数）的这一方式对于通过继承实现代码重用是至关重要的。但是，这么做也的确存在一些问题。

例如，假设一个程序或程序块包含下列类定义：

```

class Pet
{
public:
    string name;
    virtual void print( ) const;
};

class Dog : public Pet
{
public:

```

```

        string breed;
        virtual void print() const; // 此处并不需要关键字 virtual,
                                   // 这里写出来只是为了理解起来更清晰明了。
    };

    Dog vdog;
    Pet vpet;

```

现在，我们来具体讨论一下数据成员：name 和 breed（为了简化此例，我们把成员变量定义为 public 类型。在实际应用中，成员函数所操作的成员变量应该是 private 类型）。

任何一只 Dog 都是一个 Pet。在程序中认为 Dog 对象的值也就是 Pet 对象的值，这看起来似乎是合理的，那么下列情况也是允许的：

```

    vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;

```

C++ 语言的确允许这种赋值方式。你可以把一个变量的值，比如 vdog 的值，赋给它的父类变量，比如 vpet，但你不能反过来赋值。尽管这种赋值是可以的，但赋给变量 vpet 的值丢失了字段 breed。这种现象被称为**切片问题**。下列试图对 breed 进行取值的操作将会出现错误消息：

```

    cout << vpet.breed;
    // 非法：类 Pet 没有 breed 成员。

```

你可以认为这是合理的，因为一旦一个 Dog 对象变为 Pet 对象后，像其他任何 Pet 对象一样，它会被作为 Pet 类型的对象进行处理，并且不再拥有任何 Dog 对象所特有的属性。这可能会引发一场生动、热烈的哲学争论，但通常情况下，它只会在编程时为你制造一些麻烦。虽然在某些地方我们想把一只叫 Tiny 的狗当作 Pet 来看，但它仍然只是一只丹麦大狗，我们希望使用它的 breed 成员。

幸运的是，C++ 语言确实为我们提供了一种把 Dog 视为 Pet 的方法，同时还可以不丢弃 breed 成员。为了达到这个目的，我们使用指向动态变量的指针。

假设我们在程序中添加了下列声明：

```

    Pet *ppet;
    Dog pdog;

```

如果我们使用指针和动态变量，我们能够把 Tiny 当作 Pet 来处理，而不丢失 Tiny 的 breed 成员。在 C++ 中下列代码是合法的：<sup>1</sup>

```

    pdog = new Dog;
    pdog->name = "Tiny";
    pdog->breed = "Great Dane";
    ppet = pdog;

```

而且，我们仍然可以访问 ppet 所指向节点的 breed 字段。假设

```

    Dog::print() const;

```

<sup>1</sup> 如果你对->运算符不熟悉，请参考第10章中标题为“运算符->”的部分。

被定义为如下代码：

```
void Dog::print( ) const
{
    cout << "name:" << name << endl;
    cout << "breed:" << breed << endl;
}
```

下列语句

```
ppet->print ( );
```

将会在屏幕上输出如下信息：

```
name: Tiny
breed: Great Dane
```

由于 print( ) 是一个虚成员函数，因此我们才得到了所期望的输出结果。示例 15.7 是解决切片问题的测试代码。

### 示例 15.7 解决切片问题

```
1 // 使用虚函数解决切片问题的样例程序。
2 #include <string>
3 #include <iostream>
4 using std::string;
5 using std::cout;
6 using std::endl;

7 class Pet
8 {
9 public :
10     string name;
11     virtual void print( ) const ;
12 };

13 class Dog : public Pet
14 {
15 public :
16     string breed;
17     virtual void print( ) const;
18 };
```

为了简单起见，我们把例中的成员变量声明为 public。在一个真实的应用中，应该把成员变量声明为 private，通过成员函数去访问成员变量。

```
19 int main( )
20 {
21     Dog vdog;
22     Pet vpet;
23     vdog.name = "Tiny";
24     vdog.breed = "Great Dane";
25     vpet =.vdog;
26     cout << "The slicing problem:\n";
27     //vpel.breed; 这是非法的，因为类 Pet 没有名为 breed 的成员变量。
28     vpet.print( );
29     cout << "Note that it was print from Pet that was invoked.\n";
30     cout << "The slicing problem defeated:\n";
```

此处并不需要关键字 virtual，这里写出来只是为了理解起来更清晰明了。

```

31     Pet *ppet;
32     Dog *pdog;
33     pdog = new Dog;
34     pdog->name = "Tiny";
35     pdog->breed = "Great Dane";
36     ppet = pdog;
37     ppet->print();
38     pdog->print();

39     // 下面的代码直接访问成员变量,
40     // 而不是通过虚函数, 使用这种方式会产生错误:
41     //cout << "name: " << ppet->name << " breed: "
42     //      << ppet->breed << endl;
43     // 上述代码产生的错误消息是:
44     // 类 Pet 没包含名为 breed 的成员变量。

45     return 0;
46 }

47 void Dog::print() const
48 {
49     cout << "name: " << name << endl;
50     cout << "breed: " << breed << endl;
51 }

52 void Pet::print() const
53 {
54     cout << "name: " << name << endl;
55 }

```

这两个 print 的输出结果是一样的。  
name: Tiny  
breed: Great Dane

注意, 这里没有提到 breed。

### 示例运行结果

```

The slicing problem:
name: Tiny
Note that it was print from Pet that was invoked.
The slicing problem defeated:
name: Tiny
breed: Great Dane
name: Tiny
breed: Great Dane

```

基于动态变量的面向对象编程是一种非常特殊的编程考虑方式。刚开始时, 我们可能会对此感到很迷惑, 但如果记住如下两条简单的规则, 这种编程方法将会对我们有所帮助:

1. 如果指针 pAncestor 所指向的对象类型是指针 pDescendant 所指向的对象类型的祖先类, 那么下列指针赋值是合法的:

```
pAncestor = pDescendant;
```

此外, pDescendant 所指向的动态变量的所有数据成员或者成员函数都不会因此而丢失。

2. 尽管动态变量所有多余的数据成员依然存在，但你还是需要通过虚成员函数才能访问到它们。



### 陷阱：切片问题

尽管将一个派生类对象赋给一个基类变量是合法的，但是派生类对象赋给基类对象会导致数据成员的丢失。在赋值过程中，任何在基类中没有而在派生类中存在的数据成员都会丢失。相似地，对于赋值后的基类对象，任何在基类中未定义的成员函数都是不可用的。

例如，如果类 `Dog` 是类 `Pet` 的派生类，那么下面的代码是合法的：

```
Dog vdog;
Pet vpet;
vpel = vdog;
```

但是，对象 `vpel` 不可能去调用类 `Dog` 的成员函数，除非这个函数也是类 `Pet` 的成员函数，并且，在对象 `vdog` 的成员变量中，除了继承于类 `Pet` 的成员变量外，其他所有成员变量都丢失了。这就是对象切片问题。

请注意，简单地将一个成员函数声明为虚成员函数并不能解决切片问题。注意看下列来自于示例 15.7 的代码片段：

```
Dog vdog;
Pet vpet;

vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpel = vdog;
...
vpel.print();
```

尽管对象 `vdog` 的类型是 `Dog`，当 `vdog` 被赋值给变量 `vpel`（其类型为 `Pet`）后，它变成了一个类型为 `Pet` 的对象。因此，`vpel.print()` 函数是类 `Pet` 中定义的函数版本，而不是类 `Dog` 中定义的函数版本。尽管 `print()` 函数是虚函数，`vpel` 对象所使用的 `print()` 函数依然是 `Pet` 类中定义的函数版本。为了解决对象切片问题，函数必须是虚函数，此外，我们必须使用指针和动态变量。■

### 自测练习题

- 为什么不能将一个基类对象赋给一个派生类变量？
- 把一个派生类对象赋给一个基类变量会产生什么问题？
- 假设基类和派生类都有一个同名的成员函数。当你有一个指向派生类对象的基类指针，并且通过该指针调用了成员函数，请讨论一下是什么决定了实际调用的是基类成员函数还是派生类成员函数？



## 提示：使析构函数成为虚函数

总是把析构函数声明为一个虚函数是一个好的编程准则。在解释为什么这是一个好的准则之前，让我们先简单地介绍一下析构函数和指针是如何交互的，以及把析构函数声明为虚函数到底意味着什么。

考虑如下代码，其中 `SomeClass` 是一个带析构函数的类，它的析构函数不是虚函数：

```
SomeClass *p = new SomeClass;
...
delete p;
```

当执行 `delete p` 语句时，类 `SomeClass` 的析构函数被自动调用。现在，我们来看看当把一个析构函数标记为 `virtual` 时发生了哪些情况。

有一种最简单的方法可以用来描述析构函数与虚函数进行交互的机制：把所有的基类及其派生类的析构函数看成具有相同名字的函数。例如，假设 `Derived` 是类 `Base` 的派生类，并且类 `Base` 的析构函数被标记为 `virtual`。那么考虑下面的代码：

```
Base *pBase = new Derived;
...
delete pBase;
```

当对 `pBase` 对象进行 `delete` 操作时，程序会去调用一个析构函数。由于类 `Base` 中的析构函数被标记为 `virtual`，并且所操作的对象类型为 `Derived`，那么类 `Derived` 的析构函数将会被调用（它会接下来调用类 `Base` 的析构函数）。如果类 `Base` 的析构函数没有被声明为 `virtual`，那么只有类 `Base` 的析构函数会被调用。

另外还需要注意的一点是，当一个析构函数被标记为 `virtual` 时，其所有派生类的析构函数将自动成为 `virtual`（不管它们是否被标记为 `virtual`）。再次提一下，析构函数的这种行为好像基类和所有派生类的析构函数都具有相同的名字（尽管实际上它们的析构函数名字是不一样的）。

现在我们再来解释为什么所有的析构函数都应当被声明为 `virtual`。考虑一下当基类中的析构函数没有被声明为 `virtual` 将会发生哪些情况。特别是，考虑基类 `PFFArrayD`（`double` 类型的部分填充数组）和其派生类 `PFFArrayDBak`（带备份功能的 `double` 类型的部分填充数组）。我们在第 14 章已经讨论过这两个类了，那时候我们还不了解虚函数的概念，因此，类 `PFFArrayD` 的析构函数并未被标记为 `virtual`。在示例 15.8 中，我们汇总了类 `PFFArrayD` 和类 `PFFArrayDBak` 的所有代码，这样我们就不必回头再去查看第 14 章中两个类的相关代码。

考虑如下代码：

```
PFFArrayD *p = new PFFArrayDBak;
...
delete p;
```

由于基类的析构函数并没有被标记为 `virtual`，因此只有基类 `PFFArrayD` 的析构函数会被调用。这样的话，成员数组 `a` 所占用的内存将会被释放，但成员数组 `b` 所占用的内存将永远不会被释放（直到程序结束）。

另一方面，如果（不像示例 15.8 那样）基类 PFArrayD 的析构函数被标记为 virtual，那么，当对 p 执行 delete 操作时，类 PFArrayDBak 的构造函数将会被调用（因为所创建的对象 p 的数据类型为 PFArrayDBak）。类 PFArrayDBak 的析构函数将会删除数组 b 并接着自动地调用基类 PFArrayD 的析构函数，PFArrayD 的析构函数将会释放成员数组 a 所占用的内存。因此，如果基类的析构函数被标记为 virtual，所有被占用的内存将会被释放。为了防止出现如上述讨论所提及的问题，最好总是将析构函数标记为 virtual。■

## 向下类型转换和向上类型转换

你或许误以为通过一些类型转换的方法可以规避切片问题。但是，事实并非如你想象中那么简单。下面的类型转换是非法的：

```
Pet vpet;
Dog vdog; //Dog 是基类 Pet 的派生类。
...
vdog = static_cast<Dog>(vpel); // 非法！
```

但是，向下的类型转换却是合法的，甚至你可以不需要写类型转换运算符也都是合法的：

```
vpel = vdog; // 合法（但会产生切片问题）
```

### 向上类型转换

从子孙类向祖先类的转换被称为**向上类型转换**，因为在类的层次结构上它是向上移动的。向上类型转换是安全的，因为你只是简单地丢弃了一些信息（丢弃了一些成员变量和函数）。因此，下列表达式是安全的：

```
vpel = vdog;
```

### 向下类型转换

从祖先类到子孙类的转换被称为**向下类型转换**，这种类型转换是危险的，因为这样做会增加一些信息（增加了成员变量和函数）。在第 1 章中，我们简单地讨论了 dynamic\_cast 这个运算符，可以用它来进行向下的类型转换。在解决切片问题时，它可能有些用处，但却是危险的、不可靠的，充满了陷阱。dynamic\_cast 可以允许我们进行向下类型转换，但是它只对指针类型变量有效，如下所示：

```
Pet *ppet;
ppet = new Dog;
Dog *pdog = dynamic_cast<Dog*>(ppet); // 危险！
```

即使像这样简单的向下类型转换，也可能会出现失败的情况，因此，我们不推荐使用向下类型转换。

dynamic\_cast 会在向下类型转换失败时发出通知。如果转换失败，dynamic\_cast 会返回 NULL（实际上返回的是整数 0）。<sup>2</sup>

如果打算使用向下类型转换，请务必记住以下两点：

2 标准的说法是：“指针类型的类型转换失败时，会返回一个指向结果类型的空指针。引用类型的类型转换失败会抛出一个 bad\_cast。”

1. 需要时刻清楚在进行向下类型转换时所添加的信息的确存在。
2. 成员函数必须是虚函数，因为 `dynamic_cast` 使用虚函数的信息进行向下类型转换。

### C++如何实现虚函数

我们并不需要为了使用虚函数而去了解编译器是如何工作的，这就是所谓的信息隐藏原则。在所有的编程思想中，信息隐藏原则是最基本的原则。特别是，当我们使用虚函数时，我们并不需要知道虚函数是如何实现的。但是，很多人会发现一个具体的实现模型有助于提高他们的理解；当我们学习其他书里关于虚函数的知识时，我们可能也会碰到一些提到虚函数实现的相关内容。因此，我们将简要介绍一下虚函数是如何实现的。所有语言（包括 C++）的所有编译器实现虚函数的方法基本上相同。

#### 示例 15.8 回顾类 `PFArrayD` 和类 `PFArrayDBak`

```
class PFArrayD
{
public:
    PFArrayD( );
    ...
    ~PFArrayD( );
protected:
    double *a; // 表示 double 类型的数组。
    int capacity; // 表示数组的容量。
    int used; // 表示数组当前已经使用的容量。
};

PFArrayD::PFArrayD( ) : capacity(50), used(0)
{
    a = new double[capacity];
}

PFArrayD::~~PFArrayD( )
{
    delete [] a;
}
```

这里给出了基类 `PFArrayD` 的一些具体细节。该类比较完整的定义参见示例 14.8 和示例 14.9，但是，在本例中，我们给出了在学习本章内容时所需要了解的所有具体细节。

```
class PFArrayDBak : public PFArrayD
{
public:
    PFArrayDBak( );
    ...
    ~PFArrayDBak( );
private:
    double *b; // 用于原始数组的备份。
    int usedB; // 用于对继承的成员变量 used 进行备份。
};
```

析构函数应当被标记为 `virtual`，但当我们编写上述两个类时，我们还未学习虚函数的相关知识。

```

PFArrayDBak::PFArrayDBak( ) : PFArrayD( ), usedB(0)
{
    b = new double [capacity];
}
PFArrayDBak::~PFArrayDBak( )
{
    delete [] b;
}

```

这是关于派生类 PFArrayDBak 的一些具体实现细节。类 PFArrayDBak 的完整定义参见示例 14.10 和示例 14.11，但本例也已经给出了学习本章知识所需要的该类的实现细节。

## 虚函数表

如果一个类有一个或多个成员函数都是虚函数，那么编译器会为该创建表，该表被称为**虚函数表**。对于每一个虚成员函数，虚函数表都有一个对应的指针（内存地址）指向虚函数。每一个指针指向其对应的成员函数的代码所在的位置。如果一个虚函数是继承于其父类并且未作任何变化，那么它的表入口指向父类中该函数的定义。（或者其他祖先类，如果可能的话）。如果在该类中其他虚函数有了新的定义，那么表中新定义的虚函数的指针指向新定义函数的代码入口。（记住，虚函数的属性是会继承下来的，因此一旦一个类有一个虚函数表，那么所有它的子孙类也会有一个虚函数表。）

一旦我们创建了一个包含一个或多个虚函数的类的对象，C++ 运行时系统就会自动增加另外一个指针来描述这个存储在内存中的对象。这个指针指向类的虚函数表。当我们使用一个指向该对象的指针（对，这是另外一个指针）来调用一个成员函数时，运行时系统使用虚函数表来决定应该调用这个成员函数的哪一个定义，而不是根据指针的类型来做决定。

当然，这些都是自动化的，因此我们根本不必担心它的具体运行过程。一个编写编译器的程序员甚至都可以不用采用其他方式来实现虚函数，只要虚函数能正确运行（事实上，从来不会有人采用其他方式来实现这一机制）。

## 自测练习

### 8. 为什么下列表达式非法？

```

Pet vpet;
Dog vdog; //Dog 是基类 Pet 的派生类。
...
vdog = static_cast(vpet); //非法！

```

## 本章小结

- 延迟绑定是指程序应该使用成员函数的哪一个具体版本是在运行时决定的。在 C++ 中，使用延迟绑定的成员函数被称为虚函数。多态是延迟绑定的另一种说法。
- 纯虚函数是一种没有具体定义的成员函数。在成员函数声明中，用关键字 `virtual` 及符号 `=0` 来表示它。具有一个或多个纯虚函数的类被称为抽象类。
- 抽象类是一种数据类型，它可以用来作为基类来派生其他类。但是，你不能创建一个抽象类类型的对象（除非它是一些派生类的对象）。
- 可以把一个派生类对象赋值给它的基类对象，但是基类中没有的成员变量将会丢失。这就是所谓的切片问题。
- 如果指针 `pAncestor` 所指的对象的数据类型是指针 `pDescendant` 所指对象数据类型的基类，那么下列赋值是允许的：

```
pAncestor = pDescendant;
```

此外，由 `pDescendant` 所指对象的动态变量的所有数据成员和成员函数都不会丢失。尽管该动态变量的所有多余的字段都存在，我们还是需要使用虚成员函数来访问它们。

- 把析构函数声明为 `virtual` 是一个良好的编程习惯。

## 自测练习题答案

1. 从本质上来说，三个术语没什么区别。它们指的是同一主题。只是在用法上略有区别。（虚函数是成员函数的一种；延迟绑定是一种在运行时决定使用虚函数的哪个具体定义的机制；多态是延迟绑定的另一种叫法。）
2. 输出结果将变成下列内容：
 

```
Discounted item is not cheaper.
```
3. 是的，一个抽象类的所有成员函数都是纯虚函数是合法的。
4. a. 非法，因为 `Employee` 是抽象类。  
b. 合法。  
c. 合法，因为抽象类也是一种数据类型。
5. 没有成员可以用于给派生类新增的成员赋值。
6. 尽管把一个派生类对象赋值给一个基类变量是合法的，但是派生类对象中的一些成员是基类中所没有的成员，派生类的这部分成员将会丢失。这种情况就是我们常说的切片问题。
7. 如果基类中的函数带了 `virtual` 关键字修饰符，那么派生类的成员函数将会被

调用。如果基类的成员函数没有 `virtual` 关键字修饰符，那么基类成员函数将被调用。

8. 由于 `Dog` 比 `Pet` 有更多的成员变量，因此，对象 `vpeter` 就可能没有足够的数据来提供给类型为 `Dog` 的对象的成员变量。

## 编程练习

1. 考虑在一个图形系统中，它有很多表示各种图形的类——比如说，矩形 (`Rectangle`)、正方形 (`Square`)、三角形 (`Triangle`)、圆形 (`Circle`) 等。例如，矩形可能会有高、宽和中心点三个数据成员，而正方形和圆形可能只有中心点及边长或半径两个数据成员。在一个设计良好的系统中，这些图形类都可以从一个被称为 `Figure` 的公共类中派生。请实现这个图形系统。

类 `Figure` 是基类。你应该只增加类 `Rectangle` 和 `Triangle` 作为基类 `Figure` 的派生类。每个类都包含成员函数 `erase` 和 `draw`。每一个成员函数输出一个消息告诉程序调用者正在调用的是什么函数，以及当前调用对象属于哪个类。这些程序除了输出消息之外，其他什么也不做。成员函数 `center` 调用 `erase` 和 `draw` 函数分别用来擦除屏幕中心的图，并重新绘制图形。由于函数 `erase` 和 `draw` 只是用来输出信息，因此，`center` 函数除了调用函数 `erase` 和 `draw` 之外，并不会产生什么实际的“居中”行为。此外，需要在一个成员函数 `center` 中增加一条输出消息来宣布函数 `center` 正在被调用。成员函数不要包含任何参数。这个编程练习题有三部分需要考虑：

- a. 类定义一定不要使用虚函数。编译所编写的程序并进行测试。
- b. 把基类成员函数声明为虚函数。编译所编写的程序并进行测试。
- c. 解释上面两种结果的不同。

对一个真实的例子来说，你必须使用相关代码来替换掉上面所定义的每一个成员函数，替换后的成员函数可以进行真正的画图。下面的编程练习 2 需要你完成这个真实的例子。使用下面这个 `main` 函数进行所有的测试。

```
// 下面的程序用来对编程练习 1 进行测试。
#include <iostream>
#include "figure.h"
#include "rectangle.h"
#include "triangle.h"
using std::cout;
int main()
{
    Triangle tri;
    tri.draw();
    cout <<
        "\nDerived class Triangle object calling center().\n";
    tri.center(); // 调用函数 draw 和 center。
```

```

    Rectangle rect;
    rect.draw();
    cout <<
        "\nDerived class Rectangle object calling center().\n";
    rect.center(); // 调用函数 draw 和 center.
    return 0;
}

```

2. 完善编程练习1。给出下列各个构造函数及成员函数的新的定义：Figure::center、Figure::draw、Figure::erase、Triangle::draw、Triangle::erase、Rectangle::draw 和 Rectangle::erase。重新定义的目的是使得 draw 函数能够在屏幕上真正地画出来，在画图时把字符串“\*”放在合适的位置构成所需要画的图。对于 erase 函数来说，你可以简单地把屏幕清空（通过使用空行或者其他更为复杂的方法来实现清空屏幕）。在实现过程中包含许多具体细节，你必须自行决定如何处理这些细节。
3. 本道编程练习的目标是创建一个简单的二维掠食者 - 被掠食者的模拟程序。在这个模拟程序中，被掠食者是蚂蚁，掠食者是蚁狮。这些动物居住在一个  $20 \times 20$  的网格中。一个动物一次只能占一个网格。网格是封闭的，因此不允许动物们移出它所居住的小网格。把时间作为仿真步。每一个动物在每一个时间步上执行一个动作。

蚂蚁的行为依据下面这个模型：

- 移动。在每一个时间步，蚂蚁随机地朝上、下、左或者右进行移动。如果选定方向的邻居网格被占用或者蚂蚁将跨出整个网格，那么蚂蚁就停留在当前单元格。
- 繁殖。如果一只蚂蚁已经存活了三个时间步，那么在最后一个时间步（也就是说，蚂蚁移动之后），这只蚂蚁将会繁殖。我们通过在相邻的空单元（上、下、左或者右）上生成一只新蚂蚁来模拟这一过程。如果蚂蚁所在位置紧邻的四个方向都没有空格，那么，它将不会繁殖新蚂蚁。一旦一只蚂蚁繁殖了后代，它就不能再繁殖了，直到它再活三个时间步之后，才可以再次繁殖。

蚁狮的行为依据下面这个模型：

- 移动。在每一个时间步，蚁狮将向和它紧邻的、有蚂蚁的网格移动，并吃掉那个网格中的蚂蚁。如果蚁狮紧邻的四个网格没有蚂蚁，那么它将移动到一个位置，具体移动规则和蚂蚁的移动规则是一样的。注意，一只蚁狮不能吃其他蚁狮。
- 繁殖。如果一只蚁狮已经存活了八个时间步，那么在最后一个时间步，这只蚁狮将会繁殖。它的繁殖方式同蚂蚁的繁殖方式是一样的。
- 饿死。如果一只蚁狮在三个时间步内没有吃到蚂蚁，那么它将会饿死。这时，请将饿死的蚁狮从网格中移除。

在新一轮移动中，蚁狮的移动要先于蚂蚁的移动。

编写一个程序来对上述过程进行模拟。使用 ASCII 字符“O”来代表蚂蚁，使用 ASCII 字符“X”来表示蚁狮。创建一个名为 Organism 的类，它封装了蚂蚁和蚁狮所包含的公共数据。这个类应该包含一个名为 move 的虚函数，这个虚函数是在类 Ant 和 Doodlebug 中定义的。可能你需要一个多余的数据结构来追踪哪个动物移动了。

使用五只蚁狮和 100 只蚂蚁来初始化这个模拟过程。每个时间步之后，提示用户按下“Enter”键来移到下一个时间步。你应该可以看到一个掠食者和被掠食者物种数目的循环模式，随机的扰动可能会导致一个或两个物种的灭绝。

#### 4. 这道编程练习题需要你先完成第 14 章的编程练习 9。

```
class Creature
{
    private :
        int type; // 0 human, 1 cyberdemon, 2 balrog, 3 elf
        int strength; // 我们所造成的损伤程度
        int hitpoints; // 我们所承受的损伤程度
        string getSpecies(); // 返回物种的类型
    public :
        Creature();
        // 初始化人的两个属性，10 strength, 10 hitpoints

        Creature(int newType, int newStrength, int newHit);
        // 初始化动物的三个属性：type、strength 和 hitpoints。
        // 并为属性 type、strength 和 hitpoints 增加适当的取值函数和赋值函数。

        int getDamage();
        // 返回这个动物在一轮战争中所造成的损伤程度。
};
```

这是函数 getSpecies() 的实现：

```
string Creature::getSpecies()
{
    switch (type)
    {
        case 0: return "Human";
        case 1: return "Cyberdemon";
        case 2: return "Balrog";
        case 3: return "Elf";
    }
    return "Unknown";
}
```

函数 getDamage() 输出并返回在一轮战斗中动物造成的损伤。计算损伤的规律如下所示：

- 每一个动物造成的损伤是一个随机数  $r$ ，其中  $0 < r \leq \text{strength}$ 。
- 魔鬼 (demon) 有 5% 的机会实施一次有魔力的攻击，这会造成额外的 50 个



损伤点。Balrog 和 Cyberdemon 都是恶魔。

- Elf 有 10% 的机会实施一次有魔力的攻击，它比正常攻击会产生双倍的损伤。
- Balrog 动作非常快，因此，它们可以攻击两次。

函数 `getDamage()` 的代码如下所示：

```
int Creature::getDamage()
{
    int damage;
    // 所有动物均会造成损伤，损伤程度是一个不超过其 strength 属性值的随机数。
    damage = (rand() % strength) + 1;
    cout << getSpecies() << " attacks for " <<
        damage << "points!" << endl;

    // 魔鬼造成 50 个损伤点的机会是 5%。
    if ((type == 2) || (type == 1))
        if ((rand() % 100) < 5)
        {
            damage = damage + 50;
            cout << "Demonic attack inflicts 50 "
                << "additional damage points!" << endl;
        }

    // Elf 造成双倍损伤的机会是 10%。
    if (type == 3)
    {
        if ((rand() % 10) == 0)
        {
            cout << "Magical attack inflicts " << damage <<
                "additional damage points!" << endl;
            damage = damage * 2;
        }
    }

    // Balrog 速度如此之快，以至于它们可以攻击两次。
    if (type == 2)
    {
        int damage2 = (rand() % strength) + 1;
        cout << "Balrog speed attack inflicts " << damage2 <<
            "additional damage points!" << endl;
        damage = damage + damage2;
    }
    return damage;
}
```

这个实现存在一个问题，那就是增加新的动物不是很方便。使用继承来重新编写这个类，这样我们可能不再需要变量 `type` 了。类 `Creature` 应该是基类。类 `Demon`、`Elf` 和 `Human` 应该是类 `Creature` 的派生类。类 `Cyberdemon` 和 `Balrog` 应该是类 `Demon` 的派生类。我们需要重新编写函数 `getSpecies()` 和 `getDamage()`，保证它们可正常应用于每个类中。

例如，在每个类中的函数 `getDamage()` 应该只用来计算适用于这个类对象

的损伤。因此，可以通过组合每一级继承层次的 `getDamage()` 的结果来计算整个损伤。举一个例子，调用 `Balrog` 对象的 `getDamage()` 时应该调用对象 `Demon` 的 `getDamage()`，而调用 `Demon` 的 `getDamage()` 时，应该调用 `Creature` 对象的 `getDamage()`。依据 `demon` 会造成随机的 5% 破坏，`Balrog` 会带来双倍的破坏这些规则，计算出所有动物造成的基本破坏。

请在你的程序中包含对私有变量进行赋值和取值的函数。编写一个 `main` 函数，它包含可以测试你的类的代码片段。为每种动物创建一个对象，重复输出函数 `getDamage()` 的结果。首先，把函数 `getDamage()` 声明为虚函数。然后，在你的主程序中编写一个函数 `battleArena(Creature &creature1, Creature &creature2)`，这个函数把两个 `Creature` 对象作为输入。函数应该计算出 `creature1` 所造成的破坏，减去 `creature2` 的生命点数，反之亦然。如果两种动物的生命点数都是零或者更小，那么这次战斗打成平局。否则，在最后一轮，如果一种动物有正的生命点数，另一种动物没有，那么战斗结束。除非战斗打成平局或者结束了，否则，函数一直循环下去。由于函数 `getDamage()` 是虚函数，因此，它应该调用每个特定动物类中所定义的 `getDamage()` 函数。通过包含不同动物的几场战争来测试你的程序。

5. 下列代码表示的是一个猜测游戏。在游戏中，两位选手玩猜数字游戏。你的任务是扩展这个程序，使用两个对象来进行游戏，一个对象表示人类选手，另一个表示计算机选手。

```
bool checkForWin(int guess, int answer)
{
    if (answer == guess)
    {
        cout << "You're right! You win!" << endl;
        return true;
    }
    else if (answer < guess)
        cout << "Your guess is too high." << endl;
    else
        cout << "Your guess is too low." << endl;
    return false;
}

void play(Player &player1, Player &player2)
{
    int answer = 0, guess = 0;
    answer = rand() % 100;
    bool win = false;

    while (!win)
    {
        cout << "Player 1's turn to guess." << endl;
        guess = player1.getGuess();
        win = checkForWin(guess, answer);
        if (win) return;

        cout << "Player 2's turn to guess." << endl;
```

```

        guess = player2.getGuess();
        win = checkForWin(guess, answer);
    }
}

```

函数 `play` 接收两个 `Player` 对象作为输入参数。定义 `Player` 类，它包含一个名字为 `getGuess()` 的虚函数。在成员函数 `Player::getGuess()` 的实现中，可以简单地返回 0。接着，定义一个名字为 `HumanPlayer` 的类，它派生于类 `Player`。在函数 `HumanPlayer::getGuess()` 的实现中，应当提示用户输入一个数字并返回用户从键盘输入的这个数字。然后，再定义一个名字为 `ComputerPlayer` 的类，它也派生于类 `Player`。在函数 `ComputerPlayer::getGuess()` 的实现中，应当在 0 ~ 100 中随机地选择一个数字。最后，构建一个 `main` 函数，它使用两个实例作为输入来调用函数 `play`。这两个实例可能是：两个 `HumanPlayer` 实例（人和人对弈），一个 `HumanPlayer` 实例和一个 `ComputerPlayer` 实例（人和计算机对弈），两个 `ComputerPlayer` 实例（计算机和计算机对弈）。

- 在编程练习 5 中，计算机选手在玩猜数字时，表现不是很好，因为它是在随机选一个数字进行猜测。请修改程序，使得计算机选手能根据一些信息进行猜测。具体策略由你来决定，但是你必须为类 `Player` 和类 `ComputerPlayer` 增加函数，使得 `play(Player &player1, Player &player2)` 函数能够发送猜测结果给计算机选手。换句话说，计算机必须被告知，它的上一次猜测结果太高还是太低；也必须被告知，它的对手的上一次猜测结果是太高还是太低。这样的话，计算机选手就可以使用这些信息来修正下一次猜测。
- 在下面的代码清单中，类 `Dice` 用来模拟滚动一个骰子，使它呈现不同的面。默认是一个有六个面的标准骰子。函数 `rollTwoDice` 模拟摇动两次骰子，并返回两次骰子点数之和。函数 `srand` 要求引入 `cstdlib` 库。

```

class Dice
{
public:
    Dice();
    Dice(int numSides);
    virtual int rollDice() const;
protected:
    int numSides;
};

Dice::Dice()
{
    numSides = 6;
    srand(time(NULL)); // 种子随机数生成器
}

Dice::Dice(int numSides)
{
    this->numSides = numSides;
    srand(time(NULL)); // 种子随机数生成器
}

```

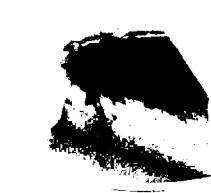
```
int Dice::rollDice() const
{
    return (rand() % numSides) + 1;
}
// 接收两个骰子对象，滚动这两个骰子，返回两次结果之和。
int rollTwoDice(const Dice& die1, const Dice& die2)
{
    return die1.rollDice() + die2.rollDice();
}
```

写一个创建两个 Dice 对象的 main 函数，这两个 Dice 对象有许多你选择的面。在一个循环中，调用函数 rollTwoDice，迭代 10 次，来验证函数能够按预期正常工作。

接下来，创建一个自己的类 LoadedDice，它派生于类 Dice。增加一个默认构造函数和一个包含一个参数的构造函数，这个参数表示骰子面的数字。在类 LoadedDice 中覆盖函数 rollDice，使得这个函数能够有 50% 的机会返回可能出现的最大数（也就是 numSides），否则，它返回类 Dice 的 rollDice 函数所返回的值。

测试你所编写的类时，把 main 函数中的 Dice 对象替换为 LoadedDice 对象，其他的什么也不需要改变。应该会有更多的骰子滚动出可能的最高值。多态机会导致类 LoadedDice 的 rollDice 函数被调用，而不是调用函数 rollTwoDice 中类 Dice 的 rollDice 函数。





模板

16



## 16.1 函数模板 586

函数模板语法 587

陷阱：编译器的复杂性 590

提示：如何定义模板 592

示例：一个通用的排序函数 592

陷阱：使用模板时误用了不正确的数据类型 596

## 16.2 类模板 597

类模板语法 598

示例：一个数组模板类 601

模板中的vector 和 basic\_string 606

## 16.3 模板和继承 606

示例：带备份的部分填充数组模板类 607

# 第 16 章 模板

所有人都会死，  
亚里士多德是个人，  
那么，亚里士多德是会死的。  
所有 X 都是 Y，  
Z 是 X，  
那么，Z 是 Y。  
所有猫都很顽皮，  
加菲猫是一只猫，  
那么，加菲猫很顽皮。

一个简短的推论

## 概述

本章讨论 C++ 模板，通过使用模板我们可以定义出能够适用于不同参数类型的函数和类。通过使用模板我们可以设计出面向不同参数类型的函数，也可以定义出比之前章节所给出的更具通用性的类。

学习 16.1 节的内容只需要用到第 1 ~ 5 章的知识，16.2 节需要用到第 1 ~ 11 章的知识，但对第 12 ~ 15 章的知识不做要求。16.3 节需要 16.1 节和 16.2 节的相关知识，还需要第 14 章中关于继承的相关知识，同时也需要所有与 16.2 节相关的所有章节的知识。16.3 节将一些成员函数标记为 virtual。虚函数（virtual function）是第 15 章重点介绍的内容。但是，对于本章内容而言，不需要使用虚函数，在学习 16.3 节内容时，我们可以忽略甚至完全忘记所出现的所有关键字 virtual。

## 16.1 函数模板

之前我们讨论的许多 C++ 函数定义中的算法比我们在实际定义中实现的算法更具有通用性。比如说，我们之前所讨论的函数 swapValues，该函数第一次出现在第 4 章中。我们再来回顾一下该函数的定义：

```
void swapValues(int& variable1, int& variable2)
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

注意，上面这个函数定义仅适用于参数类型为 int 的变量。但是，函数体中所实现的算法也应该可以用来交换两个参数类型同为 char 的变量。如果我们想使用参数类型为 char 的 swapValues 函数，可以通过增加下列定义来重载函数名 swapValues：

```
void swapValues(char& variable1, char& variable2)
{
    char temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

但是通过这种方式来定义两个函数并不高效，难以令人满意，这是因为这两个函数几乎是完全一样的。唯一的区别就是，一个函数定义在三个地方使用了 `int` 作为参数类型，另一个函数定义在相同的三个地方使用了 `char`。如果继续按照这种函数定义方式，当我们想使用函数 `swapValues` 交换两个 `double` 类型的变量时，我们将不得不写出第三个几乎与前两个完全一样的函数定义。如果我们还是想让 `swapValues` 能够适用于更多类型的变量交换，我们将会写出更多几乎一模一样的函数定义，这将会花费大量的时间去录入代码。此外，由于许多函数定义看起来几乎完全一样，这也会使代码显得比较混乱。我们需要一种能够交换各种数据类型变量的函数定义，它的表示方式如下所示：

```
void swapValues(Type_Of_The_Variables& variable1,
               Type_Of_The_Variables& variable2)
{
    Type_Of_The_Variables temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

接下来我们将会看到，在 C++ 语言中，上述函数定义方式是可行的。我们可以定义一个适用于各种数据类型变量的函数，只是实际使用的语法比上述表示方式略微复杂一点。下面我们将介绍函数模板的语法。

## 函数模板语法

示例 16.1 为函数 `swapValues` 定义了一个 C++ 函数模板。使用该函数模板，可以交换任意类型的两个变量，只要这两个变量具有相同的数据类型。模板定义和函数声明以如下方式来表示：

```
template <class T>
```

这通常被称为**模板前缀**。它告诉编译器函数定义或者接下来的函数声明是一个模板，`T` 表示一个**类型参数**。在这里，`class` 实际上就是数据类型 (`type`<sup>1</sup>)。正如接下来我们将要看到的那样，不管类型参数 `T` 是不是类，它都可以被任意类型替换。在函数体定义内，类型参数 `T` 的使用方式与其他任何类型的使用都是相同的。

模板前缀  
类型参数

<sup>1</sup> 实际上，ANSI/ISO 标准提供了关键字 `typename` 作为模板前缀，而不使用关键字 `class`。使用关键字 `typename` 比使用 `class` 更加合理，但是作为惯例大家都使用 `class`，因此，我们也遵循大家的习惯，使用 `class` 来代替 `typename`。（通常情况下，代码的一致性比对代码进行优化更为重要。）



使用模板重  
载函数名

事实上，函数模板定义是关于多个函数定义的一个大集合。在示例 16.1 所定义的函数模板中，对于每一种可能存在的类型名，实际上都有一个与之对应的函数定义。通过使用一个具体的类型名来替换类型参数 *T* 即可得到该类型的函数定义。比如，用类型名 *double* 来替换 *T* 即可得到下面的函数定义：

```
void swapValues(double& variable1, double& variable2)
{
    double temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

用类型名 *int* 替换函数模板中的类型参数 *T*，可以得到另一个 *swapValues* 的函数定义。同样，用 *char* 替换 *T* 也可以得到一个 *char* 类型的 *swapValues* 函数定义。示例 16.1 所表示的函数模板覆盖了函数名 *swapValues*，通过这种方式来实现：对于每一种可能的数据类型，在该函数模板中都有一个略微不同的函数定义。

事实上，编译器并不会为函数名 *swapValues* 生成各种可能的参数类型的实现代码，但是在我们看来，似乎它确实为每种不同参数类型都生成了对应的实现代码。对于在程序中使用了模板的任何参数类型，编译器都会为其生成一份独立的实现代码；而对于那些没有使用模板的参数类型，编译器不会产生任何对应的实现代码。对于使用了模板的任何一种参数类型，不管在程序中它对应的函数被调用了多少次，编译器也只会为这种参数类型生成一份实现代码。我们注意到在示例 16.1 中函数 *swapValues* 被调用了两次：一次参数类型是 *int*，而另一次参数类型是 *char*，我们来看看示例 16.1 中下面的这行函数调用：

```
swapValues(integer1, integer2);
```

## 示例 16.1 函数模板

```
1 // 函数模板程序。
2 #include <iostream>
3 using std::cout;
4 using std::endl;

5 // 交换变量 1 和变量 2 的值。
6 // 赋值运算符必须能对类型 T 进行操作。
7 template <class T>
8 void swapValues(T& variable1, T& variable2)
9 {
10     T temp;

11     temp = variable1;
12     variable1 = variable2;
13     variable2 = temp;
14 }

15 int main( )
```

许多编译器在处理模板时仍存在问题。为了确保你所编写的模板能够在尽可能多的编译器上正确工作，请把模板定义和调用模板的程序放在同一个文件中，并且把模板定义部分放到调用模板的程序之前。

```

16 {
17     int integer1 = 1, integer2 = 2;
18     cout << "Original integer values are "
19         << integer1 << " " << integer2 << endl;
20     swapValues(integer1, integer2);
21     cout << "Swapped integer values are "
22         << integer1 << " " << integer2 << endl;

23     char symbol1 = 'A', symbol2 = 'B';
24     cout << "Original character values are: "
25         << symbol1 << " " << symbol2 << endl;
26     swapValues(symbol1, symbol2);
27     cout << "Swapped character values are: "
28         << symbol1 << " " << symbol2 << endl;
29     return 0;
30 }

```

### 示例运行结果

```

Original integer values are: 1 2
Swapped integer values are: 2 1
Original character values are: A B
Swapped character values are: B A

```

当 C++ 编译器开始处理上述函数调用时，编译器会注意到本次调用函数的参数类型是 `int`，那么它会用参数类型 `int` 去替换类型参数 `T` 并生成函数实现代码。同样地，当编译器看到下面的函数调用时：

```
swapValues(symbol1, symbol2);
```

它注意到在这个例子中参数类型是 `char`，于是它会使用模板来产生函数定义，在函数定义中使用类型名 `char` 来替换类型参数 `T`。

调用一个  
函数模板

注意，当我们调用一个使用函数模板而定义的函数时，不必做任何特殊的处理，我们可以像调用其他任何函数一样去调用它。编译器会为我们完成所有从函数模板生成具体函数定义的工作。

就像普通函数一样，一个函数模板可能会包含函数声明和函数定义。普通函数一般会将函数声明和函数定义分开放置，你可能会遵从这种方式来放置模板函数的声明和定义。但是，对于大多数编译器而言，它们还不支持将模板函数的定义和声明分离。因此，就像我们在示例 16.1 中所示，将模板函数的定义放到调用模板函数的文件中是最安全的方式。实际上，大多数编译器要求在第一次调用模板前先看到模板函数的定义。在调用模板函数前，可以简单地使用 `#include` 符号把要使用的模板函数的定义文件放到所编写的程序中。不同的编译器在处理模板函数时的处理方式可能会有所不同，你可以咨询一下自己身边的专家来获知编译过程的细节。

在示例 16.1 的函数模板中，我们使用了字母“`T`”来作为类型参数。这只是一种习惯而不是 C++ 语言所必需的要求。类型参数可以是任何标识符（除了 C++ 的关键词外）。用“`T`”来表示类型参数是一个不错的标识符号，但是完全可以使用其他符号。

### 多个类型 参数

在 C++ 语言中, 允许函数模板有多个类型参数。例如, 包含两个类型参数 T1 和 T2 的函数模板的开始部分如下:

```
template <class T1, class T2>
```

但是, 大多数函数模板只需要一个类型参数。在模板函数中, 不能够出现未被使用的模板参数; 也就是说, 每个模板参数必须出现在模板函数的定义中。

### 陷阱: 编译器的复杂性

许多 C++ 编译器不允许对模板进行单独编译, 因此, 可能需要把模板定义纳入到需要调用模板的代码中。通常, 至少要做到: 函数模板的使用必须先于函数模板的声明。

一些 C++ 编译器对使用模板有一些额外的、特殊的要求。如果在编译模板程序时遇到问题, 请参考所使用的编译器的相关手册或者请教相关专家。有时候, 可能需要设置一些特殊的编译器选项或者重新安排模板定义和文件中一些其他项之间的顺序。

如果想使带有模板函数的程序在尽可能多的编译器下都能被正确编译, 那么在程序布局时可以做如下设置: 将模板函数的定义放在它被调用的文件中, 并且将模板函数的定义置于所有调用它的程序代码之前。如果想把模板函数的定义与应用程序分开放置, 那么可以采用 `#include` 模板函数文件的方式来将模板函数导入到应用程序中。■

### 自测练习题

1. 编写一个名为 `maximum` 的函数模板。该函数包含两个相同类型的值作为函数的输入参数, 返回值为所输入的两个参数值中的较大者 (如果两个值相等, 返回任一)。给出这个模板的函数声明及函数定义。在函数模板的定义中你会使用到运算符 “<”。因此, 这个函数模板仅仅适用于那些可以使用运算符 “<” 的数据类型, 在函数声明中增加一段注释来对这种限制进行说明。
2. 我们已经使用过了三种不同类型的绝对值函数: `abs`、`labs` 和 `fabs`。这些函数唯一不同的地方在于它们的输入参数类型不同。对于这些绝对值函数来说, 拥有一个函数模板似乎更加合适。假设一个绝对值函数的函数模板名为 `absolute`。这个函数模板仅仅适用于那些可以使用运算符 “<” 的参数类型。使用运算符 “<” 的目的是为了便于判断输入值是否是负数, 通过和常数 0 进行比较即可得出判断。因此, 函数 `absolute` 能被任何数值类型的参数所调用, 比如 `int`、`long` 和 `double`。请给出这个模板的函数声明和函数定义。
3. 定义或者描述 C++ 模板的功能。
4. 在模板前缀的定义中:

```
template <class T>
```

参数 T 属于哪种类型的变量? 从下面所列答案中选出正确答案。

- a. T 必须是一个类。
- b. T 不应该是一个类。
- c. T 是 C++ 语言内建基本数据类型的一种。

- d. T 可以是任意类型，既可以是 C++ 内建的基本数据类型，也可以是程序员所定义的数据类型。
- e. T 可以是任意类型，既可以是 C++ 内建的基本数据类型，也可以是程序员所定义的数据类型。但 T 对数据类型的确有一些要求（请问这些要求是什么？）

### 算法抽象

正如我们在讨论 swapValues 函数时所看到的那样，有一个通用的算法可以用来交换两个具有相同类型的变量的值，变量的类型可以是任意类型。在 C++ 语言中，通过使用函数模板我们能够表示出这个更加通用的算法。这是算法抽象的一个非常简单的例子。当我们宣称我们正在使用算法抽象，那意思是说，我们正在以一种非常通用的方式来表达我们的算法，这样便于我们忽略一些次要的具体细节，而将注意力放在算法本身上面。函数模板是 C++ 编程语言的一个特征，它支持算法抽象。

### 函数模板

一个函数模板的定义及声明都是以如下代码开始的：

```
template <class Type_Parameter >
```

接下来，对于模板函数的声明（如果使用函数声明的话）和模板函数的定义来说，除了可以使用 Type\_Parameter 替换普通函数的数据类型外，其余的与普通函数的声明和定义都是一样的。

例如，下面是一个函数模板的函数声明代码：

```
template <class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

这个函数模板的定义可能如下面所示：

```
template <class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
    cout << stuff1 << endl
         << stuff2 << endl
         << stuff3 << endl;
}
```

在该例中，函数模板为每一种可能的数据类型名都编写了一个函数声明和函数定义。可以把数据类型替换成模板函数中的类型参数（在前面的例子中模板函数的类型参数是 T）。例如，考虑如下函数调用：

```
showStuff(2, 3.3, 4.4);
```

当执行这个函数调用时，编译器会通过类型名 double 替换关于 T 的函数定义。编译器会为程序中调用模板函数的每一种数据类型生成一个对应的函数定义，但它不会为其他任何一种没有调用模板函数的数据类型生成函数定义。此外，对于调用模板函数的每一种数据类型来说，不论这种数据类型的函数被调用了多少次，编译器只会为其生成一份函数定义。

## 提示：如何定义模板

在示例 16.1 中，当我们定义一个函数模板时，我们首先对一个具有 `int` 类型元素的数组进行交换。接着，通过将数组的基本类型替换为类型参数 `T`，我们创建了一个模板。这是编写函数模板的一个很好的策略。如果你想写一个函数模板，首先写一个根本不是模板的版本，这个版本的函数只是一个普通的函数。接着对这个普通函数的代码进行全面的 debug，最终通过类型参数替换一些类型名，把这个普通函数转换为一个模板函数。这么做有两个好处。首先，当你定义一个普通函数时，你正在处理一个非常具体的案例，这样的话问题比较容易显现。其次，你需要检查的关于语法的具体细节会大大减少；这时候你只需关注算法本身，不必关注模板语法规则。■

## 示例：一个通用的排序函数

第 5 章给出了一个对数据类型为 `int` 的数组进行选择排序的例子。作为一个函数排序算法，示例 5.8 给出了该算法的 C++ 实现代码。接下来，我们再来回顾一下该排序函数的定义。

```
void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
        // 在 a[index] 中放入正确的值：
        indexOfNextSmallest =
            indexOfSmallest(a, index, numberUsed);
    swapValues(a[index], a[indexOfNextSmallest]);
    // a[0] <= a[1] <= ... <= a[index] 是原始数组中元素值最小的一部分，
    // 其余元素在数组中原来的位置处。
}
}
```

如果你仔细研究上面函数 `sort` 的定义，就会发现函数定义的主要部分与数组类型没有任何关系。如果我们在函数头中用数据类型 `double` 替换了类型 `int`，我们将会得到一个可以适用于类型 `double` 的数组排序算法。当然，我们也必须调整对应的内部调用函数，使它可以适用于元素类型为 `double` 的数组。我们再来看看在 `sort` 函数体内调用的函数。这两个帮助函数是 `swapValues` 和 `indexOfSmallest`。

正如示例 16.1 所示，倘若我们把 `swapValues` 定义为一个函数模板，可以看到它可以适用于任何类型的变量（只要这种数据类型支持赋值操作）。我们再来看看函数 `indexOfSmallest` 是否对待排序数组的基本类型有一定程度的依赖性。为了便于研究该函数的具体细节，我们再次将函数代码呈现如下：

```
int indexOfSmallest(const int a[], int startIndex, int numberUsed)
{
    int min = a[startIndex],
        indexOfMin = startIndex;
    for (int index = startIndex + 1; index < numberUsed; index++)
        if (a[index] < min)
```

```

    {
        min = a[index];
        indexOfMin = index;
        //min 是从数组 a[startIndex] 到 a[index] 中的最小值。
    }
    return indexOfMin;
}

```

可以看出函数 `indexOfSmallest` 对待排序的数组类型同样没有什么依赖性。如果我们将上述代码中所强调的两处对应数组数据类型的定义 `int` 替换为 `double`，那么函数 `indexOfSmallest` 就可以用于对元素类型为 `double` 的数组进行排序了。

为了使函数 `sort` 能够对元素类型为 `double` 的数组进行排序，我们只需要把几个类型名 `int` 替换为类型名 `double`。除此之外，其他没什么需要做的。采用类似的替换方法，我们也可以使 `sort` 函数能够对其他数据类型的数组进行排序。唯一需要确认的是这个数据类型是否支持赋值操作及支持比较运算符“<”，只要满足这两个条件即可对这种数据类型的数组进行排序。这是使用函数模板的理想情况。如果使用类型参数替换函数中几个实例的类型名 `int`（出现在函数 `sort` 和 `indexOfSmallest` 中），那么函数 `sort` 能够对任何类型的数组进行排序，前提是赋值运算符和比较运算符“<”能够操作这种类型的数据值。在示例 16.2 中，我们给出了这个函数模板。

注意，在示例 16.2 中，函数模板 `sort` 能够对非数值类型的数组进行排序。样例程序调用了函数模板 `sort` 来对字符型数组进行排序。比较运算符“<”可以对字符进行操作，它是根据字符在 ASCII 码中的数值（见附录 C）顺序进行比较的。因此，当对两个大写字母进行比较时，运算符“<”判断在字母表顺序中是否第一个字符出现在第二个字符之前。同样，当比较两个小写字母时，运算符“<”判断在字母表顺序中是否第一个字符出现在第二个字符之前。当对大小写字母进行混合比较时，就不像前面那种比较那么容易了，但示例 16.2 只是对两个大写字母进行了比较。在这个程序中，通过调用函数模板 `sort` 对大写字母按照字母表中出现的先后顺序进行调用（只要你重载了比较运算符“<”使得它可以操作该类的对象，函数模板 `sort` 甚至可以对一个自定义的类对象数组进行排序）。

通过将排序函数的定义放到文件 `sort.cpp`（示例 16.3）中，我们对所编写的通用排序函数的实现与声明进行了分离。但是，通常情况下，大部分编译器禁止对模板的实现和声明进行分离。因此，从编程者的角度来看，这么做分离了实现和声明，但对于编译器来说，它会认为实现和声明在同一个文件中。文件 `sort.cpp` 通过 `#include` 引入到我们的程序入口文件中，因此，看上去实现和声明还是在一个文件中。注意，引入文件 `sort.cpp` 的 `include` 指令必须放在任何调用模板函数的代码之前。对于大多数编译器来说，这是让模板可以正常工作的唯一方式。

## 示例 16.2 通用的排序函数

```

1 // 演示一个模板函数的实现。
2 // 一个通用的选择排序算法的模板。
3 #include <iostream>
4 using std::cout;
5 using std::endl;

6 template <class T>
7 void sort(T a[], int numberUsed);
8 // 前提条件: numberUsed <= 数组 a 声明的大小。
9 // 从数组元素 a[0] 到 a[numberUsed - 1] 都有值。
10 // 赋值运算符和比较运算符 < 都可对类型为 T 的值进行操作。
11 // 运行结果: 数组元素 a[0] 到 a[numberUsed - 1] 已经被重新放置,
12 // 放置结果是 a[0] <= a[1] <= ... <= a[numberUsed - 1]。

13 template <class T>
14 void swapValues(T& variable1, T& variable2);
15 // 交换 variable1 和 variable2 的值。
16 // 赋值运算符能够对类型 T 进行操作。

17 template <class T>
18 int indexOfSmallest(const T a[], int startIndex, int numberUsed);
19 // 前提条件: 0 <= startIndex < numberUsed。数组的各元素都有值。
20 // 赋值和 < 运算符可以操作类型 T 的值。
21 // 返回数组 a[startIndex], a[startIndex + 1], ..., a[numberUsed - 1] 中的
22 // 序号 i 时, a[i] 是数组中的最小值。

23 #include "sort.cpp" ←————— 这等价于将这个函数的定义放置在此处。

24 int main( )
25 {
26     int i;
27     int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
28     cout << "Unsorted integers:\n";
29     for (i = 0; i < 10; i++)
30         cout << a[i] << " ";
31     cout << endl;
32     sort(a, 10);
33     cout << "In sorted order the integers are:\n";
34     for (i = 0; i < 10; i++)
35         cout << a[i] << " ";
36     cout << endl;
37     double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
38     cout << "Unsorted doubles:\n";
39     for (i = 0; i < 5; i++)
40         cout << b[i] << " ";
41     cout << endl;
42     sort(b, 5);
43     cout << "In sorted order the doubles are:\n";
44     for (i = 0; i < 5; i++)
45         cout << b[i] << " ";
46     cout << endl;

47     char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};

```

```

48     cout << "Unsorted characters:\n";
49     for (i = 0; i < 7; i++)
50         cout << c[i] << " ";
51     cout << endl;
52     sort(c, 7);
53     cout << "In sorted order the characters are:\n";
54     for (i = 0; i < 7; i++)
55         cout << c[i] << " ";
56     cout << endl;

57     return 0;
58 }

```

### 示例运行结果

```

Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R

```

### 示例 16.3 通用排序函数的实现

```

1  // 下面是文件 sort.cpp 的代码。

2  template <class T>
3  void sort(T a[], int numberUsed)
4  {
5      int indexOfNextSmallest;
6      for (int index = 0; index < numberUsed - 1; index++)
7          // 在 a[index] 中放入正确的值:
8          indexOfNextSmallest =
9              indexOfSmallest(a, index, numberUsed);
10         swapValues(a[index], a[indexOfNextSmallest]);
11         // a[0] <= a[1] <= ... <= a[index] 是原始数组中的值最小的一部分数组元素。
12         // 其余元素在原来的位置。
13     }
14 }

15 template <class T>
16 void swapValues(T& variable1, T& variable2)
17     < 示例 16.1 给出了函数 swapValues 其余部分的定义。 >
18     template < class T>
19     int indexOfSmallest(const T a[], int startIndex, int numberUsed)
20     {
21         // 注意，类型参数可能会被用于函数定义体内。

```



```

21     int indexOfMin = startIndex;

22     for (int index = startIndex + 1; index < numberUsed; index++)
23         if (a[index] < min)
24         {
25             min = a[index];
26             indexOfMin = index;
27             //min 是 a[startIndex] 到 a[index] 中的最小值。
28         }
29     return indexOfMin;
30 }

```

### 陷阱：使用模板时误用了不正确的数据类型

只要我们所使用的数据类型在模板函数的定义中是合理的，就可以使用这种数据类型来调用模板函数。这要求模板函数中的代码必须很清晰并且能正确运行。例如，我们不能用一个对于赋值运算符来说根本无法工作或者无法正确工作的类型参数来作为 swapValues 模板（示例 16.1）的输入参数。

举一个更具体的例子，假设你的程序定义了示例 16.1 所示的模板函数 swapValues，你不能把下列代码添加到你的程序中：

```

int a[10], b[10];
<一些填充数据元素的代码>
swapValues(a, b);

```

上述这段代码无法正常运行，因为赋值运算符无法操作这种数据类型的元素。■

### 自测练习题

5. 示例 5.6 给出了一个函数名为 search 的函数，该函数按顺序查找数组中是否存在一个指定的整数。给出函数 search 的模板函数，使得它可以对任何类型的数组元素进行搜索。给出模板函数声明和定义。（提示：模板函数和示例 5.6 给出的函数几乎相同。）
6. 请比较函数重载和函数模板之间的异同。
7. (该练习仅针对那些至少阅读过第 6 章结构体和类及第 8 章重载运算符的读者) 请问能否使用示例 16.3 中的排序模板函数对示例 6.4 所定义的 DayOfYear 数据类型进行操作？
8. (本练习仅针对那些阅读过第 10 章指针和动态数组的读者。)

尽管赋值运算符不能用于普通的数组变量的相互赋值，但它的确可用于指向动态数组的指针变量间的赋值操作。假设你的程序定义了模板函数 swapValues（如示例 16.1 所示）并且包含如下代码。请问下列代码的输出结果是什么？

```

typedef int* ArrayPointer;
ArrayPointer a, b, c;
a = new int [3];
b = new int [3];

```

```

int i;
for (i = 0; i < 3; i++)
{
    a[i] = i;
    b[i] = i * 100;
}
c = a;
cout << "a contains: ";
for (i = 0; i < 3; i++)
    cout << a[i] << " ";
cout << endl;
cout << "b contains: ";
for (i = 0; i < 3; i++)
    cout << b[i] << " ";
cout << endl;
cout << "c contains: ";
for (i = 0; i < 3; i++)
    cout << c[i] << " ";
cout << endl;

swapValues(a, b);
b[0] = 42;

cout << "After swapping a and b,\n"
    << "and changing b:\n";
cout << "a contains: ";
for (i = 0; i < 3; i++)
    cout << a[i] << " ";
cout << endl;
cout << "b contains: ";
for (i = 0; i < 3; i++)
    cout << b[i] << " ";
cout << endl;
cout << "c contains: ";
for (i = 0; i < 3; i++)
    cout << c[i] << " ";
cout << endl;

```

## 16.2 类模板

在文化上我们有均等的财富和均等的机会……这使得我们所有的人都成为同一类人。

爱德华·贝拉米，《回顾：2000-1887》

从前面所讨论的内容我们可以看到，通过使用模板定义的函数会使函数更具通用性。在本节，我们将会学习到模板可能也会使得类的定义更具通用性。

## 类模板语法

类模板的具体语法基本上与函数模板的具体语法相同。将下列代码放置在类模板定义之前：

```
template <class T>
```

类型参数

像其他所有类型定义一样，类型参数  $T$  被用于类定义。正如在函数模板中的使用方式一样，类型参数  $T$  可能代表了任意一种数据类型；不一定非要用类数据类型来替换类型参数。尽管一般惯例都是使用  $T$  来作为标识符，但正如函数模板中所介绍的那样，在类模板中也可以使用任何非 C++ 关键字的标识符来替换  $T$ 。

示例 16.4 给出了一个类模板的例子。这个类的一个对象包含一对类型为  $T$  的值：如果  $T$  是 `int` 类型，那么对象的值就是一对整数。如果  $T$  是 `char` 类型，那么对象的值就是一对字符。对于其他类型，也是如此<sup>2</sup>。

声明对象

一旦定义了类模板，我们就可以声明这个类的对象了。声明必须指出类型参数  $T$  的具体数据类型。例如，下列代码声明了一个被称为 `score` 的对象，该对象可以用来记录一对整数，下列代码也声明了对象 `seats`，该对象可以用来记录一对字符。

```
Pair<int> score;
Pair<char> seats;
```

接下来，模板对象的使用方式与其他普通对象的使用方式就没什么区别了。例如，下列代码中对象 `score` 的第一个值被设为属性 3，第二个值被设为 0：

```
score.setFirst(3);
score.setSecond(0);
```

### 示例 16.4 类模板定义

```
1 // 该类包含两个数据类型为 T 的值。
2 template <class T>
3 class Pair
4 {
5 public :
6     Pair( );
7     Pair(T firstValue, T secondValue);
8     void setFirst(T newValue);
9     void setSecond(T newValue);
10    T getFirst( ) const ;
11    T getSecond( ) const ;
12 private :
13    T first;
14    T second;
15 };
16 template <class T>
17 Pair<T>::Pair(T firstValue, T secondValue)
18 {
```

<sup>2</sup> `Pair` 是示例 8.6 所给出的类 `intPair` 的模板类。但是，考虑到 `intPair` 里的相关代码并不适用于所有类，因此，在模板类 `Pair` 中，我们去掉了递增和递减运算符。

```

19     first = firstValue;
20     second = secondValue;
21 }

22 template <class T>                                此处并未显示出所有的成员函数。
23 void Pair<T>::setFirst(T newValue)
24 {
25     first = newValue;
26 }

27 template <class T>
28 T Pair<T>::getFirst( ) const
29 {
30     return first;
31 }

```

## 定义成员函数

类模板的成员函数的定义方法与普通类的成员函数的定义方法是一样的。唯一的区别是类模板的成员函数的定义本身就是模板函数。例如，示例 16.4 给出了模板类 `Pair` 的成员函数 `setFirst` 和 `getFirst` 的恰当定义，也给出了具有两个输入参数的构造函数的定义。注意，在作用域标识符之前的类名是 `Pair<T>` 而不是简单的 `Pair`。但是，作用域标识符之后的构造函数名就是简单的 `Pair`，不带 `<T>`。

## 作为参数的类模板

类模板的名字可以用来作为函数的输入参数。例如，下列代码是一个可行的函数声明，它声明了一个输入参数代表一对整数的函数：

```

int addUp(const Pair<int>& thePair);
// 返回 thePair 中的两个整数之和。

```

请注意，我们在本例中使用了类型 `int` 来作为类型参数 `T` 的具体类型。

我们甚至可以在一个函数模板中使用类模板。例如，我们可以不用定义上面代码中所示的特殊函数 `addUp`，可以定义一个如下面代码所示的函数模板，使得这个函数可应用于各种数据类型的数值：

```

template <class T>
T addUp(const Pair<T>& thePair);
// 前提条件：运算符 + 可以操作类型为 T 的值。
// 返回 thePair 中所输入的两个值之和。

```

## 对类型参数的限制

几乎所有的模板类定义都会对选取什么样的数据类型来合理地替代一个或多个类型参数进行一些限制。即使最简单易懂的模板类如 `Pair` 也不能绝对保证可适用于所有数据类型 `T`。除非所应用的数据类型支持赋值运算符和拷贝构造函数，否则类型 `Pair<T>` 也不能正常使用，这是因为模板类中成员函数的定义使用了赋值运算符，并且模板类中包含一个以类型 `T` 作为传值参数的成员函数。如果 `T` 中包括指针和动态变量，那么 `T` 也应当有一个合适的析构函数。但是，这些要求只是期望替换了类数据类型 `T` 后代码能正常执行。因此，这些要求是最基本的要求。对于其他模板类，在可替换的数据类型方面，可能会有更多的限制条件。

## 类模板语法

类模板和其成员函数的定义以如下代码开始：

```
template <class Type_Parameter >
```

模板类和成员函数的定义除了可以使用一个数据类型来替换 `Type_Parameter` 外，其余与普通的类定义是一样的。

例如，下列代码展示了类模板定义的前几行。

```
template <class T>
class Pair
{
public:
    Pair( );
    Pair(T firstValue, T secondValue);
    ...
}
```

接下来，把成员函数和重载运算符定义为函数模板。例如，前面模板样例中带有两个参数的构造函数的定义的前几行代码如下所示：

```
template <class T>
Pair<T>::Pair(T firstValue, T secondValue)
{
    ...
}
```

我们可以通过将一个类型参数作为类名来实例化一个类模板，从而得到一个具体的类，如下列代码所示：

```
Pair<int>
```

接下来，就可以像使用任何一个普通的类名一样来使用这个具体的类名 `Pair<int>` 了。我们可以用它来声明对象或者把它作为一个形参数据类型。

## 类型定义

可以通过类型定义符 `typedef` 定义一个等价于具体类模板名的新的类类型名，如 `Pair<int>`。这种定义类类型名的语法如下：

```
typedef Class_Name<Type_Argument> New_Type_Name;
```

例如，

```
typedef Pair<int> PairOfInt;
```

接下来，类型名 `PairOfInt` 就可以用来声明类型为 `Pair<int>` 的对象了，如下所示：

```
PairOfInt pair1, pair2;
```

类型名 `PairOfInt` 也可以被用来作为一个形参的数据类型或者用于其他任何类型名可以被使用的地方。

## 自测练习题

- 给出示例 16.4 类模板 `Pair` 的默认构造函数（不带输入参数）的定义。

10. 给出下列函数的完整定义，该函数在前面讨论过：

```
int addUp(const Pair<int>& thePair);
// 返回 thePair 中的两个整数之和。
```

11. 给出下列模板函数的完整定义，该函数在前面讨论过：

```
template <class T>
T addUp(const Pair<T>& thePair);
// 前提条件：运算符“+”可以操作类型为 T 的变量的值。
// 返回 thePair 中的两个值之和。
```

### 示例：一个数组模板类

在第 10 章中，我们定义了一个元素数据类型为 double 的部分填充数组类（示例 10.10 和示例 10.11）。在本例中，我们将该类转换成适用于任意数据类型的一部分填充数组模板类。模板类 PFArray 有一个类型参数 T，它表示数组元素的数据类型。

还是按照我们的老规矩进行转换。我们只需要把 double（如果 double 是数组元素的数据类型）替换为类型参数 T 并且把类定义和成员函数的定义转换成模板形式。示例 16.5 给出了模板类的定义，示例 16.6 给出了成员函数模板的定义。

注意，我们已经将模板定义放到了一个命名空间中。命名空间在模板定义中的使用方式和在一些简单的、非模板的定义中的使用方式是一样的。

示例 16.7 给出了一个应用程序样例。注意，我们将类模板的接口、实现和调用它的应用程序分别放置在三个不同的文件中。但是很不幸，这些文件不能像传统的编译方式那样进行独立编译。大部分编译器不能兼容这种独立的编译方式。因此，我们只有尽力做到将接口和实现文件通过 #include 引入到应用程序的文件中。在编译器看来，好像三个文件的一切代码都在一个文件中一样。

### 示例 16.5 模板类 PFArray 的接口

```
1 // 这是头文件 pffarray.h。这也是类 PFArray 的接口。
2 // 这种数据类型对象是用基本类型 T 进行部分填充后的一个数组。
3 #ifndef PFARRAY_H
4 #define PFARRAY_H
5
6 namespace PFArraySavitch
7 {
8     class PFArray
9     {
10     public:
11         PFArray( ); // 把数组大小初始化为 50。
12
13         PFArray(int capacityValue);
14
15         PFArray(const PFArray<T>& pfaObject);
```

```

15      // 前提条件：数组中存在没有值的元素。
16      // 运行结果：数组元素添加成功。
17      bool full( ) const; // 如果数组的所有元素都有值，结果返回为 true，
                          // 否则返回 false。

18      int getCapacity( ) const;

19      int getNumberUsed( ) const;

20      void emptyArray( );
21      // 把数组大小重置为 0，从而有效地清空数组。

22      T& operator[](int index);
23      // 对数组中序号从 0 到 numberUsed - 1 的元素进行读和修改值操作。

24      PFArray<T>& operator =(const PFArray<T>& rightSide);

25      virtual ~PFArray( );
26      private :
27          T *a; // 定义数组 T。
28          int capacity; // 定义数组大小。
29          int used; // 定义当前数组中有值的元素的个数。
30      };
31  }// PFArraySavitch
32 #endif //PFARRAY_H

```

---

## 示例 16.6 PFArray 模板类的实现

---

```

1  // 这是模板类的实现文件：pfarray.cpp。
2  // 这是模板类 PFArray 的实现代码。
3  // 模板类 PFArray 的接口定义在文件 pfarray.h 中。

4  #include "pfarray.h"
5  #include <iostream>
6  using std::cout;

7  namespace PFArraySavitch
8  {
9      template <class T>
10     PFArray<T>::PFArray( ) :capacity(50), used(0)
11     {
12         a = new T[capacity];
13     }

14     template <class T>
15     PFArray<T>::PFArray(int size) :capacity(size), used(0)
16     {
17         a = new T[capacity];
18     }

19     template <class T>
20     PFArray<T>::PFArray(const PFArray<T>& pfaObject)
21         :capacity(pfaObject.getCapacity( )),

```

注意，在作用域运算符之前使用了 T，而在模板构造函数名中未使用 T。

```

        used(pfaObject.getNumberUsed( ))
22     {
23         a = new T[capacity];
24         for (int i = 0; i < used; i++)
25             a[i] = pfaObject.a[i];
26     }
27
28     template <class T>
29     void PFArrary<T>::addElement( const T& element)
30     {
31         if (used >= capacity)
32         {
33             cout << "Attempt to exceed capacity in PFArrary.\n";
34             exit(0);
35         }
36         a[used] = element;
37         used++;
38     }
39
40     template <class T>
41     bool PFArrary<T>::full( ) const
42     {
43         return (capacity == used);
44     }
45
46     template <class T>
47     int PFArrary<T>::getCapacity( ) const
48     {
49         return capacity;
50     }
51
52     template <class T>
53     int PFArrary<T>::getNumberUsed( ) const
54     {
55         return used;
56     }
57
58     template <class T>
59     void PFArrary<T>::emptyArray( )
60     {
61         used = 0;
62     }
63
64     template <class T>
65     T& PFArrary<T>::operator[](int index)
66     {
67         if (index >= used)
68         {
69             cout << "Illegal index in PFArrary.\n";
70             exit(0);
71         }
72         return a[index];
73     }
74
75     template <class T>

```



```

70     PFArray<T>& PFArray<T>::operator =(const PFArray<T>& rightSide)
71     {
72         if (capacity != rightSide.capacity)
73         {
74             delete [] a;
75             a = new T [rightSide.capacity];
76         }
77
78         capacity = rightSide.capacity;
79         used = rightSide.used;
80         for (int i = 0; i < used; i++)
81             a[i] = rightSide.a[i];
82
83         return *this ;
84     }
85
86     template <class T>
87     PFArray<T>::~~PFArray( )
88     {
89         delete [] a;
90     }
91 } // PFArraySavitch

```

---

### 示例 16.7 模板类 PFArray 演示程序

---

```

1  // 演示模板类 PFArray 的样例程序。
2  #include <iostream>
3  #include <string>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7  using std::string;
8
9  #include "pfarray.h"
10 #include "pfarray.cpp"
11 using PFArraySavitch::PFArray;
12
13 int main( )
14 {
15     PFArray<int> a(10);
16
17     cout << "Enter up to 10 nonnegative integers.\n";
18     cout << "Place a negative number at the end.\n";
19     int next;
20     cin >> next;
21     while ((next >= 0) && (!a.full( )))
22     {
23         a.addElement(next);
24         cin >> next;
25     }
26     if (next >= 0)
27     {
28         cout << "Could not read all numbers.\n";

```

```

26         // 清除没有读入的输入值。
27         while (next >= 0)
28             cin >> next;
29     }

30     cout << "You entered the following:\n ";
31     int index;
32     int count = a.getNumberUsed( );
33     for (index = 0; index < count; index++)
34         cout << a[index] << " ";
35     cout << endl;
36     PFArray<string> b(3);

37     cout << "Enter three words:\n";
38     string nextWord;
39     for (index = 0; index < 3; index++)
40     {
41         cin >> nextWord;
42         b.addElement(nextWord);
43     }

44     cout << "You wrote the following:\n";
45     count = b.getNumberUsed( );
46     for (index = 0; index < count; index++)
47         cout << b[index] << " ";
48     cout << endl;
49     cout << "I hope you really mean it.\n";

50     return 0;
51 }

```

### 示例运行结果

```

Enter up to 10 nonnegative integers.
Place a negative number at the end.
1 2 3 4 5 -1
You entered the following:
1 2 3 4 5
Enter three words:
I love you
You wrote the following:
I love you
I hope you really mean it

```

### 友元函数

模板类中友元函数的使用方法与普通类中友元函数的使用方法基本相同。唯一的区别是在模板类中使用友元函数必须要包含一个合适的类型参数。

## 自测练习题

12. 使下列函数成为示例 16.5 所给出的模板类 PFArray 的友元函数, 必须要做哪些工作?

```
void showData(PFArray<T> theObject);
// 把对象 theObject 中的数据 displayed 到屏幕上。
// 假设程序已经为类型为 T 的值定义了运算符 "<<"。
```

## 模板中的 vector 和 basic\_string

如果你还没有学习和使用过模板类 vector 和 basic\_string, 那么现在最好学习一下 7.3 节的内容, 那一节讲述了关于模板类 vector 的一些知识。

另一个预定义模板类是 basic\_string 模板类。这个类能够将任意类型的元素转换为字符串。类 basic\_string<char> 是能将字符类型转换为字符串的类。类 basic\_string<double> 是能将类型 double 的数值转换为字符串的类。类 basic\_string<YourClass> 是将类型 YourClass (该类可以为任意类) 转换为字符串的类。

其实我们一直在使用模板类 basic\_string 的一个特例, 这个特例就是 string, 它是类 basic\_string<char> 的别名。我们所学过的类 string 的所有成员函数都适用于模板类 basic\_string<T>。

模板类 basic\_string 是在头文件 <string> 的库中定义的, 该定义位于 std 命名空间中。当我们使用类 basic\_string 时, 需要在文件前面部分加入如下类似代码段:

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

或者

```
#include <string>
using std::basic_string;
using std::string; // 程序中仅仅使用 string 类。
```

## 16.3 模板和继承

任何一个时代的统治思想始终都不过是统治阶级的思想。

卡尔·马克思, 弗里德里希·恩格斯, 《共产党宣言》

对于模板和继承来说, 我们基本上不需要再学习什么新知识了。为了定义一个派生模板类, 先定义一个模板类 (或者是一个非模板类), 接着从所定义的这个类派生一个模板类。这和从一个普通的基类派生一个派生类的方法是一样的。我们用一个例子来说明在派生模板类的过程中遇到的所有语法细节问题。

### 示例：带备份的部分填充数组模板类

在第 14 章（示例 14.10 和示例 14.11）中，我们定义了带备份的部分填充 `double` 类型数组类 `PFArrayDBak`。我们把它定义为类 `PFArrayD`（示例 14.8 和示例 14.9）的派生类。类 `PFArrayD` 是一个部分填充数组元素的类，但是它仅对基本类型为 `double` 的数组有效。示例 16.5 和示例 16.6 把类 `PFArrayD` 转换成模板类 `PFArray`，使得它可以应用于任意基本数据类型的数组。在接下来的程序中，我们将定义一个带备份的部分填充数组模板类 `PFArrayBak`，它适用于任意基本数据类型的数组。同时，模板类 `PFArrayBak` 也是模板类 `PFArray` 的派生类。定义模板类 `PFArrayBak` 的工作几乎可以自动完成。我们可以将类 `PFArrayDBak` 中所有数组的基本类型 `double` 关键字替换为类型参数 `T`，用类 `PFArray` 替换类 `PFArrayD`，最后再整理一些语句的语法以使其完全符合模板语法。

示例 16.8 给出了模板类 `PFArrayBak` 的接口。注意，基类是带数组参数的类 `PFArray<T>`，而不是简单的类 `PFArray`。如果仔细考虑这一点，我们会发现自己确实需要 `<T>`。带备份的、元素类型为 `T` 的部分填充数组类是不带备份的、元素类型为 `T` 的部分填充数组类的派生类。类型参数 `T` 是很重要的，如何使用它也是很重要的。

示例 16.9 给出了模板类 `PFArrayBak` 的实现。在下面的代码中，我们复制了模板类实现中的第一个构造函数的定义：

```
template<class T>
PFArrayBak<T>::PFArrayBak( ) : PFArray<T>( ), usedB(0)
{
    b = new T[getCapacity( )];
}
```

注意，上述代码和任何模板类的定义一样，开头如下：

```
template <class T>
```

同时也要注意，数组的基本类型（`new` 运算符之后）是类型参数 `T`。其他可能还有一些不是十分明显的细节问题，但一定要确保它们是合理的。

接下来考虑下面这行代码：

```
PFArrayBak<T>::PFArrayBak( ) : PFArray<T>( ), usedB(0)
```

和其他任何模板类函数的定义一样，上述代码的作用域运算符之前是带类型参数 `T` 的类的声明 `PFArrayBak<T>`，但是构造函数名则比较简单，是不带类型参数 `T` 的 `PFArrayBak`。同时也要注意，在调用基类构造函数时包含了类型参数 `T`，即 `PFArray<T>( )`。这么做可以使得构造函数满足基类 `PFArray<T>` 接口，下面的代码行展示了这一用法：

```
class PFArrayBak : public PFArray<T>
```

示例 16.10 给出了一个关于模板类 `PFArrayBak` 的样例程序。

### 示例 16.8 模板类 PFArrayBak 的接口

---

```

1 // 这是头文件 pfarraybak.h 的代码。
2 // 这是模板类 PFArrayBak 的接口。
3 // 这种类型的对象是元素类型为 T 的部分填充数组。这个版本的代码允许程序员通过拷贝来备份数组，并且能
4 // 恢复到最后一次保存的部分填充数组。
5 #ifndef PFARRAYBAK_H
6 #define PFARRAYBAK_H
7 #include "pfarray.h"

8 namespace PFArraySavitch
9 {
10     template<class T>
11     class PFArrayBak : public PFArray<T>
12     {
13     public :
14         PFArrayBak( );
15         // 把数组大小初始化为 50。

16         PFArrayBak(int capacityValue);

17         PFArrayBak(const PFArrayBak<T>& Object);

18         void backup( );
19         // 备份部分填充数组。

20         void restore( );
21         // 把部分填充数组恢复到最后一次保存的版本。
22         // 如果从未调用过部分填充数组，下面的代码将清空部分填充数组。

23         PFArrayBak<T>& operator =(const PFArrayBak<T>& rightSide);
24         virtual ~PFArrayBak( );
25     private :
26         T *b; // 备份主数组。
27         int usedB; // 备份继承的成员变量 used。
28     };

29 } // PFArraySavitch
30 #endif //PFARRAYBAK_H

```

---

### 示例 16.9 模板类 PFArrayBak 的实现

---

```

1 // 这是文件 pfarraybak.cpp 的代码清单。
2 // 这是模板类 PFArrayBak 的实现代码。
3 // 模板类 PFArrayBak 的接口在文件 pfarraybak.h 中。
4 #include "pfarraybak.h"
5 #include <iostream>
6 using std::cout;

7 namespace PFArraySavitch
8 {

```

```

9      template <class T>
10      PFArrayBak<T>::PFArrayBak( ) : PFArray<T>{ }, usedB(0)
11      {
12          b = new T[getCapacity( )];
13      }

14      template <class T>
15      PFArrayBak<T>::PFArrayBak(int capacityValue)
16          : PFArray<T>(capacityValue), usedB(0)
17      {
18          b = new T[getCapacity( )];
19      }

20      template <class T>
21      PFArrayBak<T>::PFArrayBak(const PFArrayBak<T>& oldObject)
22          : PFArray<T>(oldObject), usedB(0)
23      {
24          b = new T[getCapacity( )];
25          usedB = oldObject.getNumberUsed();
26          for (int i = 0; i < usedB; i++)
27              b[i] = oldObject.b[i];
28      }

29      template <class T>
30      void PFArrayBak<T>::backup( )
31      {
32          usedB = getNumberUsed( );
33          for (int i = 0; i < usedB; i++)
34              b[i] = operator[](i);
35      }

36      template <class T>
37      void PFArrayBak<T>::restore( )
38      {
39          emptyArray( );

40          for (int i = 0; i < usedB; i++)
41              addElement(b[i]);
42      }

43      template<class T>
44      PFArrayBak<T>& PFArrayBak<T>::operator =
45          (const PFArrayBak<T>& rightSide)
46      {
47          PFArray<T>:: operator =(rightSide);

48          if (getCapacity( ) != rightSide.getCapacity( ))
49          {
50              delete [] b;
51              b = new T[rightSide.getCapacity( )];
52          }
53          usedB = rightSide.usedB;
54          for (int i = 0; i < usedB; i++)
55              b[i] = rightSide.b[i];

56          return *this ;

```

```

56     }

57     template<class T>
58     PFArrayBak<T>::~PFArrayBak( )
59     {
60         delete [] b;
61     }
62 } // PFArraySavitch

```

---

### 示例 16.10 模板类 PFArrayBak 的演示程序

---

```

1  // 演示模板类 PFArrayBak 的程序。
2  #include <iostream>
3  #include <string>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7  using std::string;

8  #include "pfarraybak.h"
9  #include "pfarray.cpp"
10 #include "pfarraybak.cpp"
11 using PFArraySavitch::PFArrayBak;

12 int main( )
13 {
14     int cap;
15     cout << "Enter capacity of this super array: ";
16     cin >> cap;
17     PFArrayBak<string> a(cap);

18     cout << "Enter " << cap << " strings\n";
19     cout << "separated by blanks.\n";

20     string next;
21     for (int i = 0; i < cap; i++)
22     {
23         cin >> next;
24         a.addElement(next);
25     }
26     int count = a.getNumberUsed( );
27     cout << "The following " << count
28         << " strings read and stored:\n";
29     int index;
30     for (index = 0; index < count; index++)
31         cout << a[index] << " ";
32     cout << endl;

33     cout << "Backing up array.\n";
34     a.backup( );
35     cout << "Emptying array.\n";
36     a.emptyArray( );

```

不要忘了 include 基类模板的实现。

```

37     cout << a.getNumberUsed( )
38         << " strings are now stored in the array.\n";
39     cout << "Restoring array.\n";
40     a.restore( );
41     count = a.getNumberUsed( );
42     cout << "The following " << count
43         << " strings are now stored:\n";
44     for (index = 0; index < count; index++)
45         cout << a[index] << " ";
46     cout << endl;

47     cout << "End of demonstration.\n";
48     return 0;
49 }

```

### 示例运行结果

```

Enter capacity of this super array: 3
Enter 3 strings
separated by blanks.
I love you
The following 3 strings read and stored:
I love you
Backing up array.
Emptying array.
0 strings are now stored in the array.
Restoring array.
The following 3 strings are now stored:
I love you
End of demonstration.

```

### 自测练习题

13. 以如下代码开始来表示一个派生模板类是否合法？模板类 `TwoDimPFArryBak` 表示的是一个带备份的二维部分填充数组。

```

template<class T>
class TwoDimPFArryBak : public PFArry< PFArry<T> >
{
public:
    TwoDimPFArryBak( );

```

注意，在 `< PFArry<T> >` 中的空格是很重要的，或者说至少最后一个空格是重要的。如果忘记了倒数第二个 “>” 和最后一个 “>” 之间的空格，那么编译器会将这两个符号解释为表达式中用于输入的提取运算符 “>>”，如 `cin>>n;`，而不会把它解释为一个嵌套的 “< >”。

14. 写出自测练习题 13 中给出的类 `TwoDimPFArryBak` 的默认构造函数（没有输入参数的构造函数）实现代码的开头部分（假设所有实例变量在构造函数定义体内被初始化，因此你不需要考虑去做这些变量的初始化工作）。



## 本章小结

- 通过使用函数模板,可以定义带有类型参数的函数。
- 通过使用类模板,可以定义一个带类型参数的类,这个类可以作为其他类的一部分。
- 预定义类 `vector` 和 `basic_string` 实际上是模板类。
- 可以通过模板基类定义一个派生模板类。

## 自测练习题答案

### 1. 函数声明:

```
template<class T>
T maximum(T first, T second);
// 前提条件: 数据类型 T 支持运算符 "<".
// 返回 first 和 second 中的较大值。
```

### 函数定义:

```
template<class T>
T maximum(T first, T second)
{
    if (first < second)
        return second;
    else
        return first;
}
```

### 2. 函数声明:

```
template<class T>
T absolute(T value);
// 前提条件: 无论 x 是什么类型, 表达式  $x < 0$  和  $-x$  都是合法的。
// 返回输入参数的绝对值。
```

### 定义:

```
template<class T>
T absolute(T value)
{
    if (value < 0)
        return -value;
    else
        return value;
}
```

### 3. 模板的作用是允许对带类型参数的函数和类进行定义。

4. e. 任何类型,无论是基本类型(C++所提供的),还是用户自定义的类型(class、struct、enum、数组类型,或者int、float、double等数据类型),代码中的参数类型T必须符合语法规范。例如,对于模板函数swapValues(示例

- 16.1), 类型 T 对于赋值运算符必须有效。即赋值运算符可以操作类型 T 的数据。
5. 下列代码给出了函数声明和函数定义。它们与示例 5.6 给出的代码版本基本相同 (除了将参数列表中的两个实例的类型从 int 改为 T)。

函数声明：

```
template<class T>
int search(const T a[], int numberUsed, T target);
// 前提条件：变量 numberUsed 小于或等于所声明的数组 a 的大小。
// 并且，从元素 a[0] 到 a[numberUsed - 1] 都有值。
// 如果数组中存在索引为 index 的元素使得 a[index] == target,
// 则返回最小的那个 index；否则，函数返回 -1。
```

函数定义：

```
template<class T>
int search(const T a[], int numberUsed, T target)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < numberUsed))
        if (target == a[index])
            found = true;
    else
        index++;

    if (found)
        return index;
    else
        return -1;
}
```

6. 进行函数调用时，函数重载仅对于那些支持重载的数据类型有效。(尽管重载可能支持将一种数据类型自动转换为另外某种支持重载的数据类型，但有可能转换后的这种数据类型并不是你所需要的数据类型)。而如果采用模板这种方案，它对于函数调用时所输入的任意数据类型都是可以工作的，当然，前提是所输入的数据类型对于模板的函数体来说是合法的。
7. 不可以，模板函数 sort 不能对数据类型为 DayOfYear 的数组进行排序，这是因为运算符“<”不适用于数据类型 DayOfYear。(正如我们在第 8 章所讨论的那样，如果针对数据类型 DayOfYear 重载运算符“<”，使得该运算符可以对这种数据类型进行前后顺序的比较，那么就可以通过模板函数 sort 对这种数据类型进行排序。例如，我们可以重载“<”，使得它前后的数据依照日历上的顺序给出，那么接下来就可以对这种数据类型按日历顺序进行输出。)

8. a contains: 0 1 2  
 b contains: 0 100 200  
 c contains: 0 1 2  
 After swapping a and b,  
 and changing b:  
 a contains: 0 100 200  
 b contains: 42 1 2

c contains: 42 1 2

注意，在代码 `swapValues(a, b);` 执行之前，c 是数组 a 的一个别名（另一个名字）。在代码 `swapValues(a, b);` 执行之后，c 是数组 b 的一个别名。尽管从某种意义上来说，我们交换了 a 和 b 的值，但实际情况并不如我们所想象的那般简单。由于使用了指针变量，在使用 `swapValues` 时，可能会带来一些副作用。

在此，需要进行一些说明，赋值运算符对数组指针的操作或许并不像你所想象的那样。因此，模板函数 `swapValues` 对于数组指针变量的操作也并不像你所想象的那样。赋值运算符不会对两个数组中的元素进行一个一个地交换，而是仅仅将指向两个数组的指针进行交换。因此，以数组指针为输入参数的 `swapValues` 函数仅仅简单地交换了两个指针而已。除非你对带数组指针（或指向其他任何类型的指针）参数的 `swapValues` 函数的工作机制很清楚，否则，最好不要使用它。使用带类型参数 T 的 `swapValues` 模板函数对数组的操作效果和使用赋值运算符对类型 T 进行操作的效果是一样的。

9. 由于类型参数有可能是任意数据类型，因此不存在适用于任意数据类型的默认初始值。故，构造函数什么也不做，但它的确允许你声明（未初始化）不带任何构造函数参数的对象。

```
template<class T>
Pair<T>::Pair()
{
    //Do nothing.
}
```

```
10. int addUp(const Pair<int>& thePair)
{
    return (thePair.getFirst() + thePair.getSecond());
}
```

```
11. template<class T>
T addUp(const Pair<T>& thePair)
{
    return (thePair.getFirst() + thePair.getSecond());
}
```

12. 把下列代码放到 `PFArrray` 模板类定义的 `public` 部分：

```
friend void showData(PFArrray<T> theObject);
// 把 theObject 中的数据 display 到屏幕上。
// 假设已经定义了运算符 "<<", 它可以对类型 T 的值进行操作。
```

有可能你需要增加一个 `showData` 的函数模板定义。下列代码是一种可行的定义：

```
namespace PFArrraySavitch
{
    template<class T>
    void showData(PFArrray<T> theObject)
    {
        for (int i = 0; i < theObject.used; i++)
```

```

        cout << theObject[i] << endl;
    }
} //PFAArraySavitch

```

13. 是的，它完全合法。可能也有其他更好的实现方式，但它的确是合法的，而且一点也不奇怪。

```

14. template <class T>
    TwoDimPFAArrayBak<T>::TwoDimPFAArrayBak( )
        : PFAArray< PFAArray<T> >( )

```

## 编程练习

1. 示例 13.8 给出的迭代二分查找算法只能在一个整数数组中搜索一个整数类型的关键字，根据这个示例，编写一个该算法的模板函数。给出模板函数参数类型的限制条件。讨论对模板参数类型的要求。
2. 根据示例 13.6 编写一个递归二分查找函数模板。指出模板函数参数类型的限制条件。讨论对模板参数类型的要求。
3. 示例 16.3 给出的排序函数模板是基于选择排序算法实现的。另外一个相关的排序算法是插入排序。插入排序算法是在玩桥牌时经常会使用到的一种方法。依次考虑每一个要进行排序的元素，从一个已经排好序的数组的第一个元素处开始进行判断，把每一个要进行排序的元素插入到这个数组的合适位置处。在插入一个元素时，把比这个元素大的那些元素移动到它的右边，为这个元素腾出位置，并把它放在空出来的位置处。例如，下列代码给出了针对整型数组 *a* 的选择排序的具体步骤。每行给出了从数组 *a*[0] 到 *a*[4] 的值。星号标记出了数组中已经排好序的元素和未排好序的元素的边界。

```

2 * 5 3 4
2 5 * 3 4
2 3 5 * 4
2 3 4 5 *

```

首先，编写一个可以对整型数组进行插入排序的函数。接着，把这个排序函数改写成模板函数。最后，使用几种基本类型的数据和自定义数据类型对其进行全面的测试。

4. 编写一个模板函数，它计算并返回所传人的两个数值的绝对值。函数可以操作任何数值数据类型（例如，float、int、double 和 char）。
5. 编写一个基于模板的类来实现项（item）的集合。这个类应当允许用户进行如下操作：
  - a. 向集合中增加一个新的项。
  - b. 获取集合汇总项的个数。
  - c. 获取一个指向动态创建的数组的指针，这个数组里包含集合中的每一个项。函数调用者负责释放所分配的内存。

通过创建不同数据类型的集合（例如，整型和字符串）来测试你所编写的程序。

6. 第 10 章的编程练习 6 给出了一个关于字符串类型动态数组的包装类，在该动态数组中可以动态增加和删除字符串。本编程练习要求编写一个基于模板的类，使得它可以应用于其他类型的数据，而不仅仅受限于字符串类型。除了使用字符串类型的动态数组对所编写的类进行测试外，也请使用整数类型的动态数组进行测试。
7. 在本章中，我们仅仅使用了包含一个类型参数的模板类。但其实 C++ 允许我们指定多个类型参数。例如，下列代码指定类可以接受两个类型参数：

```
template<
class T, class V>
class Example
{
    ...
}
```

当创建该类的实例时，我们必须指定两个数据类型，比如：

```
Example<int, char> demo;
```

修改示例 16.4 所给出的 Pair 类，使得一对数据具有不同的数据类型。编写一个 main 函数来对这个包含两个不同数据类型的类进行测试。

## 17.1 节点和链表 619

节点 619

链表 623

在链表头插入节点 625

陷阱：丢失节点 627

插入和删除链表内的节点 628

陷阱：对动态数据结构使用赋值运算符 631

搜索链表 631

搜索函数伪代码 632

双向链表 633

为双向链表增加一个节点 635

从双向链表中删除一个节点 635

示例：使用双向链表实现的通用排序模板

函数 641

## 17.2 链表的应用 644

示例：栈模板类 644

示例：队列模板类 650

提示：命名空间的注解 652

友元类以及类似的其他选择 653

示例：包含节点链的哈希表 655

字符串哈希函数 656

哈希表的效率 660

示例：集合 (set) 模板类 661

集合的基本操作 662

链表创建的集合的效率 667

## 17.3 迭代器 668

指针作为迭代器 668

迭代器类 669

示例：迭代器类 670

## 17.4 树 676

树的属性 676

示例：树模板类 678

# 第 17 章 链式数据结构

如果那里碰巧有爱我之人，

我的心将会为我认出他；

我将会把他指给你看。

吉尔伯特与沙利文，《恶毒的诅咒》

## 概述

链表就是使用指针构建的一个列表。链表的大小不是固定的，它可以在程序运行过程中动态地增长和减小。树是使用指针构建的另一种数据结构。本章将介绍使用指针如何构建这种数据结构。C++ 标准模板库 (STL) 中已经预定义了这些数据结构以及其他相类似的数据结构。STL 的内容将会在第 19 章介绍。通常情况下，推荐使用 STL 预定义的数据结构版本，而不建议去自定义这些数据结构。但是，在某些情况下，你可能需要使用指针去定义一些自己的数据结构（总得有些人去定义 STL）。此外，通过学习本章内容，你将会对 STL 的定义方式有更深入的认识，本章也会为你提供一些基本的、广泛使用的技术。

链式数据结构是通过动态变量来生成的，这些动态变量是通过 new 运算符创建的。链式数据结构使用指针来把这些动态变量连接起来。通过这种实现方式，程序员可以完全自行控制如何构建和管理数据结构，包括如何管理内存。这种方式可以使我们的程序在一些情况下变得更为高效。例如，将一个值插入到排好序的链表中比插入到一个排好序的数组中更为简单，并且效率更高。

基本上可以通过三种方式来构建本章所讨论的数据结构：

1. 使用 C 语言风格的全局函数和结构体 (struct) 方式，其中，结构体的所有成员都是 public 的。
2. 使用带成员变量和成员函数的类的方式，其中，成员变量都是 private 的，成员函数有取值函数和赋值函数。
3. 使用友元类（或者一些类似的东西，比如 private 或 protected 性质的继承以及内部类）。

对于上述三种使用方式我们都准备了对应的例子来帮助大家理解。我们使用方法 1 来介绍链表这种数据结构。接着，我们进一步讨论了一个基本链表数据结构的一些更加具体的细节知识，并且使用方法 2 介绍了栈和队列这两种数据结构。通过使用友元类（方法 3），我们给出了另外一种队列模板的定义，并使用友元类（方法 3）给出了一个树模板类的定义。通过这种介绍方式，可以使读者看到各种方法的优缺点。我们的建议是尽量使用友元类，但每种方法都有它的拥护者。

从 17.1 节到 17.3 节没有用到 13 章到 15 章的知识（递归、继承和多态），只有一点例外：按照第 15 章的建议，我们使用关键字标识符 virtual 来标记类的析构函数。如果你还没有阅读虚函数的相关内容（在第 15 章），你可以假设代码中没有出现

“virtual”关键字。考虑到学习本章的主要目的,是否有关键字“virtual”并不重要。第 17.4 节使用了递归 (第 13 章),但是没有使用第 14 章和第 15 章的相关知识。

## 17.1 节点和链表

### 动态数据 结构

如示例 17.1 图中所表示的那样,链表是动态数据结构的一个简单例子。它之所以被称为动态数据结构,是因为示例 17.1 图中每一个方盒表示一个 struct 或 class 类型的变量,而这些变量都是通过 new 运算符创建的。在一个动态数据结构中,这些方盒或者说这些节点包含指向其他节点的指针,指针在图上是用箭头表示的。本节将讨论构建和维护链表的一些基本技术。

### 节点

### 节点结构

如示例 17.1 所示,一个结构体是由多个元素构成的,我们把这些元素画成一个一个的方盒,方盒间用箭头连接。这些方盒被称为节点,箭头表示指针。示例 17.1 中的每个节点包含一个字符串、一个整数和一个指针。这个指针指向和它所在节点具有相同类型的另外一个节点。注意,指针指向的是节点,而不是节点内部的元素(比如 10 或者 “rolls”)。

### 节点类型 定义

在 C++ 中,节点是通过结构体或者类来实现的。例如,示例 17.1 中的节点是用结构体来定义的,节点以及指向节点的指针、它们的类型定义如下列代码所示:

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
}

typedef ListNode* ListNodePtr;
```

类型定义的顺序是很重要的。必须先定义 ListNode,因为 ListNodePtr 的定义需要用到它。

示例 17.1 中被标记为 head 的方盒表示的不是一个节点而是一个指向节点的指针变量。指针变量 head 的声明如下:

```
ListNodePtr head;
```

尽管我们对类型的定义在次序上进行了安排,以此来避免出现某些非法的循环定义形式,但上述结构体类型的 ListNode 的定义仍然存在循环定义。在类型 ListNode 的定义中,对成员变量 link 的定义使用了类型名 ListNode。这种特定的循环定义方式没什么错误的,在 C++ 语法规范中是允许的。这个定义在逻辑上并非矛盾,因为你依然可以画一个图来表示这种结构,如示例 17.1 所示。

在示例 17.1 中,我们把指针放到了结构体内部,并让指针指向结构体,而结构体



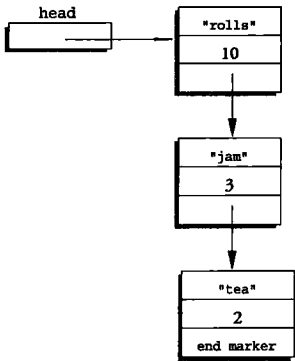
改变节点  
的数据

又包含了指针，如此等等。对于这种情形，有时可能会涉及语法问题。但是几乎在所有情况下，语法并不遵循我们之前所描述的那些为指针和结构体制定的规则。为了解释这一现象，我们假设链表的声明如前面所给出的声明代码所示，关于这种情形的描述，如示例 17.1 的图所示，如果我们想把第一个节点中的数值 10 改为 12。可以采用如下语句来完成这一更改：

```
(*head).count = 12;
```

我们来对赋值运算符左侧的表达式做点解释。变量 head 是一个指针变量。因此，表达式 \*head 是它所指向的节点（动态变量），这个节点包含一个字符串“rolls”和一个整数数值 10。由 \*head 所指向的这个节点是一个结构体，这个结构体包含一个被称为 count 的成员变量，它是一个类型为 int 的数值，那么，(\*head).count 是第一个节点中 int 变量的名字。注意，\*head 两边的括号是必需的。它表示我们希望解引用运算符 \* 在点运算符之前执行。但是，由于点运算符比解引用运算符具有更高的优先执行权，因此，如果没有括号，就将先执行点操作，然后再执行解引用操作（这将会导致错误发生）。下面将介绍一种更便捷的符号，通过使用它可以不用再担心括号问题。

示例 17.1 节点和指针



-> 运算符

C++ 为指针提供了一个运算符，通过它可以简化指定结构体或类的成员。在第 10 章中，介绍了箭头运算符 ->，但到目前为止，我们还未曾广泛使用它。因此，我们先来复习一下关于它的相关知识。箭头运算符结合了解引用运算符和点运算符的行为，它跟在一个指向结构体或者类的指针后面，用来引用结构体或者类的成员。例如，前面用于更改第一个节点内的数值的赋值运算符可以简单地写成如下代码：

```
head->count = 12;
```

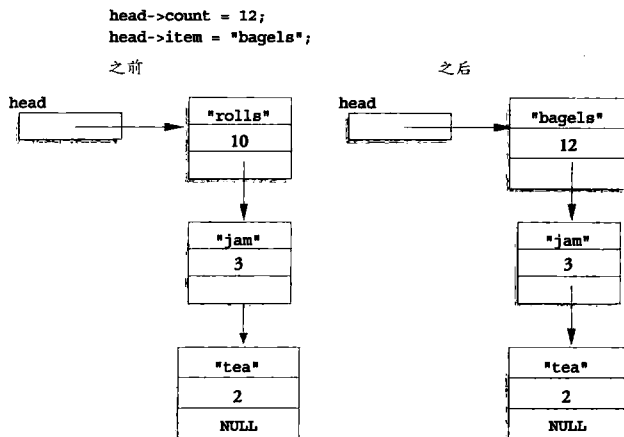
这种赋值表达方式和之前所用的表达方式具有相同的效果，但通常情况下还是会使用这种表达形式。

采用下列语句，可以把第一个节点的字符串值“rolls”改变为“bagels”：

```
head->item = "bagels";
```

示例 17.2 以图的方式表示了对第一个节点的两个值做了改变之后的结果。

示例 17.2 访问节点数据



### 箭头运算符 ->

箭头运算符 `->` 跟在一个指向结构体或者类的指针变量的后面，它指定了结构体或者类对象的成员。其语法如下：

`Pointer_Variable->Member_Name`

`Pointer_Variable` 是一个指向结构体或类的指针，箭头运算符指向结构体或类的成员变量。例如，假设你做了如下定义：

```

struct Record
{
    int number;
    char grade;
};

```

下列代码创建了一个类型为 `Record` 的动态变量，并把这个动态结构体变量的成员变量设置为 2001 和 'A'：

```

Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';

```

NULL

我们再来看看示例 17.2 中链表的最后一个节点的指针成员。最后一个节点本该有一个指针的地方却是一个 NULL。在示例 17.1 中，我们使用词组“end marker”来填充这个位置，但词组“end marker”并不是一个 C++ 表达式。在 C++ 编程中，我们使用常量 NULL 标记链表的结束（或者其他链式数据结构的结束）。

NULL 通常被用于实现两种不同（但常常却又是一致的）的目的。首先，NULL 被用于给一个指针变量赋值，否则该指针变量将没有值。这样做可以避免对内存的误引用，因为 NULL 不可能是任何内存块的地址。NULL 的第二种用途是作为结束标志。如示例 17.2 所示的节点列表，程序能够从列表的开头，顺着一个个节点依次访问下去，直到列表的尽头。此时，程序也到达了那个包含 NULL 的节点。

NULL 是 0

正如我们在第 10 章中表示的那样，常量 NULL 实际上是数字 0，但我们还是更偏向于将它写成 NULL，以 NULL 去思考，这样会使得它所代表的意义更为清楚，它就是用于赋给指针变量的一个特殊的值。标识符 NULL 的定义出现在很多标准库中，比如 `<iostream>` 和 `<cstdint>`，因此，当我们要使用 NULL 时，应该使用 `include` 指令把 `<iostream>`、`<cstdint>` 或者其他一些合适的库导入到程序中。NULL 的定义是由 C++ 预处理器来处理的，它负责把 NULL 替换为 0。接下来，编译器就再也看不到“NULL”了。因此，不会存在命名空间的问题；使用 NULL 时也不需要 using 指令了。

通过使用赋值运算符，我们可以把指针设置为 NULL，下面的代码声明了一个叫 `there` 的指针变量，并把它初始化为 NULL：

```
double *there = NULL;
```

可以将常量 NULL 赋值给指向任何数据类型的指针变量。

### NULL

NULL 是一个特殊的常量值，当指针变量还没有值时，它被用于给指针变量赋值。可以把 NULL 赋值给指向任意数据类型的指针变量。C++ 的许多库都定义了标识符 NULL，包括头文件 `<cstdint>` 和 `<iostream>`。常量 NULL 实际上是 0，但我们更偏向于将它写成 NULL，并且以 NULL 来思考它。

### 作为参数的链表

最好保持一个指针变量指向链表的头。这个指针变量可以用来作为一个链表的名字。当在一个函数中把一个链表作为参数时，这个指针（指向链表的开头）可以作为链表参数。

## 自测练习题

1. 假设你的程序包含下列类型定义：

```
struct Box
{
    string name;
    int number;
    Box *next;
};
typedef Box* BoxPtr;
```

请问下列代码的输出结果是什么？

```
BoxPtr head;
head = new Box;
head->name = "Sally";
head->number = 18;
cout << (*head).name << endl;
cout << head->name << endl;
cout << (*head).number << endl;
cout << head->number << endl;
```

2. 假设你的程序中包含自测练习题 1 所给出的类型定义和代码。练习题 1 的代码创建了一个节点，这个节点包含一个字符串为“Sally”和数值为 18 的成员。如果想把这个节点的 next 成员变量设置为 NULL，需要增加什么样的代码？
3. 考虑下列结构体的定义：

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
ListNode *head = new ListNode;
```

请写出代码把字符串“Wilbur's brother Orville”赋给 head 所指向变量的字符串成员变量 item。

## 链表

### 链表

形如示例 17.1 所示的列表被称为链表。链表是由多个节点构成的一个列表，每个节点都有一个指针成员变量，该指针指向列表的下一个节点。链表的第一个节点被称为头，这就是为什么我们把指向第一个节点的指针变量命名为 head 的原因。最后一个节点没有特别的名字，但它却有特殊的属性：它包含 NULL，用 NULL 来作为成员指针变量的值。在测试一个节点是否为一个链表的最后一个节点时，你只需要测试一下是否这个节点的指针变量为 NULL。

### 节点类型 定义

在本节中，我们的目标是写一些简单的函数来对链表进行操作。出于尽可能多地给出一些例子以及尽可能简化表示符号的目的，我们使用新的链表示例，它的数据类型比示例 17.2 更为简单。新链表示例中的节点只包含一个整数和指针。但是，在某种程度上我们会将节点变得更为复杂些。我们将把它们设计为类对象，而不是简单的结

构体。我们将要使用到的节点和指针类型的定义如下所示：

```
class IntNode
{
public:
    IntNode( ) {}
    IntNode(int theData, IntNode* theLink)
        : data(theData), link(theLink) {}
    IntNode* getLink( ) const { return link; }
    int getData( ) const { return data; }
    void setData( int theData) { data = theData; }
    void setLink(IntNode* pointer) { link = pointer; }
private:
    int data;
    IntNode *link;
};
typedef IntNode* IntNodePtr;
```

注意，类 IntNode 的所有成员函数都非常简单，因此可以使用内联定义。

请注意类 IntNode 的那个带两个输入参数的构造函数。该构造函数允许我们使用指定的整数和指定的链表作为成员变量来创建节点。例如，如果指针 p1 指向节点 n1，那么下列代码创建了一个由指针 p2 所指向的新节点，该节点包含一个数值为 42 的数据以及一个指向 n1 节点的指针：

```
IntNodePtr p2 = new IntNode(42, p1);
```

接下来，我们将使用节点类型派生一些基本函数来创建和操作链表，我们将会把节点类型和函数转换为模板版本，这样就可以用来在节点中存储任意类型的数据了。

一个单节点  
的链表

作为一个热身练习，我们先来看看如何用这种类型的节点来创建一个链表的开始部分。我们首先声明一个指针变量，称为 head，它指向链表头：

```
IntNodePtr head;
```

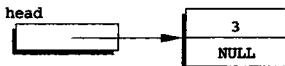
为了创建第一个节点，我们使用运算符 new 来创建一个动态变量，这个动态变量将成为链表的第一个节点：

```
head = new IntNode;
```

接下来，我们为这个新节点的成员变量进行赋值：

```
head->setData(3);
head->setLink(NULL);
```

注意，这个节点的指针成员被设置为 NULL，因为这个节点是链表中的最后一个节点（同时也是链表中的第一个节点）。此时，我们创建的链表看起来就像这样：



实际上我们不需要做这么多工作。通过使用类 IntNode 的带两个参数的构造函数，我们可以更加容易地创建单节点链表。可以用如下更简单的代码创建上图所示的单节点链表：

```
head = new IntNode(3, NULL);
```

正如我们将会看到的那样，在接下来的内容中，我们将总会使用类 `IntNode` 的带两个参数的构造函数来创建新的节点。许多程序甚至在定义 `IntNode` 时省略了不带参数的构造函数，这样的话，如果不指定每个成员变量的值就不能创建节点。

我们的单节点链表是通过一种特定方式创建的。如果想创建一个更大的链表，那么程序必须能够以一种系统的方式自动添加节点。接下来我们将描述一种简单的方法来在链表中插入节点。

### 在链表头插入节点

在本小节中，我们假设链表已经包含了一个或多个节点，并且我们也已经开发了一个添加其他节点的插入函数。插入函数的第一个参数是一个引用参数，这个参数代表一个指向链表头的指针变量，同样它也是一个指向链表第一个节点的指针变量。另外一个参数表示在新节点中存储的数值。下列代码给出了插入函数的声明：

```
void headInsert(IntNodePtr& head, int theData);
```

为了把新节点插入到链表中，我们的插入函数将使用 `new` 运算符以及类 `IntNode` 的包含两个参数的构造函数。新节点将会把 `theData` 作为它的数据，并把它链接指针成员指向链表中（插入前的链表）的第一个节点。动态变量的创建代码如下：

```
new IntNode(theData, head)
```

我们希望指针 `head` 指向新节点，因此函数体可以简单地表示成如下代码：

```
{
    head = new IntNode(theData, head);
}
```

示例 17.3 给出了这一行为的示意图。

```
head = new IntNode(theData, head);
```

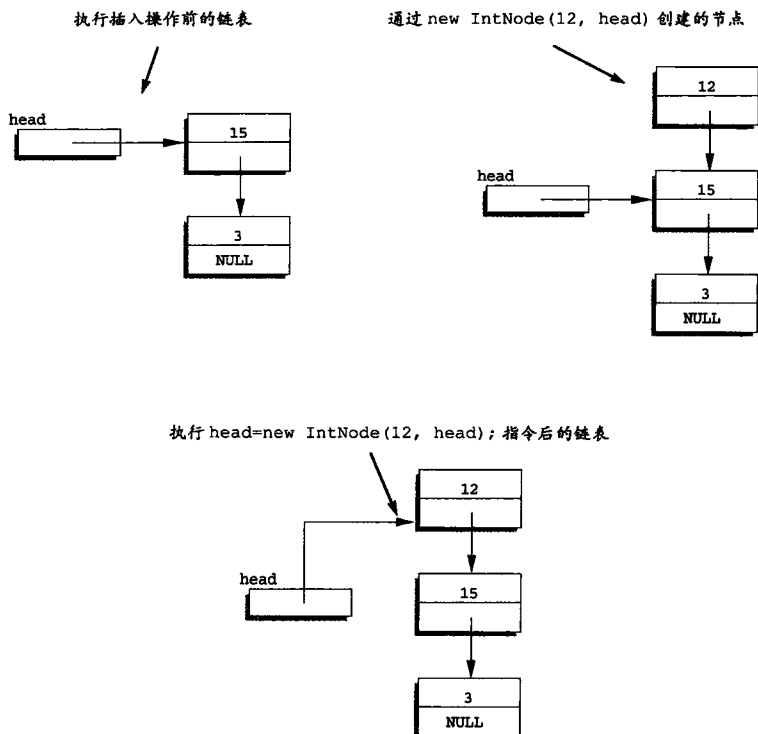
示例 17.4 给出了当 `theData` 等于 12 时函数的完整定义。

或许，有时我们希望我们的链表不包含任何东西。例如，一个购物清单可能什么都没有，因为本周没有购买任何商品。如果一个链表里面什么东西都没有，我们就称这个链表为**空链表**。一个链表是通过一个指向链表头的指针来命名的，但是空链表没有头节点。为了指定一个空链表，可以使用 `NULL`。假设指针变量 `head` 指向链表的头节点，如果你想把这个链表表示为空链表，那么请将 `head` 设置为 `NULL`，如下所示：

```
head = NULL;
```

无论什么时候，当你设计一个函数来操作链表时，请务必先去检查一下你设计的函数是否能够正常操作空链表。如果不能，那么请为空链表增加一个特例。如果你设计的函数不能适用于空链表，那么你必须采用其他方式来处理空链表，或者在你的程序中严格要求不允许出现空链表。所幸的是，通常情况下空链表也可以被视为普通链表。例如，示例 17.4 的函数 `headInsert` 是为非空链表设计的。但是通过检查可以发现它同样也可以正确地操作空链表。

### 示例 17.3 为链表头添加一个节点



### 示例 17.4 为链表增加节点的函数

#### 节点和指针类型定义

```

class IntNode
{
public :
    IntNode( ) {}
    IntNode(int theData, IntNode* theLink)
        : data(theData), link(theLink) {}
    IntNode* getLink( ) const { return link; }
    int getData( ) const { return data; }
    void setData(int theData) { data = theData; }
    void setLink(IntNode* pointer) { link = pointer; }
private :
    int data;
    IntNode *link;
};
typedef IntNode* IntNodePtr;
  
```

### 在链表头部增加节点的函数

#### 函数声明

```
void headInsert(IntNodePtr& head, int theData);
// 前提条件：指针变量 head 指向链表头。
// 运行结果：一个包含 theData 数据的新节点被增加到了链表头。
```

#### 函数定义

```
void headInsert(IntNodePtr& head, int theData)
{
    head = new IntNode(theData, head);
}
```

### 在链表中间增加节点的函数

#### 函数声明

```
void insert(IntNodePtr afterMe, int theData);
// 前提条件：afterMe 指向链表中的一个节点。
// 运行结果：一个包含 theData 数据的新节点被增加到了由指针 afterMe 所指节点的后面。
```

#### 函数定义

```
void insert(IntNodePtr afterMe, int theData)
{
    afterMe->setLink(new IntNode(theData, afterMe->getLink()));
}
```



### 陷阱：丢失节点

或许你试图写出 headInsert 函数的定义（示例 17.4），你可以使用无参数的构造函数来创建节点，并为新节点设置成员变量。如果你已经尝试过编写这个函数，那么，所编写代码的开始部分可能会如下面代码所示：

```
head = new IntNode;
head->setData(theData);
```

至此，新节点已经创建好了，它包含一个正确的数据，并且指针 head 指向新节点——所有一切都像所期望的那样。剩下需要做的工作是在新节点中设置指针成员，把链表的剩余部分挂在这个节点上，这样，新节点的指针将指向原先链表的第一个节点。你可以使用下列代码来实现，前提是你能够指出问号处应该用什么指针：

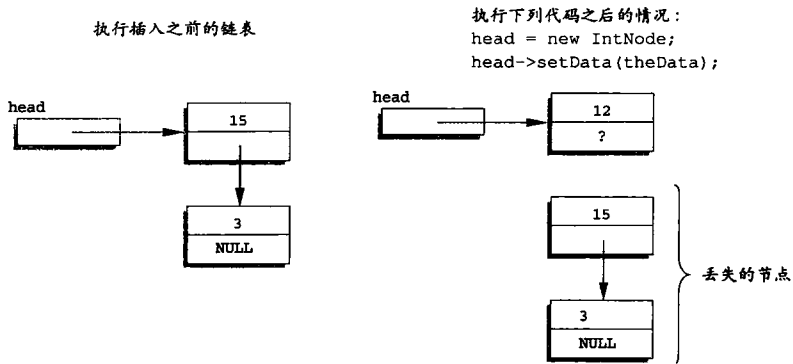
```
head->setLink(?);
```

示例 17.5 给出了新数据值为 12 时的情形，并对这个问题进行了解释。如果按照上面这种方式插入数据，那么将导致没有指针指向包含数值 15 的这个节点。因为没有任何命名的指针指向它（也没有一个指针链能扩展至这个节点），程序无法引用这个节点。这个节点以及它之后的所有节点将全部丢失。程序无法使得有个指针能够指向任何一个这些丢失掉的节点，也不能够访问这些节点的数据，不能够对这些节点进行任何其他操作。简单地说，程序无法指向这些节点。在这种情况下，程序持续运行会耗尽内存。丢失节点的这种程序有时被称为内存泄漏。一个严重的内存泄漏会导致程序



耗光内存并异常终止。更为糟糕的是，一个普通用户程序的内存泄漏（丢失节点）在某些极端情况下，会引起操作系统崩溃。为了避免这种丢失节点的现象，程序必须总是保留一些指针用来指向链表头，通常 head 之类的指针变量就是用来完成这一任务的。■

示例 17.5 丢失节点



插入和删除链表内的节点

在链表中间  
进行插入

接下来，我们设计一个函数来把一个节点插入到链表的指定位置。如果你希望链表中的节点按一定的顺序排列，比如按数值大小或者字母先后顺序，那么你就不能简单地把节点插到链表的开头或结尾。因此，我们将设计一个函数来把一个节点插入到链表中一个指定节点的后面。

假设某些其他函数或者程序的一部分已经将一个叫作 afterMe 的指针正确无误地放到了链表中，这个 afterMe 指针指向链表中的某个节点。如果我们想把新节点放到由指针 afterMe 所指向的节点的后面，如示例 17.6 图中所示的那样。不管节点内部包含的数据是什么类型的，下面的技术都适用。我们采用和前面小节相同的数据类型，来对这个问题进行具体详细的阐述。示例 17.4 给出了类型定义。下列代码给出了我们想定义的函数的声明：

```
void insert(IntNodePtr afterMe, int theData);  
// 前提条件：afterMe 指向链表中的一个节点。  
// 运行结果：包含数据 theData 的新节点被添加到指针 afterMe 所指的节点之后。
```

新节点被插入到链表的方法和我们之前所讨论的把一个节点增加到链表头部所采用的方法基本相同。唯一的区别是我们使用指针 afterMe->link，而不是指针 head。插入新节点的代码如下：

```
afterMe->setLink(new IntNode(theData, afterMe->getLink( )));
```

示例 17.6 的图给出了使 theData 等于 5 的详细过程，示例 17.4 给出了最终的函数实现。

在链表末  
进行插入

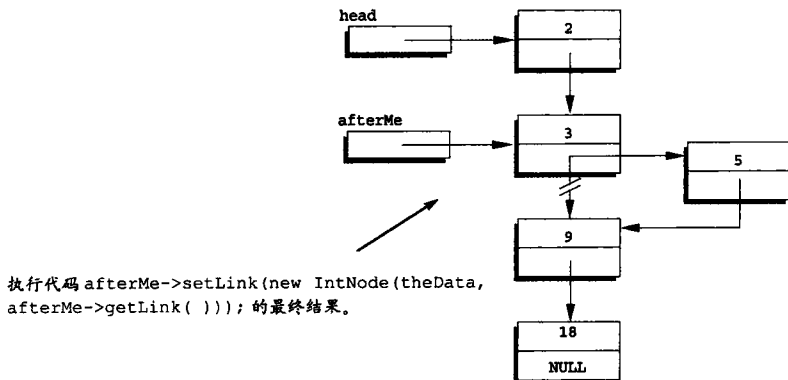
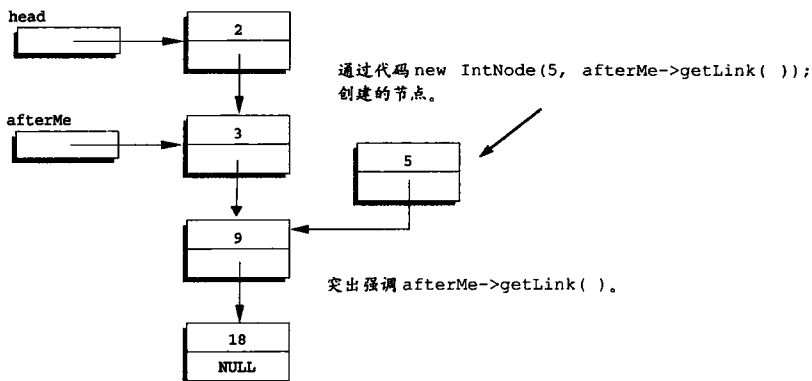
如果你仔细阅读函数 insert 的代码，你将发现即使指针 afterMe 所指向的节点

是链表中的最后一个节点，这个函数也能正确运行。但是，如果在链表的开始部分插入一个节点，insert 函数将不能正确执行。示例 17.4 中的函数 headInsert 可被用来在链表的开始部分插入节点。

与数组的  
比较

通过使用函数 insert，我们可以按照某种顺序来维护一个链表，如数值大小顺序、字母先后顺序或其他顺序。通过简单地调整两个指针就可以把一个新节点放置到一个正确的位置。这和链表有多长或者你想把新节点放到链表的哪个位置都没有关系。如果你不使用链表而使用数组实现这一功能，那么在最坏的情况下，数组的所有元素都要被复制以便为新节点放到正确位置腾出空间。尽管定位指针 afterMe 会占用一定的开销，但在链表中进行插入操作往往比使用数组做插入要高效得多。

### 示例 17.6 在链表中间插入节点



删除一个  
节点

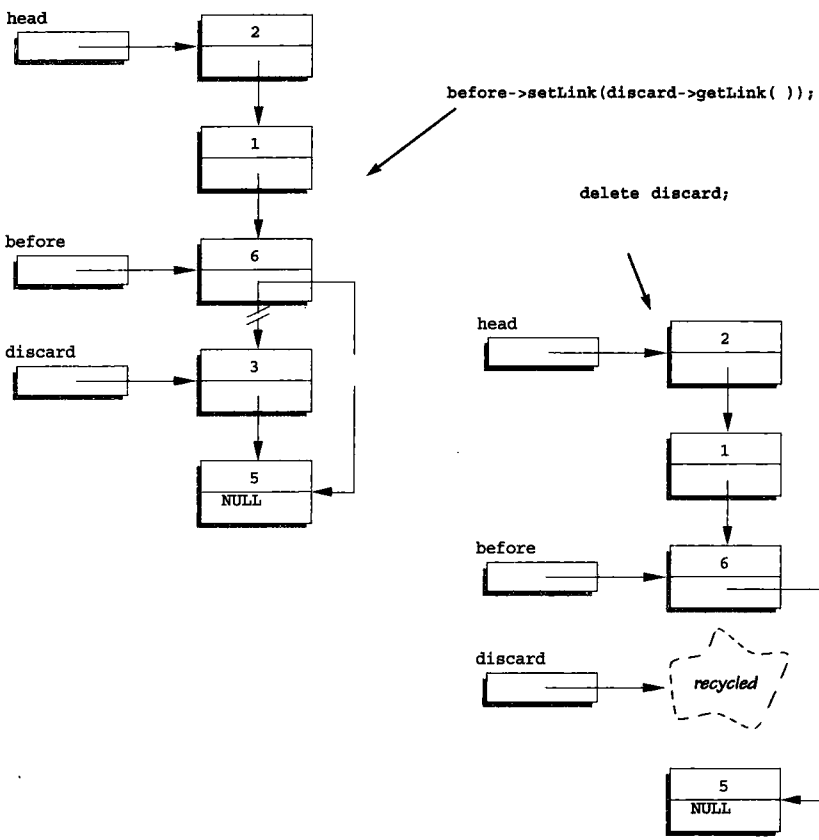
从一个链表中删除节点也非常容易。示例 17.7 是这一方法的图形表示。一旦定位到了指针 before 和 discard，剩下需要做的就只是删除节点，代码如下所示：

```
before->setLink(discard->getLink( ));
```

这足以将节点从链表中删除了。但是，如果出于某种原因，不再打算使用这个被删除的节点，那么应当把这个节点销毁，并释放它所占用的内存；可以通过调用 delete 来实现上述目的，代码如下：

```
delete discard;
```

示例 17.7 删除节点



正如我们在第 10 章中所讨论的那样，动态变量被保存在内存的自由存储区。由于自由存储区对变量的保存时间没有限制，因此，当你的程序不再需要这个变量时，你应当使用 delete 运算符把动态变量所占用的内存空间返回给内存。下面对 delete 运算符进行了总结。

### delete 运算符

delete 运算符用于删除动态变量，并释放动态变量所占的内存空间，把它返回给自由存储区。返回的存储区可被重用用来创建新的动态变量。例如，下列代码删除了指针变量 p 所指的动态变量：

```
delete p;
```

执行 delete 调用后，指针变量（例如 p）的值变得不确定。

### 陷阱：对动态数据结构使用赋值运算符

如果 head1 和 head2 是指针变量，并且 head1 指向链表头节点，下列代码将使 head2 也指向相同的头节点，即指向同一个链表：

```
head2 = head1;
```

但是，必须记住这里只有一个链表，而不是两个。如果你对 head1 所指向的链表进行了更改，那么你也对 head2 指向的链表进行了更改，因为它们本身指向的都是同一链表。

假设 head1 指向一个链表，如果你想让 head2 指向另一个链表，这个链表是 head1 所指向的链表的一个副本，那么，前面的赋值语句是做不到这一点的。你必须把整个链表的节点进行一一复制才可以。■

### 搜索链表

接下来，我们将设计一个函数来搜索链表以查找某个节点。我们将沿用前面子章节所定义的类型：IntNode（节点和指针类型的定义见示例 17.4），作为这个函数中链表的节点类型。我们设计的函数包含两个输入参数：链表名和一个我们需要查找的整数。函数将返回指向这个整数的第一个节点的指针。如果链表中不存在要查找的整数，函数返回 NULL。通过这种方式，只要查看程序运行所返回的指针是否为 NULL，我们的程序即能判断出一个整数是否在一个链表中。以下代码给出了函数的声明和函数头部的注释：

搜索

```
IntNodePtr search(IntNodePtr head, int target);  
// 前提条件：头指针指向链表头。  
// 最后一个节点的指针变量为 NULL。  
// 如果链表为空，那么头指针指向 NULL。  
// 返回结果为包含目标整数的第一个节点的指针。  
// 如果没找到包含目标整数的节点，函数返回 NULL。
```

我们将使用一个本地指针变量，在此将它称为 here，用它来在链表中移动以寻找目标节点。遍历链表或者其他任何由节点和指针所构成的数据结构，唯一的方式就是通过指针。接下来我们将把指针 here 指向第一个节点并沿着链表尾部方向一步一步一个节点地移动指针。示例 17.8 给出了这个方法的图形表示。

考虑到空链表会有一些小问题，这会扰乱我们对搜索方法的讨论。因此，我们先

假设链表至少包含一个节点。随后，我们将再回来确认我们的算法同样也可以工作于空链表。这个搜索方法会产生如下代码。

### 算法 搜索函数伪代码

```

让指针变量 here 指向链表的头节点（也就是第一个节点）。
while (here 不是指向目标节点，也不是指向最后一个节点)
{
    让 here 指向链表中的下一个节点
}
if (here 所指向节点包含目标)
    return here;
else
    return NULL;

```

为了使指针 here 能够移动到下一个节点，我们必须认为所命名的这些指针是有效的。指针 here 所指向的节点是当前节点，当前节点的指针成员变量所指向的节点就是下一个节点。我们用如下代码来表示 here 指向的当前节点的指针成员变量：

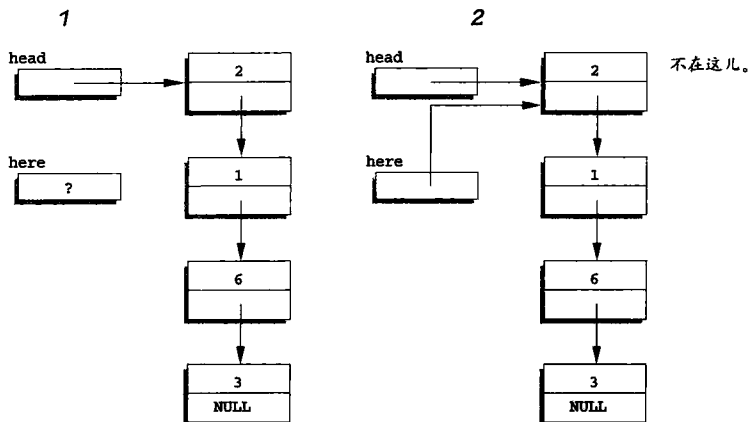
```
here->getLink()
```

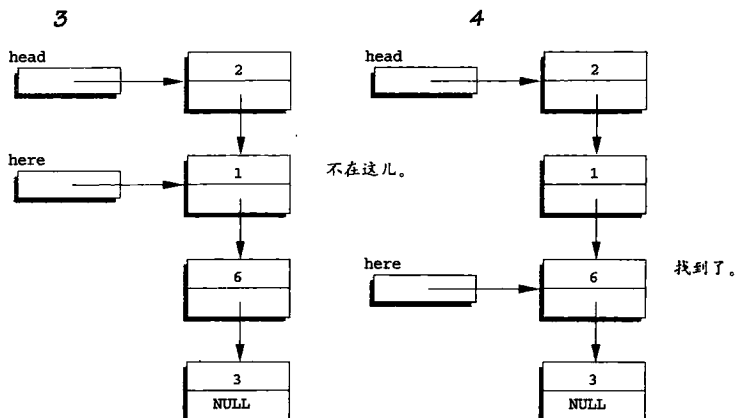
为了把 here 移到下一个节点，我们要改变 here，使它指向上面代码所给出的指针所代表的节点。下面的代码将把指针 here 移动到下一个节点：

```
here = here->getLink();
```

### 示例 17.8 搜索链表

target 是 6





将这些代码片段放到一起，将会得出搜索函数优化后的算法伪代码，如下所示：

#### 算法优化

```

here = head;

while (here->getData() != target && here->getLink() != NULL)
    here = here->getLink();

if (here->getData() == target)
    return here;
else
    return NULL;

```

注意上述代码中 while 语句的 Boolean 表达式。我们通过测试来看看成员变量 `here->getLink()` 是否等于 NULL，以此来判断 `here` 是否指向最后一个节点。

#### 空链表

最后，我们还是必须回过头来看看空链表的问题。如果我们检查前面的代码，则发现对于空链表来说会存在这样一个问题。如果链表是空的，那么 `here` 就等于 NULL，继而下面的表达式将成为未定义的表达式：

```

here->getData()
here->getLink()

```

当 `here` 为 NULL 时，它不会指向任何节点，因此，也不会有数据成员或者指针成员。为此，我们需要对空链表进行特殊处理。示例 17.9 给出了函数的完整定义。

#### 双向链表

#### 双向链表

普通的链表允许我们沿着链表尾部的方向移动（利用每个节点的 link 变量信息实现逐点移动）。双向链表有两个链接，一个指针指向后一个节点，另外还有一个指针指向前一个节点。在某些情况下，链表包含一个指向前一个节点的链接，能够简化程序的代码。例如，如果从一个链表中删除一个节点，我们就不再需要准备一个变量来记住待删除节点的前一个节点。示例 17.10 以图形的方式给出了一个双向链表的样例。

### 示例 17.9 定位链表中一个节点的函数

#### 函数声明

```
IntNodePtr search(IntNodePtr head, int target);
// 前提条件: 指针 head 指向链表头。
// 最后一个节点的指针变量为 NULL。
// 如果链表为空链表, 那么 head 为 NULL。
// 返回结果为链表中指向目标数值的第一个节点的指针。
// 如果没有一个节点的数值为目标数值, 那么函数返回为 NULL。
```

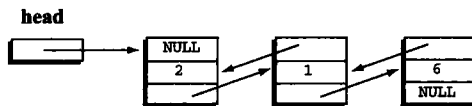
#### 函数定义

```
// 使用 cstddef。
IntNodePtr search(IntNodePtr head, int target)
{
    IntNodePtr here = head;
    if (here == NULL) // 如果是空链表
    {
        return NULL;
    }
    else
    {
        while (here->getData() != target && here->getLink() != NULL)
            here = here->getLink();

        if (here->getData() == target)
            return here;
        else
            return NULL;
    }
}
```

示例 17.4 已经给出了 IntNode 和 IntNodePtr 的定义。

### 示例 17.10 双向链表



如果一个双向链表的节点数据类型为整型, 那么它的节点类的定义如下列代码所示:

```
class DoublyLinkedIntNode
{
public:
    DoublyLinkedIntNode() {}
    DoublyLinkedIntNode(int theData, DoublyLinkedIntNode* previous,
        DoublyLinkedIntNode* next)
        : data(theData), nextLink(next), previousLink(previous) {}
```

```

DoublyLinkedListNode* getNextLink( ) const { return nextLink; }
DoublyLinkedListNode* getPreviousLink( ) const
{ return previousLink; }
int getData( ) const { return data; }
void setData(int theData) { data = theData; }
void setNextLink(DoublyLinkedListNode* pointer)
{ nextLink = pointer; }
void setPreviousLink(DoublyLinkedListNode* pointer)
{ previousLink = pointer; }

private:
    int data;
    DoublyLinkedListNode *nextLink;
    DoublyLinkedListNode *previousLink;
};
typedef DoublyLinkedListNode* DoublyLinkedListNodePtr;

```

双向链表节点类的代码和单向链表节点类的代码几乎完全一样，除了我们为双向链表增加了一个私有成员变量 `previousLink` 外（该变量用于存储一个指向前一个节点的链接）。函数 `setPreviousLink` 和函数 `getPreviousLink` 用于获得和设置指向前一个节点的链接，此外还为构造函数增加了一个额外的参数用来初始化 `previousLink`。前面我们所说的链接现在特指 `nextLink`，用来区分节点的前一个节点和后一个节点。

## 为双向链表增加一个节点

为双向链表增加一个节点

为了给一个双向链表增加一个节点，我们需要设置两个链接而不再是一个链接了。示例 17.11 给出了一般的处理过程。双向链表插入函数的声明基本上和单向链表的插入函数声明是一样的：

```
void headInsert(DoublyLinkedListNodePtr& head, int theData);
```

首先，我们创建一个新节点，它的 `nextLink` 链接指向链表原来的头节点，它的 `previousLink` 为 `NULL`。这么做是因为这个新节点将成为新的头节点：

```
DoublyLinkedListNode* newHead = new DoublyLinkedListNode
(theData, NULL, head);
```

原来的头节点必须把它的前向链接指向新的头节点：

```
head->setPreviousLink(newHead);
```

最后，我们把 `newHead` 设置为新的头节点：

```
head = newHead;
```

示例 17.13 给出了函数的完整定义。

## 从双向链表中删除一个节点

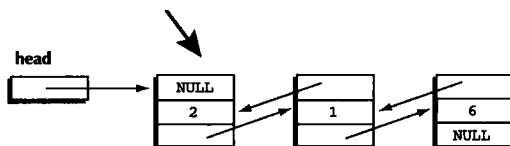
从双向链表中删除一个节点

在删除双向链表中的一个节点时，我们同样需要更新待删除节点的前面和后面节点的链接。由于有后向链接，所以我们没必要再像单向链表那样需要一个单独的变量来记录前面的节点。示例 17.12 给出了通过指定位置来删除节点的一般过程。记住，对于一些特例必须进行专门处理，比如删除链表的头节点以及尾节点。



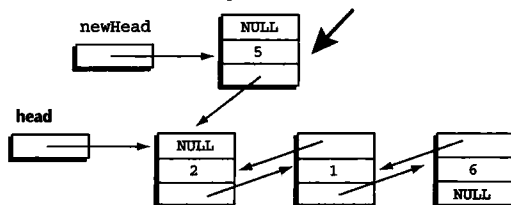
### 示例 17.11 为双向链表增加一个节点

插入新节点前的列表：



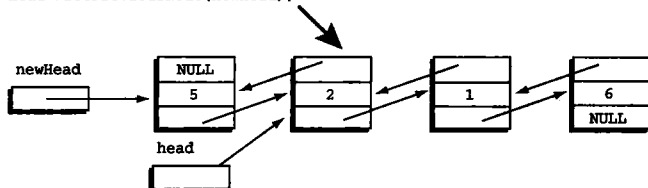
通过下列代码创建的节点：

```
newHead = new DoublyLinkedIntNode(5, NULL, head);
```



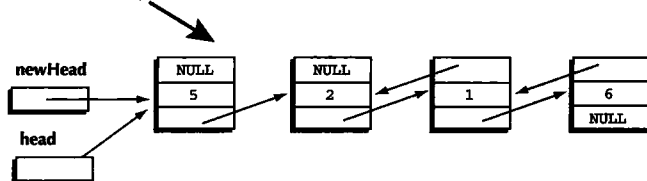
设置原始头节点的前向链接：

```
head->setPreviousNode(newHead);
```



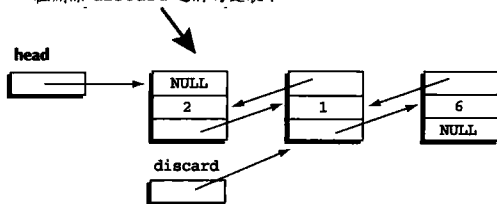
把 head 设置为 newHead：

```
head = newHead;
```



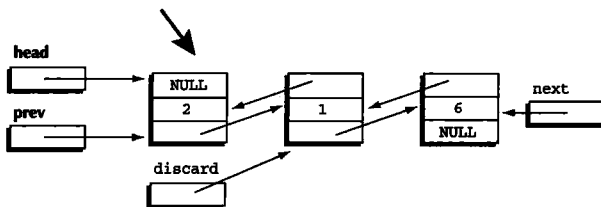
### 示例 17.12 从双向链表中删除一个节点

在删除 discard 之前的链表：



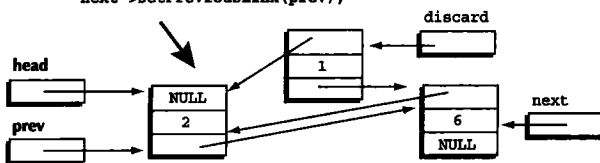
设置指向前一个节点和下一个节点的指针：

```
DoublyLinkedListNodePtr prev = discard->getPreviousLink();
DoublyLinkedListNodePtr next = discard->getNextLink();
```



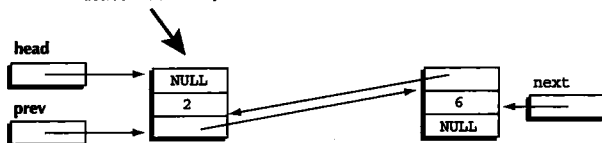
跳过 discard：

```
prev->setNextLink(next);
next->setPreviousLink(prev);
```



删除 discard：

```
delete discard;
```



函数 delete 的声明如下：

```
void delete(DoublyLinkedListNodePtr& head,
            DoublyLinkedListNodePtr discard);
```

参数 discard 是我们打算删除的节点的指针。我们也必须把链表的头节点 head 作为输入参数来处理当 discard 为 head 时的情况：

```
if (head == discard)
{
    head = head->getNextLink( );
    head->setPreviousLink(NULL);
}
```

在本例中，我们必须在链表中把 head 传递给它的下一个节点。接着我们再把这个新的 head 节点的前向链接设置为 NULL，因为它不再有前一个节点了。在一个更加通用的例子中，变量 discard 指向任何一个非 head 节点。对于这种情况，通过将 discard 之后的第一个节点的前向链接重置来绕开 discard 节点，如示例 17.12 所示：

```
else
{
    DoublyLinkedListNodePtr prev = discard->getPreviousLink( );
    DoublyLinkedListNodePtr next = discard->getNextLink( );
    prev->setNextLink(next);
    if (next != NULL)
    {
        next->setPreviousLink(prev);
    }
}
```

现在，discard 之前的第一个节点的后向链接指向 discard 之后的第一个节点，而 discard 之后的第一个节点的前向链接指向 discard 之前的第一个节点。考虑到 discard 有可能是链表中的最后一个节点，因此，我们必须检查并确认 next != NULL，这样的话就可以避免在函数 setPreviousLink 中对一个 NULL 进行解引用操作。

示例 17.13 给出了函数的完整定义。

### 示例 17.13 在双向链表中增加和删除节点函数

#### 节点与指针类型定义

```
class DoublyLinkedListNode
{
public:
    DoublyLinkedListNode ( ) {}
    DoublyLinkedListNode (int theData, DoublyLinkedListNode* previous,
                          DoublyLinkedListNode* next)
        : data(theData), nextLink(next), previousLink(previous) {}
    DoublyLinkedListNode* getNextLink( ) const
    { return nextLink; }
    DoublyLinkedListNode* getPreviousLink( ) const
    { return previousLink; }
    int getData( ) const
    { return data; }
    void setData(int theData)
    { data = theData; }
```

```

void setNextLink(DoublyLinkedListNode* pointer)
    { nextLink = pointer; }
void setPreviousLink(DoublyLinkedListNode* pointer)
    { previousLink = pointer; }
private :
    int data;
    DoublyLinkedListNode *nextLink;
    DoublyLinkedListNode *previousLink;
}
typedef DoublyLinkedListNode* DoublyLinkedListNodePtr;

```

### 在链表头增加一个节点的函数

#### 函数声明

```

void headInsert(DoublyLinkedListNode& head, int theData);
// 前提条件：指针变量 head 指向链表的头部。
// 运行结果：包含数据 theData 的一个新节点被增加到链表的头部。

```

#### 函数定义

```

void headInsert(DoublyLinkedListNodePtr& head, int theData)
{
    DoublyLinkedListNode* newHead = new DoublyLinkedListNode
                                   (theData, NULL, head);
    head->setPreviousLink(newHead);
    head = newHead;
}

```

### 删除节点的函数

#### 函数声明

```

void deleteNode(DoublyLinkedListNodePtr& head,
                DoublyLinkedListNodePtr discard);
// 前提条件：指针变量 head 指向链表头，discard 指针指向待删除的节点。
// 运行结果：discard 所指向的节点被从链表中删除了。

```

#### 函数定义

```

void deleteNode(DoublyLinkedListNodePtr& head,
                DoublyLinkedListNodePtr discard);
{
    if (head == discard)
    {
        head = head->getNextLink();
        head->setPreviousLink(NULL);
    }
    else
    {
        DoublyLinkedListNodePtr prev = discard->getPreviousLink();
        DoublyLinkedListNodePtr next = discard->getNextLink();
        prev->setNextLink(next);
        if (next != NULL)
        {
            next->setPreviousLink(prev);
        }
    }
}

```

```

delete discard;
}

```

### 自测练习题

4. 请给出链表中节点和指针的类型定义。假设节点类型名为 `NodeType`，指针类型名为 `PointerType`，链表中节点的数据对象是字母。
5. 在使用链表时，通常通过一个指向第一个节点的指针进行引用，但是空链表没有第一个节点。那么，对于空链表来说，通常会采用什么样的指针值来表示？
6. 假设你的程序中包含下列节点类型定义和指针变量声明：

```

struct Node
{
    double data;
    Node *next;
}

typedef Node* Pointer;
Pointer p1, p2;

```

假设我们创建了一个以 `Node` 为节点类型的链表，指针变量 `p1` 指向链表中的一个节点。请写出使得 `p1` 指向其下一个节点的代码。（本道练习题没有使用指针变量 `p2`，它用于下一道练习题，和本道题目无关。）

7. 假设你的程序中包含自测练习题 6 中所示的节点类型定义和指针变量声明。进一步假设指针变量 `p2` 是指向上述类型的一个节点，这个节点是链表中的某个节点，但不是链表中的最后一个节点。请写出一段代码，它能删除指针 `p2` 所指节点之后的所有节点。要求这段代码执行后，链表除了减少了节点外，其他保持不变。（提示：你可能还需要再声明一个指针变量来完成上述功能。）
8. 假设你的程序包含下列类型定义和指针变量声明：

```

class Node
{
public :
    Node(double theData, Node* theLink)
        : data(theData), next(theLink) {}
    Node* getLink( ) const { return next; }
    double getData( ) const { return data; }
    void setData(double theData) { data = theData; }
    void setLink(Node* pointer) { next = pointer; }
private :
    double data;
    Node *next;
}

typedef Node* Pointer;
Pointer p1, p2;

```

假设指针变量 `p1` 指向链表中的一个节点。请写出使得 `p1` 指向其下一个节点的代码。（本道练习题没有使用指针变量 `p2`，它用于下一道练习题，和本道题目无关。）

9. 假设你的程序中包含自测练习题 8 中所示的节点类型定义和指针变量声明。进一步

假设指针变量 `p2` 是指向上述类型的一个节点，这个节点是链表中的某个节点，但不是链表中的最后一个节点。请写出一段代码，它能删除指针 `p2` 所指节点之后的所有节点。要求这段代码执行后，链表除了减少了节点外，其他保持不变。（提示：你可能还需要再声明一个指针变量来完成上述功能。）

10. 选择关于下列陈述语句表述正确的选项，并解释原因：

对于一个大数组和一个大链表，如果它们的节点数据类型都是一样的，那么，把一个新对象插入到链表中的指定位置处和插入到数组的指定位置处，链表相对于指针：

- 更高效
- 更低效
- 效率基本相同

11. 完成下列函数体：

```
void insert(DoublyLinkedListNodePtr afterMe, int theData);
```

该函数的目标是在一个双向链表中，把值为 `theData` 的新节点插入到节点 `afterMe` 之后。

12. 双向链表相对于单向链表来说，什么操作比较容易实现？什么操作实现起来相对困难？

### 示例：使用双向链表实现的通用排序模板函数

按照惯例，我们将把类型定义和函数实现转换成模板，这样的话不管链表中的节点类型是什么数据类型，函数都可以正常工作。但是，我们还需要考虑一些具体细节。我们需要做的最核心的事就是把节点中数据的类型（如，在示例 17.4 中被替换的数据类型为 `int`）用类型参数 `T` 替换掉，并在合适的位置处插入如下代码：

```
template<class T>
```

但是，我们还应当考虑到类型参数 `T` 有可能是类类型。如果类型参数 `T` 是类类型，一个类型 `T` 的传值参数应当被改为常量引用参数，并且返回类型应该为带 `const` 关键字的类型 `T`，这样可以保障函数返回结果为常量值（在第 8 章中，解释了为什么需要返回一个 `const` 的值）。

示例 17.14 和示例 17.15 给出了最终形成的模板函数，它体现了我们之前所描述的那些变化。对于这个简单的整数链表例子，我们有必要再做一个改变。由于大多数编译器并未实现模板 `typedefs`，因此我们还不能使用它们。这意味着在某些场合下，我们需要使用下列晦涩难懂的参数类型说明：

```
Node<T>*&
```

这是一个传引用参数，参数所代表的指针指向一个节点类型为 `Node<T>` 的节点。接下来，我们从示例 17.15 复制了一个函数声明，目的是使我们能够明白在此上下文关于参数类型的说明：

```
template<class T>
void headInsert(Node<T>*& head, const T& theData);
```

### 示例 17.14 链表库的接口文件

```

1 // 这是头文件 listtools.h。这个文件包含了对存储类型为 T 的链表进行操作
2 // 的链表节点类型定义和函数声明。
3 // 链表用类型为 Node<T>* 的指针来表示，
4 // 这个指针指向链表的头节点。
5 // 函数实现在文件 listtools.cpp 中给出。
6 #ifndef LISTTOOLS_H
7 #define LISTTOOLS_H

8 namespace LinkedListSavitch
9 {
10     template<class T>
11     class Node
12     {
13     public :
14         Node(const T& theData, Node<T>* theLink) : data(theData),
15                                                     link(theLink){}
16         Node<T>* getLink() const { return link; }
17         const T getData() const { return data; }
18         void setData(const T& theData) { data = theData; }
19         void setLink(Node<T>* pointer) { link = pointer; }
20     private :
21         T data;
22         Node<T> *link;
23     };
24     template<class T>
25     void headInsert(Node<T>*& head, const T& theData);
26     // 前提条件：
27     // 指针变量 head 指向链表头。
28     // 运行结果：
29     // 包含数据 theData 的一个新节点被添加到链表头。

30     template<class T>
31     void insert(Node<T>* afterMe, const T& theData);
32     // 前提条件：afterMe 指向链表中的一个节点。
33     // 运行结果：包含数据 theData 的一个新节点被加入到链表中，
34     // 它是指针 afterMe 所指节点的下一个节点。

35     template<class T>
36     void deleteNode(Node<T>* before);
37     // 前提条件：指针 before 指向链表中的某个节点，
38     // 该节点之后至少还有一个节点。
39     // 运行结果：指针 before 所指节点的后序节点被从链表中删除，
40     // 该后序节点所占的空间被释放，
41     // 返还给自由存储区。

42     template<class T>
43     void deleteFirstNode(Node<T>*& head);
44     // 前提条件：指针 head 指向链表中的第一个节点，
45     // 链表至少包含一个节点。
46     // 运行结果：head 所指向的节点已经从链表中被删除，
47     // 它所占用的存储归还给了自由存储区。

48     template<class T>

```

把 T 作为类型参数放在 const T& 处是可以接受的。我们使用常量引用参数，是因为我们想到 T 可能会经常被用作基类型。

```

48     Node<T>* search(Node<T>* head, const T& target);
49     // 前提条件：指针 head 指向链表头。
50     // 最后一个节点的指针变量为 NULL。
51     // 我们为类型 T 定义了运算符 ==。
52     // (== 作为判断是否相等的一个准则。)
53     // 如果链表为空，那么 head 为 NULL。
54     // 返回链表中节点值为目标值的第一个节点的指针。
55     // 如果链表中没有一个节点的数值等于目标数值，
56     // 那么函数返回 NULL。
57 } //LinkedListSavitch

58 #endif //LISTTOOLS_H

```

---

### 示例 17.15 链表库的实现文件

---

```

1  // 这是实现文件 listtools.cpp。
2  // 这个文件包含文件 listtools.h 中所声明的那些函数的实现。
3  #include <cstddef>
4  #include "listtools.h"

5  namespace LinkedListSavitch
6  {
7      template<class T>
8      void headInsert(Node<T>*& head, const T& theData)
9      {
10         head = new Node<T>(theData, head);
11     }

12     template<class T>
13     void insert(Node<T>* afterMe, const T& theData)
14     {
15         afterMe->setLink(new Node<T>(theData, afterMe->getLink( )));
16     }

17     template<class T>
18     void deleteNode(Node<T>* before)
19     {
20         Node<T> *discard;
21         discard = before->getLink( );
22         before->setLink(discard->getLink( ));
23         delete discard;
24     }

25     template<class T>
26     void deleteFirstNode(Node<T>*& head)
27     {
28         Node<T> *discard;
29         discard = head;
30         head = head->getLink( );
31         delete discard;
32     }

```



```

33     // 使用 cstdint.
34     template<class T>
35     Node<T>* search(Node<T>* head, const T& target)
36     {
37         Node<T>* here = head;
38         if (here == NULL) // 如果链表为空
39         {
40             return NULL;
41         }
42         else
43         {
44             while (here->getData( ) != target && here->getLink( ) != NULL)
45                 here = here->getLink( );
46
47             if (here->getData( ) == target)
48                 return here;
49             else
50                 return NULL;
51         }
52     } //LinkedListSavitch

```

---

## 17.2 链表的应用

然而，有许多在前的，将要在后；在后的，将要在前。

马太福音，19:30

先来先服务。

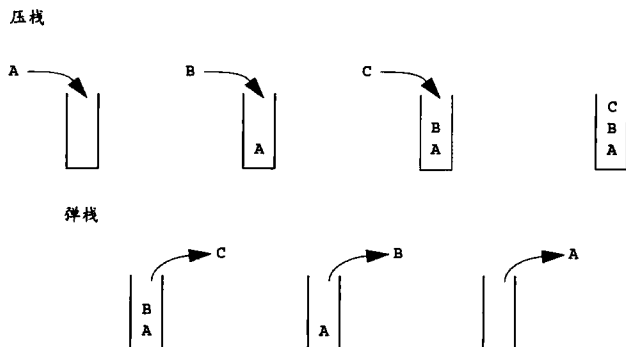
一句俗语（更为通俗的话）

链表可以应用在多个场景下。本节仅举出个别小例子来说明它的应用——那些常用的数据结构，通常都会把链表作为它们的实现核心。

### 示例：栈模板类

栈（stack）这种数据结构的特点是从栈里取数据时的顺序和存入数据到栈的顺序恰好相反。假设你把字母 ‘A’、‘B’ 和 ‘C’ 按顺序放入到栈里。当你把数据从栈中取出时，它们从栈里被删除的次序依次是 ‘C’、‘B’ 和 ‘A’。示例 17.16 以图形的方式表示了栈的使用。如示例 17.16 所示，你可以把栈看作地上的一个洞。为了从栈中取出某元素，你必须先删除它上面的那些元素。由于这个原因，栈通常被称为后进先出式数据结构。

## 示例 17.16 栈



栈被用于许多语言处理任务中。第 13 章中讨论了计算机系统如何使用栈来追踪 C++ 的函数调用。但是，在此处我们将只讨论一个非常简单的应用。这个应用示例的目的是向你展示如何使用链表技术来实现一个特别的数据结构，比如栈。

示例 17.17 给出了类 `stack` 的接口。这是一个带类型参数 `T` 的模板类，参数类型 `T` 是指存储在栈中的数据的数据类型。栈中存储的元素是类型为 `T` 的数值。在本例中，我们展示了 `T` 被类型 `char` 所替换的情况。但是，在大多数应用中，在栈中所存储的元素通常是结构体或者类对象。存储在栈中的每条记录（类型为 `T` 的元素）被称为栈帧（`stack frame`），这就是我们有时会在栈模板类的定义中使用 `stackFrame` 作为一个标识符名的原因。对于栈来说，通常有两种操作：增加一个元素到栈，从栈中删除一个元素。增加元素的操作被称为压（`pushing`）元素入栈，因此，我们把对应此操作的成员函数称为 `push`。从栈中删除元素的操作被称为弹（`popping`）元素出栈，我们把对应此操作的成员函数称为 `pop`。

`push` 和 `pop` 的名字来源于对栈的操作过程的可视化。栈的机制可以类比于咖啡馆中放置器皿的机制。咖啡馆工作台上有个洞，洞里放的是器皿。器皿下面有一个弹簧，弹簧的张力正好可使得只有最上面的一个器皿能伸出到工作台上。如果把这种机制用在栈数据结构上，数据将会被写到器皿上（这可能有违健康法则，但却是一个不错的类比）。为了在栈上增加一个器皿，把器皿放到其他器皿的顶部，这个新器皿的重量会下压弹簧。当把上面一个器皿拿走后，下面的器皿又弹出到工作台上。

示例 17.18 给出了一个简单的例子来描述如何使用 `Stack` 类。这个程序从一行文本中每次读取一个字符，并把字符放到栈里。接着，这个程序把所有字符从栈里一个一个地删除，并把它们打印到屏幕上。由于数据被删除的顺序和数据进入栈的顺序正好相反，因此，输出结果是输入行的反向序列。就像我们通常处理模板类那样，在应用程序中，我们已经把 `Stack` 类的实现 `#include` 进来了。这意味着直到我们已经实

现了 Stack 类模板，我们才能编译和运行我们的应用程序。

示例 17.19 给出了模板类 Stack 的实现文件，它对成员函数进行了定义。我们的 Stack 类采用链表实现，链表的头节点充当栈的顶部。成员变量 top 是一个指向链表头的指针。指针 top 的作用和我们之前在链表中所讨论的 head 指针的作用是一样的。

### 示例 17.17 Stack 模板类的接口文件

```

1  // 这是头文件 stack.h。这是类 Stack 的接口。
2  //Stack 是一个模板类，栈里的数据元素的类型为 T。
3  #ifndef STACK_H
4  #define STACK_H                                或许你更偏向于使用 const T& 来代替
                                                    类型参数 T。
5  namespace StackSavitch
6  {
7      template<class T>
8      class Node
9      {
10     public:
11     Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
12         Node<T>* getLink( ) const { return link; }
13         const T getData( ) const { return data; }
14         void setData(const T& theData) { data = theData; }
15         void setLink(Node<T>* pointer) { link = pointer; }
16     private:
17         T data;
18         Node<T> *link;
19     };
20     template<class T>
21     class Stack
22     {
23     public:
24         Stack( );
25         // 初始化对象为空栈。
26         Stack(const Stack<T>& aStack); ← 拷贝构造函数。
27         Stack<T>& operator =(const Stack<T>& rightSide);
28         virtual ~Stack( ); ← 析构函数用于删除对象 stack 并将
                                其所占内存归还给自由存储区。
29         void push(T stackFrame);
30         // 运行结果：stackFrame 已经被添加到栈中了。。
31
32         T pop( );
33         // 前提条件：栈为非空栈。
34         // 返回最上面的栈帧并从栈中删除顶部的栈帧。
35         bool isEmpty( ) const ;
36         // 如果栈为空，则返回 true；否则，返回 false。
37     private:
38         Node<T> *top;
39     };
40 } //StackSavitch

```

---

```
40 #endif //STACK_H
```

---

### 示例 17.18 应用 Stack 模板类的程序

---

```
1 // 本程序演示了 Stack 模板类的用法。
2 #include <iostream>
3 #include "stack.h"
4 #include "stack.cpp"
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using StackSavitch::Stack;
9 int main( )
10 {
11     char next, ans;
12
13     do
14     {
15         Stack<char> s;
16         cout << "Enter a line of text:\n";
17         cin.get(next);
18         while (next != '\n')
19         {
20             s.push(next);
21             cin.get(next);
22
23             cout << "Written backward that is:\n";
24             while ( ! s.isEmpty( ) )
25                 cout << s.pop( );
26             cout << endl;
27
28             cout << "Again?(y/n): ";
29             cin >> ans;
30             cin.ignore(10000, '\n');
31         }while (ans != 'n' && ans != 'N');
32
33     }while (ans != 'n' && ans != 'N');
34
35     return 0;
36 }
```

#### 示例运行结果

```
Enter a line of text: straw
Written backward that is:
warts
Again?(y/n): y
Enter a line of text: I love C++
Written backward that is:
++C evol I
Again?(y/n): n
```

第9章讨论了 cin 的 ignore 成员。它丢弃了遗留在行中的输入。

---

## 示例 17.19 Stack 模板类的实现

---

```

1  // 这是实现文件 stack.cpp。
2  // 这是 Stack 模板类的实现。
3  // 模板类 Stack 的接口在头文件 stack.h 中。

4  #include <iostream>
5  #include <cstdlib>
6  #include <cstdlib>
7  #include "stack.h"
8  using std::cout;

9  namespace StackSavitch
10 {

11     // 使用 cstdlib。
12     template<class T>
13     Stack<T>::Stack( ) : top(NULL)
14     {
15         // 特意为空。
16     }

17     template<class T>
18     Stack<T>::Stack(const Stack<T>& aStack)
19     < 自测练习题 14 为拷贝构造函数的实现。 >
20     template<class T>
21     Stack<T>& Stack<T>::operator =(const Stack<T>& rightSide)
22     < 自测练习题 15 为重载赋值运算符的定义。 >

23     template<class T>
24     Stack<T>::~~Stack( )
25     {
26         T next;
27         while (! isEmpty( ))
28             next = pop( ); // pop 调用 delete。
29     }
30
31     // 使用 cstdlib。
32     template<class T>
33     bool Stack<T>::isEmpty( ) const
34     {
35         return (top == NULL);
36     }
37     template<class T>
38     void Stack<T>::push(T stackFrame)
39     < 自测练习题 13 为函数定义的剩余部分。 >

40     // 使用 cstdlib 和 iostream。
41     template<class T>
42     T Stack<T>::pop( )
43     {
44         if (isEmpty( ))
45         {
46             cout << "Error: popping an empty stack.\n";
47             exit(1);

```

```

48     }
49     T result = top->getData( );
50     Node<T> *discard;
51     discard = top;
52     top = top->getLink( );
53     delete discard;
54     return result;
55 }
56 } //StackSavitch

```

自测练习题 13 要求写出成员函数 `push` 的定义。但是，实际上我们已经给出了这个任务的算法。`push` 成员函数的代码和示例 17.15 所示的函数 `headInsert` 本质上是相同的，只不过我们在 `push` 函数中使用名称为 `top` 的指针而不是 `head`。

一个空栈仅仅是一个空链表，因此，可以通过设置指针 `top` 为 `NULL` 来实现空栈。一旦意识到 `NULL` 代表空栈，默认构造函数和成员函数 `empty` 的实现就变得显而易见了。

拷贝构造函数的实现略微有些复杂，但是它的实现并不需要使用任何我们之前没有讨论过的技术。关于其中的具体实现细节，我们把它留作自测练习题 14。

`pop` 成员函数首先检查是否栈为空。如果非空，它接着会删除栈顶的字符。它设置本地变量 `result` 的值使其等于栈中 `top` 节点的值，如下列代码所示：

```
T result = top->getData( );
```

`top` 节点的数据保存到变量 `result` 后，指针 `top` 被移动到下一个节点，这样即把 `top` 节点从链表中有效地删除了。移动指针 `top` 的代码如下列代码所示：

```
top = top->getLink( );
```

在移动指针 `top` 前，一个被称为 `discard` 的临时指针引入到链表中，使它指向将要被删除的节点。可以使用 `delete` 对被删除节点所占用的存储空间进行回收，代码如下所示：

```
delete discard;
```

通过成员函数 `pop` 从链表中删除的每个节点都会使用 `delete` 对其所占用的内存进行回收，这样的话，析构函数所需要做的所有工作只是调用 `pop` 从栈中删除每一个元素。接着，每个节点会把它所占用的存储归还给自由存储区。

## 栈

栈是一个后进先出的数据结构；也就是说，从栈中获取数据元素的顺序与把数据存入栈的顺序正好相反。

**压榨和弹栈**

把一个数据元素增加到栈数据结构是指把这个数据元素压入到栈上。把一个数据元素从栈中删除是指把它从栈中弹出。

**自测练习题**

13. 请给出示例 17.17 和示例 17.19 中所描述的模板类 Stack 的成员函数 push 的定义。
14. 请给出示例 17.17 和示例 17.19 中所描述的模板类 Stack 的拷贝函数。
15. 请给出示例 17.17 和示例 17.19 中所描述的模板类 Stack 的重载赋值运算符的定义。

**示例：队列模板类**

栈是一个后进先出的数据结构。另一个常用的数据结构是队列，它用来处理先进先出模式的数据。可以通过链表的方式来实现队列，类似于栈模板类那样。但是，队列需要链表的头和尾都有一个指针，这是因为在这两个地方都会有操作。从链表头删除一个节点比从链表另一端删除一个节点要更容易一些。因此，我们将从链表头部删除节点（现在我们把它称为链表前端），并在链表另一端增加一个节点，我们将它称为链表的后端（或者队列的后端）。

示例 17.20 给出了队列模板类 Queue 的定义。示例 17.21 给出了使用类 Queue 的一个应用。我们把类 Queue 成员函数的定义留作自测练习题（但是，注意答案在本章的最后给出，用以解决你可能会碰到的一些具体的细节问题）。

**示例 17.20 队列模板类的接口文件**

```

1
2 // 这是头文件：queue.h。这是类 Queue 的接口，
3 // 类 Queue 是一个队列模板类，队列中数据元素的类型为 T。
4 #ifndef QUEUE_H
5 #define QUEUE_H
6 namespace QueueSavitch
7 {
8     template<class T>
9     class Node
10     {
11     public:
12         Node(T theData, Node<T>* theLink) : data(theData),
13             link(theLink){}
14         Node<T>* getLink() const { return link; }
15         const T getData() const { return data; }
16         void setData(const T& theData) { data = theData; }

```

模板类 Node 的定义和我们在示例 17.17 给出的栈接口中 Node 的定义是一样的。参考“提示：命名空间的注解”来讨论这个复制类。

```

16         void setLink(Node<T>* pointer) { link = pointer; }
17     private:
18         T data;
19         Node<T> *link;
20     };
    或许你更偏向于使用 const T& 来代替类型参数 T。

21     template<class T>
22     class Queue
23     {
24     public:
25         Queue( );
26         // 把对象初始化为空队列。

27         Queue(const Queue<T>& aQueue);  ←—— 拷贝构造函数。

28         Queue<T>& operator =(const Queue<T>& rightSide);

29         virtual ~Queue( );  ←—— 析构函数删除队列，并释放队列所占的存
    储空间，把它返还给自由存储区。

30
31         void add(T item);
32         // 运行结果：数据元素被添加到队列的后端。

33         T remove( );
34         // 前提条件：队列非空。
35         // 在队列的前端返回数据元素。
36         // 从队列中删除数据元素。
37         bool isEmpty( ) const ;
38         // 如果队列为空则返回 true，否则返回 false。
39     private:
40         Node<T> *front; // 指向链表头。
41         // 从头部删除元素。
42         Node<T> *back; // 指向链表另一端的节点。
43         // 数据被添加到这一端。
44     };

45 } //QueueSavitch

46 #endif //QUEUE_H

```

### 示例 17.21 使用队列模板类的程序

```

1 // 这个程序演示了队列模板类的使用。
2 #include <iostream>
3 #include "queue.h"
4 #include "queue.cpp"
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using QueueSavitch::Queue;

9 int main( )
10 {

```



```

11     char next, ans;

12     do
13     {
14         Queue<char> q;
15         cout << "Enter a line of text:\n";
16         cin.get(next);
17         while (next != '\n')
18         {
19             q.add(next);
20             cin.get(next);
21         }

22         cout << "You entered:\n";
23         while ( ! q.isEmpty() )
24             cout << q.remove( );
25         cout << endl;
26         cout << "Again?(y/n): ";
27         cin >> ans;
28         cin.ignore(10000, '\n');
29     } while (ans != 'n' && ans != 'N');

30     return 0;
31 }

```

这个关于队列的应用程序和示例 17.18 所给出的栈的应用程序相似。

### 示例运行结果

```

Enter a line of text:
straw
You entered:
straw
Again?(y/n): y
Enter a line of text:
I love C++
You entered:
I love C++
Again?(y/n): n

```

### 队列

队列是一个先进先出数据结构；也就是说数据元素从队列中删除的顺序和它们进入队列的顺序是一致的。

### 提示：命名空间的注解

注意到，不管是命名空间 StackSavitch（示例 17.17）还是 QueueSavitch（示例 17.20）都定义了一个称为 Node 的模板类。结果表明，这两个定义是相同的，但是，不管这两个定义是相同的还是不同的，我们在这里想重点对相同性进行讨论。在 C++ 中，不允许对相同的标识符进行两次定义，即使两个标识符的定义是一样的，它们的

名字必须有所区别。在我们所讨论的这个案例中，允许名字相同是因为它们在两个不同的命名空间中。在同一程序中甚至既使用 Stack 模板类又使用 Queue 模板类都是合法的。但是，我们应当使用如下代码来表示：

```
using StackSavitch::Stack;
using QueueSavitch::Queue;
```

而不是使用下面的代码：

```
using namespace StackSavitch;
using namespace QueueSavitch;
```

如果你没有使用标识符 Node，那么对于大部分编译器来说上面两组 using 指令的用法都是允许的，但是第二组 using 指令提供了两个 Node 的定义，因此应该避免使用。

在程序中，使用下列语句中的其中一个是可以的，但是不可以同时使用两个：

```
using StackSavitch::Node;
```

或者

```
using QueueSavitch::Node; ■
```

## 自测练习题

16. 给出模板类 Queue（示例 17.20）的默认（不带输入参数）构造函数的定义以及成员函数 Queue<T>::isEmpty 的定义。
17. 给出模板类 Queue（示例 17.20）的成员函数 Queue<T>::add 的定义以及成员函数 Queue<T>::remove 的定义。
18. 给出模板类 Queue（示例 17.20）的析构函数的定义。
19. 给出模板类 Queue（示例 17.20）的拷贝构造函数的定义。
20. 给出模板类 Queue（示例 17.20）的重载赋值运算符的定义。

## 友元类以及类似的其他选择

也许我们已经发现在模板类 Node 中使用取值函数 getLink 和设置函数 setLink 是一件糟糕的事情（见示例 17.17 或者示例 17.20）。可能我们会希望简单地 把类 Node 中的变量 link 设置为 public 而不是 private，这样的话就可以避免调用 getLink 和 setLink 函数了。但是，在我们打算放弃把所有成员变量设置为私有类型之前，一定要记住两件事。首先，对于一个程序员来说，使用 getLink 和 setLink 实际上并不会比直接访问节点的链接更困难（但是，getLink 和 setLink 的确引入了一些开销，这样可能会使得程序效率有些下降）。其次，要有一个方法可以避免使用 getLink 和 setLink，直接访问节点的链接但不需要把 link 成员变量设置为 public。我们来探索一下第二种办法的可行性。

第 8 章讨论了友元函数。回顾一下相关内容，如果 f 是类 C 的一个友元函数，那么 f 就不是类 C 的成员函数；但是，当编写函数 f 的定义时，它是可以访问类 C 的私有成员的，这就如同我们在类 C 的成员函数的定义中可以做到的那样。与一个函数可

**友元** 以成为另一个函数的友元一样，一个类也可以成为另一个类的友元。如果类 F 是类 C 的友元，那么类 F 的每一个成员函数都是类 C 的友元。继而，如果模板类 Queue 是模板类 Node 的友元，那么，就可以在类 Queue 的成员函数中直接访问类 Node 的 link 私有成员变量。示例 17.22 给出了我们所讨论这些内容的细节。

**示例 17.22 模板类 Node 的友元模板类 Queue**

---

```

1  // 这是头文件 queue.h。这是类 Queue 的接口，
2  // 类 Queue 是一个队列模板类，它的元素数据类型为 T。
3  #ifndef QUEUE_H
4  #define QUEUE_H

5  namespace QueueSavitch
6  {
7      template<class T>
8      class Queue;                在前面声明，不要忘了分号。

9      template<class T>
10     class Node                  这是示例 17.20 的另一种实现方法，在这个实现版本中，
11     {                          Queue 模板类是 Node 模板类的友元。
12     public:
13         Node(T theData, Node<T>* theLink) : data(theData),
14                                             link(theLink){}
15         friend class Queue<T>;
16     private:
17         T data;
18         Node<T>* link;          如果 Node<T> 只是在友元类 Queue<T> 的定义中被用
19                                 到，那么就不再需要取值函数和赋值函数了。

20     template<class T>
21     class Queue
22     {
23         <模板类 Queue 的定义和示例 17.20 所给出的定义是相同的。
24         但是，成员函数的定义不同于我们所给出的非友元类 Queue
25         的版本。(在自测练习题中)>
26     }
27     //QueueSavitch

```

---

```

26 #endif //QUEUE_H
27 #include <iostream>            实现文件将会包含这些函数的定义，并且其他成员函数做了
28 #include <cstdlib>            类似的修改后也可以通过名字来对节点的 link 和 data 成
29 #include <cstdlib>            员变量进行访问了。
30 #include "queue.h"
31 using std::cout;
32 namespace QueueSavitch
33 {
34     template<class T> // 使用 cstdlib。
35     void Queue<T>::add(T item)
36     {
37         if (isEmpty())
38             front = back = new Node<T>(item, NULL);
39         else
40             {

```

```

41         back->link = new Node<T>(item, NULL);
42         back = back->link;
43     }
44 }

```

如果效率是主要问题, 你可能会使用 (front == NULL) 而不是 (isEmpty())。

```

45 template<class T> // 使用 cstdlib 和 iostream.
46 T Queue<T>::remove()
47 {
48     if (isEmpty())
49     {
50         cout << "Error: Removing an item from an empty queue.\n";
51         exit(1);
52     }

```

```

53     T result = front->data;

```

和自测练习题 17 的答案中所给出的实现进行比较。

```

54     Node<T> *discard;
55     discard = front;
56     front = front->link;
57     if (front == NULL) // 如果你删除了最后一个节点。
58         back = NULL;

```

```

59     delete discard;
60     return result;
61 }
62 } //QueueSavitch

```

## 在前面声明

当一个类是另外一个类的友元时, 我们通常会在它们的类定义中进行相互引用。这就要求我们要把前面的类的声明引入到后面的那个类或者类模板的定义中, 如示例 17.22 所示。注意, 在前面声明只是紧随其分号之后的类定义或者类模板定义的题头。17.4 节给出了一个使用友元类的完整例子 (参考编程示例“树模板类”)。

有两种方法和友元类的目标非常相似, 并且其使用方法和类以及模板类如 Node 和 Queue 也非常相似: (1) 使用保护或私有继承来从 Node 派生出 Queue, (2) 在类 Queue 的定义中给出类 Node 的定义, 这样可使得类 Node 成为一个内部类 (模板) 定义 (第 14 章讨论了保护继承, 第 7 章讨论了在一个类的内部进行内部类定义)。

## 示例: 包含节点链的哈希表

### 哈希表 哈希映射

哈希表 (hash table) 或者哈希映射 (hash map) 是一种从内存中高效地存储和读取数据的数据结构。可以有很多方法来创建一个哈希表; 在本小节中, 我们将使用数组和单向链表的组合来创建哈希表。在 17.1 节中, 我们查询链表的实现机制是通过遍历链表中的每一个节点来查找目标节点。这个处理流程可能需要对链表中的每个节点进行检查, 如果链表很长的话, 它将会非常耗时间。相比之下, 哈希表具有快速找到目标节点的能力, 尽管在最差情况下 (出现的可能性不大), 我们采用哈希表的实现方式和单向链表的实现方式一样慢。

哈希函数

在哈希表方案中,对象被存储在哈希表中,通过键(key)来和它关联。只要给定键,我们就可以获取到对象。理想情况下,每个对象都有唯一的键。如果一个对象没有内在的唯一键,我们可以通过使用哈希函数来计算它。在大多数情况下,哈希函数计算出来的是一个数值。

冲突

链表

例如,我们使用哈希表来存储一个字典。这个哈希表可以用来进行拼写检查——从哈希表中丢失的单词不能正确拼写。我们将用一个固定大小的数组来创建哈希表,每一个数组元素对应一个链表。通过哈希函数计算出的键将会被映射为数组的索引号。实际的数据被存储在由哈希值索引号所代表的链表中。示例 17.23 以图形的方式表达了这一思想,使用一个包含 10 个元素的固定大小数组。开始时,数组 hasharray 的每一个元素包含一个指向空单向链表的引用。首先,我们把单词“cat”添加到哈希表中,为单词“cat”分配一个键,键值为 2 (后面我们将展示键值是如何快速计算出来的)。接着,我们把单词“dog”和“bird”添加到哈希表中,这两个单词被分配的键值分别为 4 和 7。所添加的每个字符串作为链表的头,它们的哈希值作为数组的索引号。最后,我们把单词“turtle”添加到哈希表中,它的哈希值为 2。由于单词“cat”已经被存入到数组的第二个元素中,这样的话就会发生冲突。单词“turtle”和“cat”都被映射到了数组的同一元素中。当包含节点链的哈希表中发生这种冲突时,我们只是简单地把新节点插入到现有的链表中。在本例中,在数组的第二个元素中现在有两个节点:“turtle”和“cat”。

为了从哈希表中取值,我们先要计算出搜索目标的哈希值。接着,我们按顺序搜索存储在 hasharray[hashvalue] 中的链表。如果在链表中没有找到搜索目标,那就说明哈希表中没有存储搜索目标。如果链表长度比较小,那取值过程就会很快。

### 字符串哈希函数

计算字符串哈希值的一个简单方法是把字符串中每个字符的 ASCII 码值进行相加,接着对相加结果和数组元素个数进行求模运算,附录 C 给出了一部分 ASCII 码表。函数 computeHash 给出了哈希值的计算代码:

```
int computeHash(string s)
{
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
    {
        hash = hash + s[i];
    }
    return hash % SIZE; // 本例中, SIZE = 10
}
```

例如,字符串“dog”的 ASCII 码是:

```
d -> 100
o -> 111
g -> 103
```

哈希函数的计算方法如下:

```
Sum          = 100 + 111 + 103 = 314
Hash = Sum % 10 = 314 % 10      = 4
```

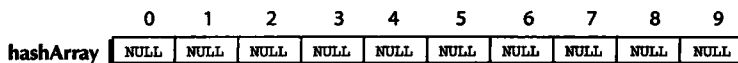
在本例中,我们先计算一个较大的值,即字符串中所有字符 ASCII 码值之和。但是,我们定义的数组仅能容纳数量有限的元素。为了使 ASCII 码值之和能收缩到数组大小的范围内,我们用数组大小对 ASCII 码值之和进行求模运算,本例中数组大小是 10。在实际应用中,数组大小一般是一个质数,这个质数要大于放置于哈希表<sup>1</sup>中的元素个数。计算得出的哈希值 4 充当字符串“dog”的辨识值。但是,其他字符串可能也会映射为同一个值。例如,我们可以计算出单词“cat”被映射为: $(99+97+116)\%10=2$ ,而单词“turtle”被映射为: $(116+117+114+116+108+101)\%10=2$ 。

示例 17.24 和示例 17.25 给出了哈希表类的完整代码清单。示例 17.26 给出了一个演示程序。哈希表的定义使用了数组,数组中的每个元素是示例 17.14 所定义的员工类。我们使用示例 17.14 和示例 17.15 所定义的通用链表库来实现链表。

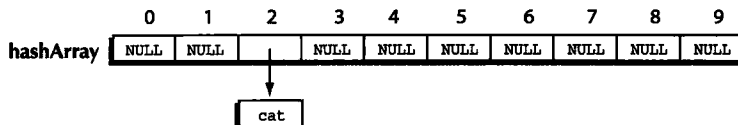
### 示例 17.23 创建一个哈希表

包含 10 个空链表的哈希表

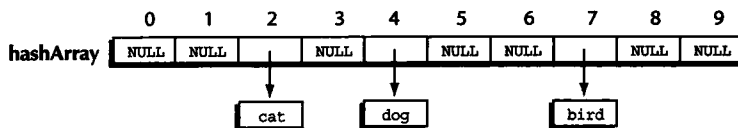
```
Node<string> *hashArray[10];
for (int i=0; i<10; i++) hashArray[i] = NULL;
```



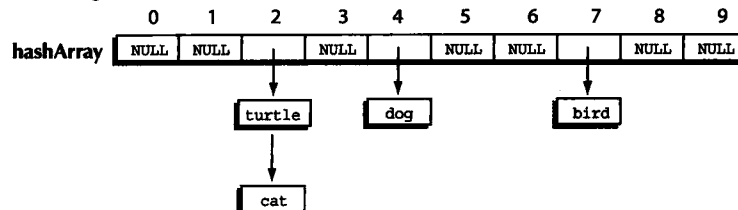
给键为 2 的元素增加一个值“cat”之后



给键为 4 的元素增加一个值“dog”,给键为 7 的元素增加一个值“bird”之后



给键为 2 的元素增加一个值“turtle”之后,会产生冲突,“turtle”会被链接到包含“cat”的链表



<sup>1</sup> 采用质数进行求模运算可以避免公约数问题,这样可以使得计算结果不相同,从而避免发生冲突。

### 示例 17.24 哈希表类的接口

---

```

1 // 这是头文件 hashtable.h。这是类 HashTable 的接口，
2 // 类 HashTable 是一个哈希表，
3 // 表中元素数据类型为字符串。
4 #ifndef HASHTABLE_H
5 #define HASHTABLE_H

6 #include <string>
7 #include "listtools.h"      库“listtools.h”是示例 17.14 中的链表库的接口。

8 using LinkedListSavitch::Node;
9 using std::string;

10 namespace HashTableSavitch
11 {
12     const int SIZE = 10; // 哈希表数组的最大元素个数。

13     class HashTable
14     {
15     public:
16         HashTable( ); // 初始化空哈希表。

17         // 通常包含一个拷贝构造函数和重载赋值运算符。
18         // 此处为了节省空间，
19         // 忽略了它们。

20         virtual ~HashTable( ); // 析构函数负责销毁哈希表。

21         bool containsString(string target) const ;
22         // 如果搜索目标在哈希表中，则返回 true，
23         // 否则返回 false。

24         void put(string s);
25         // 向哈希表中增加一个字符串。

26     private:
27         Node<string> *hashArray[SIZE]; // 实际的哈希表
28         static int computeHash(string s); // 计算哈希值
29     }; // 哈希表
30 } // HashTableSavitch
31 #endif //HASHTABLE_H

```

---

### 示例 17.25 哈希表类的实现

---

```

1 // 这是实现文件：hashtable.cpp。
2 // 这是类 HashTable 的实现。

3 #include <string>
4 #include "listtools.h"
5 #include "hashtable.h"

```

```

6  using LinkedListSavitch::Node;
7  using LinkedListSavitch::search;
8  using LinkedListSavitch::headInsert;
9  using std::string;

10 namespace HashTableSavitch
11 {
12     HashTable::HashTable( )
13     {
14         for (int i = 0; i < SIZE; i++)
15         {
16             hashArray[i] = NULL;
17         }
18     }

19     HashTable::~~HashTable( )
20     {
21         for (int i=0; i<SIZE; i++)
22         {
23             Node<string> *next = hashArray[i];
24             while (next != NULL)
25             {
26                 Node<string> *discard = next;
27                 next = next->getLink( );
28                 delete discard;
29             }
30         }
31     }

32     int HashTable::computeHash(string s)
33     {
34         int hash = 0;
35         for (int i = 0; i < s.length( ); i++)
36         {
37             hash = hash + s[i];
38         }
39         return hash % SIZE;
40     }

41     void HashTable::put(string s)
42     {
43         int hash = computeHash(s);
44         if (search(hashArray[hash], s)==NULL)
45         {
46             // 如果搜索目标不在链表中，把它插入到链表中。
47             headInsert(hashArray[hash], s);
48         }
49     }
50 //HashTableSavitch

```

---

### 示例 17.26 哈希表演示程序

```

1  //HashTable 类的演示程序。

```



```

2  #include <string>
3  #include <iostream>
4  #include "hashtable.h"
5  #include "listtools.cpp"
6  #include "hashtable.cpp"
7  using std::string;
8  using std::cout;
9  using std::endl;
10 using HashTableSavitch::HashTable;

11 int main( )
12 {
13     HashTable h;

14     cout << "Adding dog, cat, turtle, bird" << endl;
15     h.put("dog");
16     h.put("cat");
17     h.put("turtle");
18     h.put("bird");
19     cout << "Contains dog? " << h.containsString("dog") << endl;
20     cout << "Contains cat? " << h.containsString("cat") << endl;
21     cout << "Contains turtle? " << h.containsString("turtle") << endl;
22     cout << "Contains bird? " << h.containsString("bird") << endl;

23     cout << "Contains fish? " << h.containsString("fish") << endl;
24     cout << "Contains cow? " << h.containsString("cow") << endl;

25     return 0;
26 }

```

### 示例运行结果

```

Adding dog, cat, turtle, bird
Contains dog? 1
Contains cat? 1
Contains turtle? 1
Contains bird? 1
Contains fish? 0
Contains cow? 0

```

## 哈希表

哈希表是一种把数据元素关联到键的数据结构。键是根据哈希函数进行计算的。

### 哈希表的效率

哈希表的效率依赖于几个因素。首先，我们不妨来看一些极端的例子。如果插入到哈希表中的每个元素的键都相同，那这就是我们在运行程序时会碰到的性能最糟糕的情况。出现这种情况时，哈希表中的所有元素都存储在同一个单向链表上，find 操作可能要从链表中的第一个元素开始逐一搜索。幸运的是，如果我们插入哈希表中的

元素是随机的，那么它们的哈希值全部都相等的概率非常小。对应地，如果插入到哈希表中的每个元素的键都不同，那这就是我们在运行程序时会碰到的性能最好的情况。这表明没有发生任何冲突，因此，find 操作只需要在链表中搜索一个元素即可，因为搜索目标总是链表中的第一个节点。

通过使用更好的哈希函数，我们能减少发生冲突的可能性。例如，把字符串的每一个字母相加的这个简单的哈希函数没有考虑这些字母的顺序。单词“rat”和“tar”将会被哈希为同一值。对于字符串 s 来说，一个更好的哈希算法是根据每一个字母在单词中出现的位置增加其权重来增加每一个字母的值：

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
{
    hash = 31 * hash + s[i];
}
```

另一种减少冲突的方法是把哈希表变得更大一些。例如，如果哈希表数组有一个容纳 10 000 个元素的容量，而我们只插入了 1000 个元素，那么，发生冲突的概率将比只能容纳 1000 个元素的哈希表发生冲突的概率小得多。但是，只插入 1000 个元素到一个容量为 10 000 的哈希表中，会导致 9000 个存储位置未被使用，这对于内存来说是一个浪费。这是一个时间与空间的平衡问题。通常会牺牲一些空间来换取运行时的性能，反之亦然。

时间与空间  
的平衡

### 自测练习

21. 假设你所在的大学为每个学生分配了一个特有的九位数字的 ID 号。你可能想创建一个哈希表来为 ID 号建立索引，每一个索引号代表一个学生对象。哈希表的大小为  $N$ ， $N$  的位数不超过九位。请描述一个简单的哈希函数，能够把 ID 号映射为哈希索引。
22. 为类 HashTable 写一个 outputHashTable() 函数，该函数输出存储在哈希表中的每一个元素。

### 示例：集合 (set) 模板类

set 是元素的集合，在一个集合中的每一个元素只能出现一次。计算机科学中的许多问题都可以在集合这种数据结构的帮助下加以解决。对链表进行适当改动是实现集合的一种简单易懂的方法。在这种实现方式中，使用单向链表来存储每一个集合中的元素。代表每个节点的 data 变量只简单地包含集合中的一个元素。

示例 17.27 以图形的方式表示了使用这种数据结构的两个例子。集合 round 包含“peas”、“ball”和“pie”，而集合 green 包含“peas”和“grass”。字符串“peas”同时出现在两个集合 round 和 green 中。注意，如果用来填充 Node 模板的数据类型是一个指向对象的指针，那么多个链表可能会引用一个公共的对象，而不是在每个链表中为同一对象都创建一个副本。

集合的基本操作

集合类应当支持以下基本操作：

```
add element：把一个新元素添加到集合中。  
contains：判断一个目标元素是否是集合的一个成员。  
union：返回两个集合的并集。  
intersection：返回两个集合的交集。
```

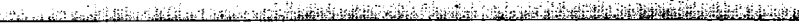
还应该有一个操作来支持对集合中的每一个元素进行遍历。其他一些有用的集合操作包括获取集合中元素的个数和从集合中删除一个元素。这些操作的实现被作为一道练习题出现在本章编程练习 7 中。

示例 17.28 和示例 17.29 实现了一个通用的集合：集合模板类。Set 类使用示例 17.14 给出的链表工具。add 函数简单地把一个节点添加到链表的前端，前提是所添加的元素不在集合中。contains 函数使用链表库中的 search 函数。我们通过简单地循环每一个链表中的每一个元素来搜索要查找的目标。

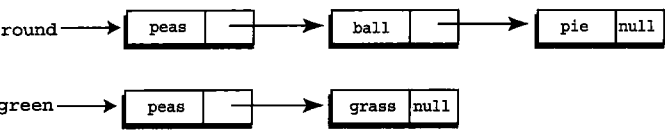
union 函数把调用对象集合中的元素和输入参数 otherSet 集合中的元素组合在一起。为了求两个集合的并集，我们先创建一个新的空的 Set 对象。接着，我们开始对调用对象集合和 otherSet 集合中的元素进行遍历。把所有元素都添加到新的集合中。由于 add 函数保障了集合中元素的唯一性，因此，我们不必去检查 union 函数中出现重复元素的情况。

intersection 函数和 union 函数比较相似，它也创建一个新的、空的 Set 对象。但这次，我们是要把既出现于调用对象集合又出现于 otherSet 对象集合中的那些共同元素填充到集合中。这是通过对调用对象集合中的每一个元素进行遍历而完成的。对于 otherSet 中的每一个元素，我们调用 contains 函数来判断其是否是公共元素。如果 contains 返回 true，那就表示元素同时存在于两个集合中，可以被添加到新集合中。

示例 17.30 给出了一个小的演示程序。



示例 17.27 使用链表来实现集合



集合

集合是一堆无序的数据元素集。

### 示例 17.28 集合模板类的接口文件

---

```

1  // 这是头文件：set.h。这是类 Set 的接口。
2  // 类 Set 是一个通用的集合类。
3  #ifndef SET_H
4  #define SET_H

5  #include "listtools.h"           库 "listtools.h" 是示例 17.14 所提供的链表库接口。
6  using LinkedListSavitch::Node;

7  namespace SetSavitch
8  {
9      template <class T>
10     class Set
11     {
12     public :
13         Set() { head = NULL; } // 初始化空集合。

14         // 通常会包括一个拷贝构造函数和一个重载赋值运算符函数。
15         // 此处，为了节省空间略去。

16         virtual ~Set(); // 析构函数销毁 set。
17         bool contains(T target) const ;
18         // 如果搜索目标在集合中，返回 true；否则，返回 false。

19         void add(T item);
20         // 把一个新元素添加到集合中。

21         void output();
22         // 把集合元素输出到控制台。

23         Set<T>* setUnion(const Set<T>& otherSet);
24         // 对调用对象集合和 otherSet 取并集，
25         // 返回一个指向新集合的指针。

26         Set<T>* setIntersection(const Set<T>& otherSet);
27         // 对调用对象集合和 otherSet 取交集，
28         // 返回一个指向新集合的指针。
29     private :
30         Node<T> *head;
31     }; //Set
32 } //SetSavitch
33 #endif //SET_H

```

---

### 示例 17.29 集合模板类的实现文件

---

```

1  // 这是实现文件 set.cpp。
2  // 这是类 Set 的实现。

3  #include <iostream>
4  #include "listtools.h"
5  #include "set.h"

```

```

6  using std::cout;
7  using std::endl;
8  using LinkedListSavitch::Node;
9  using LinkedListSavitch::search;
10 using LinkedListSavitch::headInsert;

11 namespace SetSavitch
12 {

13     template<class T>
14     Set<T>::~Set( )
15     {
16         Node<T> *toDelete = head;
17         while (head != NULL)
18         {
19             head = head->getLink( );
20             delete toDelete;
21             toDelete = head;
22         }
23     }

24     template<class T>
25     bool Set<T>::contains(T target) const
26     {
27         Node<T>* result = search(head, target);
28         if (result == NULL)
29             return false ;
30         else
31             return true ;
32     }

33     void Set<T>::output( )
34     {
35         Node<T> *iterator = head;
36         while (iterator != NULL)
37         {
38             cout << iterator->getData( ) << " ";
39             iterator = iterator->getLink( );
40         }
41         cout << endl;
42     }

43     template<class T>
44     void Set<T>::add(T item)
45     {
46         if (search(head, item)==NULL)
47         {
48             // 如果搜索目标不在链表中, 把它加入链表。
49             headInsert(head, item);
50         }
51     }

52     template<class T>
53     Set<T>* Set<T>::setUnion(const Set<T>& otherSet)
54     {

```

```

55     Set<T> *unionSet = new Set<T>( );
56     Node<T>* iterator = head;
57     while (iterator != NULL)
58     {
59         unionSet->add(iterator->getData( ));
60         iterator = iterator->getLink( );
61     }
62     iterator = otherSet.head;
63     while (iterator != NULL)
64     {
65         unionSet->add(iterator->getData( ));
66         iterator = iterator->getLink( );
67     }
68     return unionSet;
69 }

70 template<class T>
71 Set<T>* Set<T>::setIntersection(const Set<T>& otherSet)
72 {
73     Set<T> *interSet = new Set<T>( );
74     Node<T>* iterator = head;
75     while (iterator != NULL)
76     {
77         if (otherSet.contains(iterator->getData( )))
78         {
79             interSet->add(iterator->getData( ));
80         }
81         iterator = iterator->getLink( );
82     }
83     return interSet;
84 }
85 } //SetSavitch

```

---

### 示例 17.30 使用集合模板类的程序

---

```

1 //Set 类的演示程序。

2 #include <iostream>
3 #include <string>
4 #include "set.h"
5 #include "listtools.cpp"
6 #include "set.cpp"
7 using std::cout;
8 using std::endl;
9 using std::string;
10 using namespace SetSavitch;

11 int main( )
12 {
13     Set<string> round; // 球状的东西
14     Set<string> green; // 绿色的东西

```

```

15     round.add("peas"); // 在两个集合都存在的样本数据
16     round.add("ball");
17     round.add("pie");
18     round.add("grapes");
19     green.add("peas");
20     green.add("grapes");
21     green.add("garden hose");
22     green.add("grass");

23     cout << "Contents of set round: ";
24     round.output( );
25     cout << "Contents of set green: ";
26     green.output( );

27     cout << "ball in set round? " <<
28     round.contains("ball") << endl;
29     cout << "ball in set green? " <<
30     green.contains("ball") << endl;

31     cout << "ball and peas in same set? ";
32     if ((round.contains("ball") && round.contains("peas")) ||
33         (green.contains("ball") && green.contains("peas")))
34         cout << " true" << endl;
35     else
36         cout << " false" << endl;

37     cout << "pie and grass in same set? ";
38     if ((round.contains("pie") && round.contains("grass")) ||
39         (green.contains("pie") && green.contains("grass")))
40         cout << " true" << endl;
41     else
42         cout << " false" << endl;

43     cout << "Union of green and round: " << endl;
44     Set<string> *unionset = round.setUnion(green);
45     unionset->output( );
46     delete unionset;

47     cout << "Intersection of green and round: " << endl;
48     Set<string> *interiset = round.setIntersection(green);
49     interiset->output( );
50     delete interiset;

51     return 0;
52 }

```

### 示例运行结果

```

Contents of set round: grapes pie ball peas
Contents of set green: grass garden hose grapes peas
ball in set round? 1
ball in set green? 0
ball and peas in same set? true
pie and grass in same set? false
Union of green and round:
garden hose grass peas ball pie grapes

```

一些编译器可能会输出 true 或 false, 而不是 1 或 0。

```
Intersection of green and round:  
peas grapes
```

### 链表创建的集合的效率

#### 集合

我们可以对用链表创建的数据结构——集合的几种基本操作的运行时效率进行分析。在集合中增加一个新元素其实就是把一个新节点插入到链表中。这要求在链表中只设置一个链接。contains 函数通过对集合中的所有元素进行遍历来搜索目标值，这可能需要对链表中的每一个节点进行检查。当我们调用 setUnion 函数来对集合  $A$  和  $B$  取并集时，它对两个集合的所有元素都要进行遍历，并把每个元素插入到一个新的集合中。如果集合  $A$  中有  $n$  个元素，集合  $B$  中有  $m$  个元素，那么这个函数会检查  $n+m$  个元素。但是，这个操作里还有些隐藏的开销，因为 add 函数在把一个元素插入到新的集合前，会对整个链表进行搜索来去除掉重复的元素。这个开销会随着元素数目的增长而增长。最后，对集合  $A$  和集合  $B$  操作的 setIntersection 函数会调用 contains 函数来检查集合  $A$  中的每一个元素是否在集合  $B$  中。由于 contains 函数要检查  $m$  个节点，每个节点对应于集合  $A$  中的一个元素，那么 setIntersection 函数至少需要检查  $m \times n$  个节点。在我们的集合实现方案中，这些函数都是不高效的。用另外一种方法表示集合——例如，用哈希表而不是链表——能够使得 setIntersection 函数最多检查  $n+m$  个节点。但是，我们的链表实现方案对于那些使用小集合的应用或者不经常调用 setIntersection 函数的应用效率还是不错的，基于链表的实现方案的优点是代码相对简单且容易理解。

如果我们真的对效率很在意，那可以保持 Set<T> 类的接口不变，而用其他实现方案替代我们所讨论的链表方案。如果使用示例 17.25 给出的哈希表实现方案，那么 contains 函数的运行将会更加高效。但是，如果采用哈希表方案的话，会使得遍历集合中的每个元素变得更为困难。哈希表的实现方案不能再去遍历单向链表来获取集合中的每个元素，它必须通过哈希表数组来进行遍历，对于数组中的每个元素，遍历其对应的链表。对哈希表数组中的每个元素进行检查会花费一些多余的时间，因为在单链表实现的集合方案中，不需要进行这一检查。因此，当我们减少了花费在查询元素上的步骤后，我们又增加了遍历每一个元素的步骤。如果你觉得这很麻烦的话，可以通过同时使用链表（便于遍历）和哈希表（便于查询）的方案来解决这一问题。但是，使用这一方案的话，代码的复杂性就增加了。在本章的编程练习 10 中，要求采用哈希表实现方案。

#### 自测练习

23. 为 Set 类写一个名为 difference 的函数，该函数返回两个集合的区别。具体而言，函数应该返回指向一个新集合的指针，新集合里有第一个集合的部分或全部元素，这些元素是第二个集合中所没有的。例如，如果集合 setA 包含元素 {1, 2, 3, 4}，集合 setB 包含 {2, 4, 5}，那么，setA.difference(setB) 应当返回集合 {1, 3}。



## 17.3 迭代器

白兔戴上了眼镜，问道：“我该从哪儿开始呢？陛下。”

“从开始的地方开始吧，一直读到末尾，然后停止。”国王郑重地说道。

刘易斯·卡罗尔，《爱丽丝梦游仙境》

### 迭代器

数据结构中一个重要的概念就是迭代器。迭代器是一个（通常是某个迭代器类的对象）结构体，允许你遍历存储在某个数据结构中的所有元素，这样的话，我们可以在每个数据元素上执行想要执行的任何操作。

#### 迭代器

迭代器是一个（通常是某个迭代器类的对象）结构体，允许你遍历存储在某个数据结构中的所有元素，这样的话，我们可以在每个数据元素上执行想要做的任何动作。

### 指针作为迭代器

在链表的实现中，我们可以很容易地看出迭代器的基本思想，实际上链表就是一个迭代器的原型。链表是迭代器的典型数据结构之一，而指针是迭代器的典型例子之一。你可以使用指针作为一个迭代器，通过指针从链表头开始一次一个节点遍历链表中的所有节点。代码大致如下：

```
Node_Type *iterator;
for (iterator = Head; iterator != NULL;
     iterator = iterator->Link)
    对迭代器所指向的节点执行任何你想执行的操作；
```

*Head* 是一个指向链表头节点的指针，*Link* 是链表中某个节点的成员变量名，它指向该节点的下一个节点。

例如，就像我们在 17.1 节中所讨论的那样，为了输出链表中所有节点的数据，我们可以使用如下代码：

```
IntNode *iterator;
for (iterator = head; iterator != NULL;
     iterator = iterator->getLink( ))
    cout << (iterator->getData( ));
```

示例 17.4 给出了 *IntNode* 的定义。

注意，可以通过使用等于运算符 `==`，来测试两个指针是否指向同一个节点。一个指针就是一段内存的地址。如果两个指针变量包含相同的内存地址，那么它们的比较结果是相等的，而且它们指向同一节点。同样，你可以使用 `!=` 来比较两个指针，判断它们是否不是指向同一个节点。

## 迭代器类

### 迭代器类

迭代器类是一个比指针更具通用性和一般性的概念。通常情况下，它总会有一个指针成员变量作为其数据核心，但这并不是必需的，就像下一个编程示例所表示的那样。例如，迭代器的核心可能是一个数组索引号。迭代器类提供了一些函数和重载运算符，这允许我们使用指向迭代器对象的指针，而不去考虑使用什么东西来表示底层数据结构、节点类型或者基本的位置标记器（指针或数组索引号，或者其他任何东西）。此外，迭代器还提供了一个通用的框架可以适用于很多种数据结构。

一个迭代器类通常有下列重载运算符：

- ++ 重载自增运算符，它使得迭代器指向下一个元素。
- 重载自减运算符，它使得迭代器指向上一个元素。
- == 重载等于运算符，它用于比较两个迭代器，如果它们两个指向同一个元素，则返回 true。
- != 重载不等于运算符，它用于比较两个迭代器，如果它们不是指向同一个节点，则返回 true。
- \* 重载解引用运算符，它用于访问一个元素（它通常返回一个可以进行读/写操作的引用）。

当我们在思考这列运算符的实现和用法时，我们可以通过使用一个链表作为具体的例子来对这些运算符进行学习。学习这些运算符时，要记住链表中的元素就是链表中的数据，而不是整个节点，也不是节点的指针成员变量。除了数据元素外，所有其他一切都是具体的实现细节，这些实现细节对于使用迭代器和数据结构类的程序员应该是透明的，我们应该隐藏这些实现细节。

我们会把迭代器和一些特定的、存储某种数据类型元素的结构类一起来使用。这些数据结构类通常包括下面这些成员函数，这些成员函数用于对某种类的对象进行遍历：

**begin()**：不带参数的成员函数，它返回一个迭代器。返回的迭代器位于（指向）数据结构中的第一个元素。

**end()**：不带参数的成员函数，它返回一个迭代器。返回的迭代器用于测试是否已经遍历了所有的数据结构。如果 *i* 是一个迭代器，而且它所处的位置已经越过了数据结构中的最后一个元素，那么 *i* 等于 **end()**。

通过迭代器，我们可以使用下面的代码来对一个数据结构中的元素进行遍历：

```
for (i = ds.begin(); i != ds.end(); i++)
    process *i // *i 是当前的数据元素。
```

在上述代码中，*i* 表示一个迭代器。在第 19 章中，我们将讨论具有更多元素和做了更多优化的迭代器，这里只是简要地介绍一下。

对我们来说，这种抽象的讨论是很难被理解的，除非能给出具体的例子。因此，让我们通过一个具体示例来学习迭代器的知识吧。

### 迭代器类

迭代器类通常有下列重载运算符：`++`，移向下一个元素；`--`，移向上一个元素；`==`，重载等于号；`!=`，重载不等于号；`*`，重载解引用运算符，它用来访问一个节点元素的数据。

迭代器类的数据结构通常包含下列两个成员函数：`begin()`，它返回一个迭代器，迭代器位于（指向）数据结构中的第一个元素；`end()`，它返回一个迭代器，返回的迭代器用于测试是否已经遍历了所有的数据结构。如果 `i` 是一个迭代器，而且它所处的位置已经越过了数据结构中的最后一个元素，那么 `i` 等于 `end()`。

通过迭代器，我们可以使用如下代码来对一个数据结构 `ds` 中的元素进行遍历：

```
for (i = ds.begin(); i != ds.end(); i++)
    process *i // *i 是当前的数据项。
```

### 示例：迭代器类

示例 17.31 包含一个迭代器类的定义，可以把这个类应用于某些基于链表的数据结构（比如栈或队列）。我们已经把节点类和迭代器类放到了它们自己的命名空间中。这是合理的，因为迭代器和节点类是紧密相关的，而且任何使用节点类的类都可以使用迭代器类。迭代器类没有减运算符，这是因为减运算符的实现依赖于链表细节，而且并不是单单依赖于类型 `Node<T>`（让迭代器类依赖于底层链表是可以的、没有错的。使用链表只是想避免把事情复杂化）。

正如我们所能看到的那样，模板类 `ListIterator` 本质上是一个封装于类中的指针，因此它可以拥有它所需要的成员运算符。重载运算符的定义是简单明了的，而且事实上还很简短，因此我们把所有重载符的定义都作为内联函数来处理。注意使用解引用运算符“`*`”会得到其所指向的节点的数据成员变量。只有数据成员变量是数据。节点中的指针成员变量是具体实现细节的一部分，程序员用户不应该去关注这些具体细节。

如果一个链表使用了模板类 `Node`，那么对于任何一个基于该链表的类，我们都可以把 `ListIterator` 类作为它的迭代器。我们重写了类 `Queue`，使它具有迭代器的功能，把它作为一个说明迭代器的例子。示例 17.32 给出了模板类 `Queue` 的接口。模板类 `Queue` 的定义和之前的那个版本（示例 17.20）基本一样，我们只是在新版本里增加了一个类型定义和下面两个成员函数：

```
Iterator begin() const { return Iterator(front); }
Iterator end() const { return Iterator(); }
//end 迭代器使得 end().current == NULL。
```

我们先来讨论一下成员函数。

成员函数 `begin()` 返回一个定位（“指向”）于队列前端节点的迭代器，队列前端节点实际上是底层链表的头节点。每使用一次自增运算符“`++`”，迭代器就会移向下

一个节点。这样就可以遍历队列  $q$  的所有节点, 继而访问节点上的数据, 具体代码如下:

```
for (i = q.begin( ); Stopping_Condition ; i++)
    process *i // *i 是当前的数据项。
```

其中,  $i$  表示一个迭代器类型的变量。

### 示例 17.31 面向链表的迭代器类

```
1 // 这是头文件 iterator.h. 这是类 ListIterator 的接口,
2 // 类 ListIterator 是一个基于链表的迭代器模板类,
3 // 链表中的元素类型为 T.
4 // 这个文件也包含了一个链表中节点的定义.
5 #ifndef ITERATOR_H
6 #define ITERATOR_H

7 namespace ListNodeSavitch
8 {
9     template<class T>
10     class Node
11     {
12     public:
13         Node(T theData, Node<T>* theLink) : data(theData),
14             link(theLink){}
15         Node<T>* getLink( ) const { return link; }
16         const T& getData( ) const { return data; }
17         void setData(const T& theData) { data = theData; }
18         void setLink(Node<T>* pointer) { link = pointer; }
19     private :
20         T data;
21         Node<T> *link;
22     };

23     template<class T>
24     class ListIterator
25     {
26     public:
27         ListIterator( ) : current(NULL) {}

28         ListIterator(Node<T>* initial) : current(initial) {}
29         const T& operator *( ) const { return current->getData( ); }
30         // 前提条件: 不等于默认构造函数对象;
31         // 也就是说, current != NULL.
32         ListIterator& operator ++( ) // 前缀形式
33         {
34             current = current->getLink( );
35             return *this;
36         }
37         ListIterator operator ++(int) // 后缀形式
38         {
39             ListIterator startVersion(current);
40             current = current->getLink( );
41             return startVersion;
42         }
43         bool operator ==(const ListIterator& rightSide) const
```

注意。通过解引用运算符 \* 得到的只是节点的数据成员, 而不是整个节点。本程序不允许改变节点中的数据。

```

43         { return (current == rightSide.current); }

44         bool operator !=(const ListIterator& rightSide) const
45         { return (current != rightSide.current); }
46         //ListIterator 的默认赋值运算符和拷贝构造函数能正常工作。
47     private:
48         Node<T> *current;
49     };

50 } //ListNodeSavitch

51 #endif //ITERATOR_H

```

成员变量 `end()` 返回一个迭代器，迭代器的当前成员变量是 `NULL`。因此，当迭代器变量 `i` 已经越过了最后一个节点时，下列布尔表达式：

```
i != q.end()
```

由 `true` 变为 `false`。这就是我们所期待的循环终止条件 (`Stopping_Condition`)。队列类和迭代器类允许我们使用如下代码遍历队列中的所有数据：

```

for (i = q.begin(); i != q.end(); i++)
    process *i // *i 是当前的数据元素。

```

记住，当 `i` 为最后一个节点时，`i` 并不等于 `q.end()`。直到迭代器 `i` 已经越过了最后一个节点，处在最后一个节点的后一个位置时，`i` 才等于 `q.end()`。为了记住这个细节知识，我们不妨把 `q.end()` 当作结束标识符 `NULL`。示例 17.33 给出了一个用到这一知识的 `for` 循环样例程序。

注意，在我们的新队列模板类中，类型定义的代码如下所示：

```
typedef ListIterator<T> Iterator;
```

标识符 `typedef` 并不是必需的。你可以总是使用 `ListIterator<T>`，而不去使用类型名 `Iterator`。但使用类型名 `Iterator` 的确会使得代码变得更干净。如果使用 `Iterator` 这种类型定义，那么类 `Queue<char>` 的迭代器可以用如下代码来表示：

```
Queue<char>::Iterator i;
```

这种表示法会使我们更清楚地看到，哪个类正在使用迭代器。

示例 17.34 给出了新模板类 `Queue` 的实现。由于我们所增加的唯一一个成员函数被定义为内联成员函数，因此，新模板类 `Queue` 的实现文件并没有包含什么新的内容，我们只是展示了它的代码布局以及它都包含了哪些指令。

### 示例 17.32 包含迭代器的队列模板类接口文件

```

1 // 这是头文件 queue.h。这是类 Queue 的接口。
2 // 类 Queue 是一个队列模板类，其元素类型为 T，
3 // 它包含一个迭代器类。
4 #ifndef QUEUE_H                                     Node<T> 和 ListIterator<T> 在命名空间中。
5 #define QUEUE_H                                     List NodeSavitch 在文件 iterator.h 中。
6 #include "iterator.h"
7 using namespace ListNodeSavitch;

8 namespace QueueSavitch
9 {
10     template<class T>
11     class Queue
12     {
13     public:
14         typedef ListIterator<T> Iterator;

15         Queue ( );
16         Queue(const Queue<T>& aQueue);
17         Queue<T>& operator =(const Queue<T>& rightSide);
18         virtual ~Queue ( );
19         void add(T item);
20         T remove ( );
21         bool isEmpty ( ) const ;

22         Iterator begin ( ) const { return Iterator(front); }
23         Iterator end ( ) const { return Iterator( ); }
24         //end 迭代器使得 end ( ).current 为 NULL。
25         // 注意，不能对 end 迭代器进行解引用操作。
26     private:
27         Node<T> *front; // 指向链表头。
28                         // 元素在链表头被删除。
29         Node<T> *back;  // 指向链表另一端的
30                         // 节点。
31                         // 在链表的这一端增加节点。
32     };

33 } //QueueSavitch
34 #endif //QUEUE_H

```

### 示例 17.33 应用程序使用包含迭代器的队列模板类

```

1 // 使用包含迭代器的队列模板类的应用程序。
2 #include <iostream>
3 #include "queue.h" // 不需要
4 #include "queue.cpp"
5 #include "iterator.h" // 不需要

```

尽管不需要引入这两个头文件，一些编程者出于文档的目的，还是倾向于导入这两个 include 指令。

```

6  using std::cin;
7  using std::cout;
8  using std::endl;
9  using namespace QueueSavitch;
10 int main( )
11 {
12     char next, ans;
13     do
14     {
15         Queue<char> q;
16         cout << "Enter a line of text:\n";
17         cin.get(next);
18         while (next != '\n')
19         {
20             q.add(next);
21             cin.get(next);
22         }
23         cout << "You entered:\n";
24         Queue<char>::Iterator i;
25         for (i = q.begin( ); i != q.end( ); i++)
26             cout << *i;
27         cout << endl;
28         cout << "Again?(y/n): ";
29         cin >> ans;
30         cin.ignore(10000, '\n');
31     } while (ans != 'n' && ans != 'N');
32     return 0;
33 }

```

如果你的编译器对于下列代码不能正常工作：  
 Queue<char>::Iterator i;  
 试试使用命名空间：  
 ListNodeSavitch;  
 ListIterator<char> i;

### 示例运行结果

```

Enter a line of text:
Where shall I begin?
You entered:
Where shall I begin?
Again?(y/n): y
Enter a line of text:
Begin at the beginning
You entered:
Begin at the beginning
Again?(y/n): n

```

### 示例 17.34 包含迭代器的队列模板类实现文件

```

1  // 这是文件 queue.cpp。这是模板类 Queue 的实现。
2  // 模板类 Queue 的接口在头文件 queue.h 中。
3  #include <iostream>

```

```

4  #include <cstdlib>
5  #include <csddef>
6  #include "queue.h"
7  using std::cout;

8  using namespace ListNodeSavitch;
9  namespace QueueSavitch
10 {
11     template<class T>
12     Queue<T>::Queue( ) : front(NULL), back(NULL)
13 < 自测练习题 16 的答案给出了其余部分的定义。>

14     template<class T>
15     Queue<T>::Queue(const Queue<T>& aQueue)
16 < 自测练习题 19 的答案给出了其余部分的定义。>

17     template<class T>
18     Queue<T>& Queue<T>::operator =(const Queue<T>& rightSide)
19 < 自测练习题 20 的答案给出了其余部分的定义。>
20     template<class T>
21     Queue<T>::~Queue( )

22 < 自测练习题 18 的答案给出了其余部分的定义。>
23     template <class T>
24     bool Queue<T>::isEmpty( ) const
25 < 自测练习题 16 的答案给出了其余部分的定义。>

26     template<class T>
27     void Queue<T>::add(T item)
28 < 自测练习题 17 的答案给出了其余部分的定义。>

29     template<class T>
30     T Queue<T>::remove( )
31 < 自测练习题 17 的答案给出了其余部分的定义。>
32 } //QueueSavitch
33 #endif //QUEUE_H

```

成员函数的定义和之前在 Queue 模板中的定义是一致的。此处给出定义的目的是为了向读者展示文件层次结构以及命名空间的用法。

### 自测练习题

24. 写出下列模板函数声明 inQ 的定义。要求使用迭代器，并使用示例 17.32 给出的类 Queue 的定义。

```

template <class T>
bool inQ(Queue<T> q, T target);
// 如果搜索目标在队列中，返回 true;
// 否则，返回 false。

```



17.4 树

我认为我从未见过像树这样有用的数据结构。

佚名

关于树这种数据结构的详细内容超出了本章的讨论范围。本章的目的是教会大家一些基本技术，来创建和操作基于节点和指针的数据结构。在我们的讨论中，用链表来举例是个不错的选择。但是，单向链表中的节点有一些受限，我们需要注意到这个具体细节：单向链表中的每个节点只有一个指针成员变量指向另外一个节点。而树的每个节点有两个（在一些应用中不止两个）指针成员变量，分别指向其他两个节点。此外，树是一种非常重要并被广泛使用的数据结构。因此，下面简要地介绍一下创建和操作树的一些基本技术。

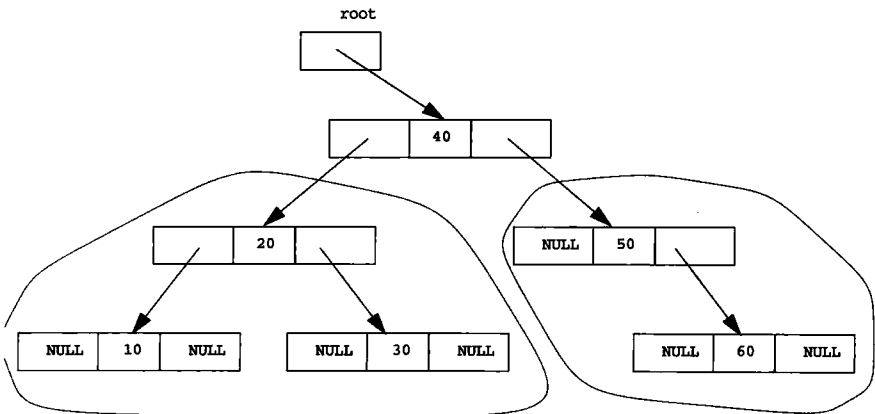
本节将使用递归，我们在第 13 章中详细讨论过这一技术。

树的属性

示例 17.35 以图形的方式表示了数据结构——树的结构。注意，一个树必须有像示例 17.35 所示的这种结构。特别是，沿着链接（指针）所指的某个路径，可以从根节点到达树中的任意一个节点。注意，树中不会出现环路。如果沿着指针所指的方向走，最终会到达一个终点。示例 17.35 也定义了一个节点类，这个类表示内部数据类型为 int 型的这种树的节点。注意，每个节点都有两个链接（两个指针）。这种树被称为二叉树，因为它只有两个链接成员变量。还有一些其他树形数据结构，它们的链接成员变量的个数不一定是 2。但是，我们知道二叉树是最为通用的一个树形数据结构。

二叉树

示例 17.35 二叉树



```

class IntTreeNode
{
public:
    IntTreeNode(int theData, IntTreeNode* left, IntTreeNode* right)
        : data(theData), leftLink(left), rightLink(right){}
private:
    int data;
    IntTreeNode *leftLink;
    IntTreeNode *rightLink;
};

IntTreeNode *root;

```

---

**根节点** 树中名为 `root` 的指针类似于链表（示例 17.1）中的 `head` 指针。`root` 指针所指向的节点被称为根节点。注意，指针 `root` 本身并不是根节点，而是一个指向根节点的指针。从根节点开始沿着链表的指向总能到达树中的任何一个节点。

**叶节点** 术语“树”（tree）似乎并不是很恰当。根在树的顶部，而且树的分支结构看起来更像是一个根的分支结构而不是一个真正的树的结构。关于这个术语命名的秘密在于把图（示例 17.35）上下倒置。这时候，图看起来的确像是一个树的结构，而且根节点的位置处在它应该在的位置。处在分支终点的节点被称为叶节点，叶节点的两个指针成员变量都为 `NULL`。现在，树的整个术语体系看起来比较合理了。

**空树** 类比于空链表，通过设置指针变量 `root` 为 `NULL`，我们可以把一个树表示为空树。

注意，树是具有递归性质的一种结构。每个树都有两个子树，两个子树的根节点分别是树的根节点的两个链接成员变量 `leftLink` 和 `rightLink` 所指向的节点。在示例 17.35 的图中，我们圈出了两个子树。树的这种天然的递归结构使得它特别适用于递归算法。比如一个搜索树的任务，我们要访问树中的每一个节点并对节点的数据进行一些操作（比如输出到屏幕上）。我们的解决方案如下所示：

#### 前序 前序处理

1. 处理根节点数据。
2. 处理左子树。
3. 处理右子树。

我们通过改变这三个步骤的顺序，就可以得到更多搜索处理方案。接下来，我们再给出两种解决方案。

#### 中序 中序处理

1. 处理左子树。
2. 处理根节点数据。
3. 处理右子树。

#### 后序 后序处理

1. 处理左子树。
2. 处理右子树。
3. 处理根节点数据。

## 二叉搜索树

示例 17.35 以一种特殊的方式存储树中的每一个数，这种方式叫作**二叉搜索树存储规则**。下面的蓝色方框中给出了规则的细节。我们把满足二叉搜索树存储规则的树称为**二叉搜索树**。

我们可以发现，如果一个树满足二叉搜索树存储规则，而且我们采用中序遍历的方式输出树中节点的值，那么将会得到一组按从小到大顺序排列的数。

服从二叉搜索树存储规则的树看起来更为矮胖而不是高瘦，使用二叉搜索算法可以迅速得到节点的值，这个算法和示例 13.5 所给出的二叉搜索算法在本质上是一样的。搜索和维护一个二叉树的效率是一个常被探讨的话题，在此，我们不予下面框图的边栏文字：二叉搜索树存储规则。

二叉搜索树  
存储规则**二叉搜索树存储规则**

1. 左子树的所有节点的值都小于根节点的值。
2. 右子树的所有节点的值都大于或等于根节点的值。
3. 此存储规则具有递归性，它适用于任意两个子树中的每一个子树。  
(关于递归的一个最基本的例子是空树，它总是满足此规则。)

**示例：树模板类**

示例 17.36 包含一个二叉搜索树模板类的定义，我们把 `SearchTree` 类作为 `TreeNode` 的友元类。这样的话，我们就可以根据树类成员变量中定义的名字来访问节点成员变量。示例 17.37 给出了 `SearchTree` 的实现，示例 17.38 给出了一个演示程序。

我们设计的模板类的目的是展示一下树这种数据结构的处理模式，它并不是一个完整的例子。一个真正实用的树类会包含更多的成员函数。特别是，一个真正的树类会有一个拷贝构造函数和重载赋值运算符函数。为了节省空间，我们并没有给出这两个函数。

关于类 `SearchTree` 的函数定义，还有其他一些需要注意的地方。函数 `insert` 和 `inTree` 都是重载的。我们需要的是一个单参数的实现版本。在逻辑上表达最清楚的算法是递归算法。对于子树的根节点来说，它需要一个额外的参数。因此，我们为每一个函数定义了一个包含两个参数的私有类型的帮助函数，并在这些包含两个参数的函数中实现了递归算法。这样，单参数函数就可以对两个参数的函数做一个简单的调用，调用时把子树的树根参数设置为整个树的树根参数。类似的这种情况也会出现在重载成员函数名 `inorderShow` 中。函数 `deleteSubtree` 和析构函数的功能是一样的。

**示例 17.36 树模板类的接口文件**

```
1 // 头文件 tree.h。把数据插入到树中的唯一方法是使用 insert 函数。
2 // 因此，树满足二叉搜索树存储规则。
```

```

3 // 函数 inTree 依赖这个文件。
4 // 必须定义运算符<, 给出类型 T 元素的恰当顺序。
5 #ifndef TREE_H
6 #define TREE_H
7 namespace TreeSavitch
8 {
9     template<class T>
10     class SearchTree; // 在前面声明

11     template<class T>
12     class TreeNode
13     {
14     public:
15         TreeNode( ) : root(NULL){}
16         TreeNode(T theData, TreeNode<T>* left, TreeNode<T>* right)
17             : data(theData), leftLink(left), rightLink(right){}
18         friend class SearchTree<T>;
19     private:
20         T data;
21         TreeNode<T> *leftLink;
22         TreeNode<T> *rightLink;
23     };

24     template<class T>
25     class SearchTree
26     {
27     public:
28         SearchTree( ) : root(NULL){}
29         virtual ~SearchTree( );
30         void insert(T item); // 把元素添加到树中。
31         bool inTree(T item) const ;
32         void inorderShow( ) const ;
33     private:
34         void insert(T item, TreeNode<T>*& subTreeRoot);
35         bool inTree(T item, TreeNode<T>* subTreeRoot) const ;
36         void deleteSubtree(TreeNode<T>*& subTreeRoot);
37         void inorderShow(TreeNode<T>* subTreeRoot) const ;
38         TreeNode<T> *root;
39     };

40 } //TreeSavitch

41 #endif

```

SearchTree 模板类应该有一个拷贝构造函数和一个赋值运算符以及其他成员函数。但是, 为了使得这个例子的代码短一点, 我们省略了这些函数。一个真正的模板类应该包含更多的成员函数和重载运算符。

### 示例 17.37 树模板类的实现文件

```

1 // 这是实现文件 tree.cpp。这是模板类 SearchTree 的实现。
2 // 接口在文件 tree.h 中。
3 namespace TreeSavitch
4 {
5     template<class T>

```

```

6      void SearchTree<T>::insert(T item, TreeNode<T>*& subTreeRoot)
7      {
8          if (subTreeRoot == NULL)
9              subTreeRoot = new TreeNode<T>(item, NULL, NULL);
10         else if (item < subTreeRoot->data)
11             insert(item, subTreeRoot->leftLink);
12         else if (item >= subTreeRoot->data)
13             insert(item, subTreeRoot->rightLink);
14     }

15     template<class T>
16     void SearchTree<T>::insert(T item)
17     {
18         insert(item, root);
19     }

20     template<class T>
21     bool SearchTree<T>::inTree(T item, TreeNode<T>* subTreeRoot) const
22     {
23         if (subTreeRoot == NULL)
24             return false;
25         else if (subTreeRoot->data == item)
26             return true;
27         else if (item < subTreeRoot->data)
28             return inTree(item, subTreeRoot->leftLink);
29         else if (item >= subTreeRoot->data)
30             return inTree(item, subTreeRoot->rightLink);
31     }

32     template <class T>
33     bool SearchTree<T>::inTree(T item) const
34     {
35         return inTree(item, root);
36     }

37     template<class T> // 使用 iostream.
38     void SearchTree<T>::inorderShow(TreeNode<T>* subTreeRoot) const
39     {
40         if (subTreeRoot != NULL)
41         {
42             inorderShow(subTreeRoot->leftLink);
43             cout << subTreeRoot->data << " ";
44             inorderShow(subTreeRoot->rightLink);
45         }
46     }

47     template<class T> // 使用 iostream.
48     void SearchTree<T>::inorderShow( ) const
49     {
50         inorderShow(root);
51     }

52     template<class T>
53     void SearchTree<T>::deleteSubtree(TreeNode<T>*& subTreeRoot)
54     {
55         if (subTreeRoot != NULL)

```

如果所有数据都是通过使用函数 insert 输入的, 那么所构建的树将会满足二叉搜索树存储规则。

Tree 中的函数使用二叉搜索算法, 它是示例 13.5 所给出的算法的变体。

使用中序遍历方式遍历树。

```

56         {
57             deleteSubtree(subTreeRoot->leftLink);  ← 使用后序遍历方式遍历树。
58             deleteSubtree(subTreeRoot->rightLink);

59             //subTreeRoot 现在指向一个单节点树。
60             delete subTreeRoot;
61             subTreeRoot = NULL;
62         }
63     }

64     template<class T>
65     SearchTree<T>::~~SearchTree( )
66     {
67         deleteSubtree(root);
68     }
69 } //TreeSavitch

```

---

### 示例 17.38 使用树模板类的演示程序

---

```

1  // 树模板类的演示程序。
2  #include <iostream>
3  #include "tree.h"
4  #include "tree.cpp"
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using TreeSavitch::SearchTree;

9  int main( )
10 {
11     SearchTree<int> t;

12     cout << "Enter a list of nonnegative integers.\n"
13         << "Place a negative integer at the end.\n";
14     int next;
15     cin >> next;
16     while (next >= 0)
17     {
18         t.insert(next);
19         cin >> next;
20     }

21     cout << "In sorted order: \n";
22     t.inorderShow( );
23     cout << endl;

24     return 0;
25 }

```

## 示例运行结果

```
Enter a list of nonnegative integers.
Place a negative integer at the end.
40 30 20 10 11 22 33 44 -1
In sorted order:
10 11 20 22 30 33 40 44
```

最后，请注意我们使用 `insert` 函数创建了一个满足二叉搜索树存储规则的树，这一点很重要。由于 `insert` 函数是创建树模板类的唯一可用函数，那么树模板类对象将总是满足二叉搜索树存储规则。函数 `inTree` 的算法默认树对象满足二叉搜索树存储规则。这会使得树的搜索非常高效。当然，这也意味着必须定义运算符“<”，使其能够适用于存储在树中的数据类型为 `T` 的数据。当把“<”运算符应用于数据类型为 `T` 的数值时，为了保证它能正常工作，它应该满足如下规则：

- 可传递性： $a < b$  并且  $b < c$ ，那么  $a < c$ 。
- 非对称性：如果  $a$  和  $b$  不相等，那么要么  $a < b$ ，要么  $b < a$ ，两者不能同时成立。
- 非自反性： $a < a$  绝不可能成立。

大部分自然次序满足这些规则。<sup>2</sup>

## 自测练习题

25. 对下列成员函数进行定义，可以把它们添加到示例 17.36 的类 `SearchTree` 中。这些函数分别以前序遍历树和后序遍历树的方式显示数据。就像我们在函数 `SearchTree<T>::inorderShow` 中那样，请为每一个函数定义一个私有的帮助函数。

```
void SearchTree<T>::preorderShow() const
void SearchTree<T>::postorderShow() const
```

## 本章小结

- 节点是一个结构体或者类对象，它有一个或多个指针成员变量。这些节点之间通过指针成员变量联系在一起构成一个数据结构，当程序运行时，这种数据结构的大小可以动态增长或收缩。
- 链表是一串节点，在链表中，每个节点包含一个指向下一个节点的指针。

<sup>2</sup> 注意，通常情况下我们既有“小于或等于”的关系式，也有“小于”关系式。但这些规则仅仅适用于“小于”关系式。事实上，我们甚至可以处理一种更弱的次序概念，称为严格弱次序，这是在第 19 章中定义的，它比我们一般所碰到的情况描述得更加详细。

- 一个链表（或者其他链式数据结构）的结束是通过把指针变量设置为 NULL 来表示的。
- 双向链表中的节点有两个链接——一个指向前一个节点，一个指向后一个节点。这会使得插入和删除操作变得稍微容易些。
- 栈是一种先进后出的数据结构。而队列是一种先进先出数据结构。两者都可以使用链表来实现。
- 哈希表是用于高效存储和获取数据的一种数据结构。哈希函数可以用来把一个对象映射为一个值，这个值被用来对对象进行索引。
- 链表可以用于实现集合，包括一些普通的操作，如取并集、交集以及设置成员。
- 迭代器是一种结构（通常是某种迭代器类的对象），它允许遍历存储在数据结构中的元素。
- 树这种数据结构，它的节点有两个或者多个指向其他节点的成员变量指针。如果树满足二叉搜索树存储规则，那么可以设计出快速找到树中节点的函数。

## 自测练习题答案

1. Sally  
Sally  
18  
18  
注意，(\*head).name 和 head->name 是一样的。类似还有，(\*head).number 和 head->number 也是一样的。
2. 最佳答案是：  
  
head->next = NULL;  
  
但是下列代码也是正确的：  
  
(\*head).next = NULL;
3. head->item = "Wilbur's brother Orville";
4. 

```
class NodeType
{
public:
    NodeType(){}
    NodeType(char theData, NodeType* theLink)
        : data(theData), link(theLink){}
    NodeType* getLink() const { return link; }
    char getData() const { return data; }
    void setData(char theData) { data = theData; }
    void setLink(NodeType* pointer) { link = pointer; }
private:
    char data;
    NodeType *link;
```



```
};
typedef NodeType* PointerType;
```

5. 值 NULL 用于表示空链表。

```
6. p1 = p1->next;
```

```
7. Pointer discard;
   discard = p2->next; //discard 指向将要被删除的节点。
   p2->next = discard->next;
```

上述代码可以从链表中删除一个节点。但是，如果出于某种原因，并没有使用这个节点，那应该调用 delete 来删除这个节点，代码如下：

```
delete discard;
```

```
8. p1 = p1->getLink( );
```

```
9. Pointer discard;
   discard = p2->getLink( ); // 指向将要被删除的节点。
   p2->setLink(discard->getLink( ));
```

上述代码可以从链表中删除一个节点。但是，如果出于某种原因，并没有使用这个节点，那应该调用 delete 来删除这个节点，代码如下：

```
delete discard;
```

10. a. 把一个新元素插入到一个大的链表中的某个已知位置要比插入到一个大的数组的已知位置处更为高效。如果你打算把数据插入到链表中，不管链表有多大，你需要大约五个操作，其中大部分是指针赋值操作。如果把数据插入到数组中，平均而言，你将不得不动数组中一半的数据。

对于小的链表，答案是 c，理由同上。

```
11. void insert(DoublyLinkedListNodePtr afterMe, int theData)
```

```
{
    DoublyLinkedListNode* newNode = new
        DoublyLinkedListNode(theData, afterMe,
            afterMe->getNextLink( ));
    afterMe->setNextLink(newNode);
    if (newNode->getNextLink( ) != NULL)
    {
        newNode->getNextLink( )->setPreviousLink(newNode);
    }
}
```

12. 对于双向链表来说，插入和删除操作相对要容易些。因为我们不再需要一个独立的变量来保存前面的节点。我们可以通过一个节点的前向链接来访问它的前一个节点。但是，这两种操作需要更新更多的链接（例如，不仅仅是更新前向链接，而需要对前向和后向链接都进行更新）。

13. 注意：这个函数和示例 17.5 所给出的 headInsert 函数本质上是一样的。

```
template<class T>
void Stack<T>::push(T stackFrame)
{
    top = new Node<T>(stackFrame, top);
}
```

## 14. // 使用 cstddef.

```

template<class T>
Stack<T>::Stack(const Stack<T>& aStack)
{
    if (aStack.isEmpty( ))
        top = NULL;
    else
    {
        Node<T> *temp = aStack.top; //temp 自顶向下遍历 aStack 的所有节点。
        Node<T> *end; // 指向新栈的底部。
        end = new Node<T>(temp->getData( ), NULL);
        top = end;
        // 创建了第一个节点, 并为其添加了数据。
        // 现在在第一个节点之后增加了新节点。
        temp = temp->getLink( ); // 把 temp 移到第二个节点。
        // 如果没有第二个节点, 则移到 NULL。
        while (temp != NULL)
        {
            end->setLink(
                new Node<T>(temp->getData( ), NULL));
            temp = temp->getLink( );
            end = end->getLink( );
        }
        //end->link == NULL;
    }
}

```

## 15. template&lt;class T&gt;

```

Stack<T>& Stack<T>::operator =(const Stack<T>& rightSide)
{
    if (top == rightSide.top) // 如果两个栈相同
        return *this ;
    else // 把左端对象所占内存返还给自由存储区。
    {
        T next;
        while (! isEmpty( ))
            next = pop( ); // 调用 delete 删除。
    }
    if (rightSide.isEmpty( ))
    {
        top = NULL;
        return *this ;
    }
    else
    {
        Node<T> *temp = rightSide.top; //temp 从右端栈的顶部节点
        // 移到其底部节点。
        Node<T> *end; // 指向左端栈对象的底部节点。
        end = new Node<T>(temp->getData( ), NULL);
        top = end;
        // 创建了第一个节点, 并为其填充了数据。
        // 现在, 在第一个节点之后增加了一个新节点。
        temp = temp->getLink( ); // 把 temp 移向第二个节点。如果没有第二个节点,
        // 就把它设置为 NULL。
        while (temp != NULL)

```

```

        {
            end->setLink(
                new Node<T>(temp->getData(), NULL));
            temp = temp->getLink();
            end = end->getLink();
        }
        //end->link == NULL;
        return *this;
    }
}

```

16. 下面的代码应该放在命名空间 QueueSavitch 中：

```

// 使用 cstdint.
template<class T>
Queue<T>::Queue() : front(NULL), back(NULL)
{
    // 特意将其设置为空。
}
// 使用 cstdint.
template<class T>
bool Queue<T>::isEmpty() const
{
    return (back == NULL); //front == NULL 也能工作。
}

```

17. 下列代码应该被置于命名空间 QueueSavitch 中：

```

// 使用 cstdint.
template<class T>
void Queue<T>::add(T item)
{
    if (isEmpty())
        front = back = new Node<T>(item, NULL); // 设置 front 和 back,
        // 让它们指向唯一一个节点。
    else
    {
        back->setLink(new Node<T>(item, NULL));
        back = back->getLink();
    }
}
// 使用 cstdlib 和 iostream.
template<class T>
T Queue<T>::remove()
{
    if (isEmpty())
    {
        cout << "Error: Removing an item from an empty queue.\n";
        exit(1);
    }
    T result = front->getData();
    Node<T> *discard;
    discard = front;
    front = front->getLink();
    if (front == NULL) // 如果你删除了最后一个节点。
        back = NULL;
}

```

```

        delete discard;
        return result;
    }

```

18. 下列代码应该被置于命名空间 QueueSavitch 中：

```

template<class T>
Queue<T>::~Queue( )
{
    T next;
    while (! isEmpty( ))
        next = remove( ); //remove 调用 delete。
}

```

19. 下列代码应该被置于命名空间 QueueSavitch 中：

```

// 使用 cstdint.
template<class T>
Queue<T>::Queue(const Queue<T>& aQueue)
{
    if (aQueue.isEmpty( ))
        front = back = NULL;
    else
    {
        Node<T> *temp = aQueue.front; //temp 从队列 aQueue 的前端节点
        // 按顺序逐一移到其后端节点。
        back = new Node<T>(temp->getData( ), NULL);
        front = back;
        // 创建了第一个节点，并为其填充了数据。
        // 现在，在第一个节点之后增加了新节点。
        temp = temp->getLink( ); //temp 现在指向第二个节点，如果第二个
        // 节点为空，则指向 NULL。
        while (temp != NULL)
        {
            back->setLink(new Node<T>(temp->getData( ), NULL));
            back = back->getLink( );
            temp = temp->getLink( );
        }
        //back->link == NULL
    }
}

```

20. 下列代码应该被置于命名空间 QueueSavitch 中：

```

// 使用 cstdint.
template<class T>
Queue<T>& Queue<T>::operator =(const Queue<T>& rightSide)
{
    if (front == rightSide.front) // 如果两个队列是同一个队列
        return *this;
    else // 把左端对象所占内存返还给自由存储区。
    {
        T next;
        while (! isEmpty( ))
            next = remove( ); //remove 调用 delete。
    }
    if (rightSide.isEmpty( ))

```

```

    {
        front = back = NULL;
        return *this;
    }
    else
    {
        Node<T> *temp = rightSide.front; //temp 从 rightSide 的前端节点
        // 按顺序逐一移到其后端节点。
        back = new Node<T>(temp->getData(), NULL);
        front = back;
        // 创建了第一个节点, 并为其填充了数据。
        // 现在, 在第一个节点之后增加了新节点。
        temp = temp->getLink(); //temp 现在指向第二个节点。如果第二个
        // 节点为空, 则指向 NULL。
        while (temp != NULL)
        {
            back->setLink(
                new Node<T>(temp->getData(), NULL));
            back = back->getLink();
            temp = temp->getLink();
        }
        //back->link == NULL;
        return *this;
    }
}

```

21. 最简单的哈希函数是使用求模运算符把 ID 号映射到哈希表的一个值范围：

hash = ID % N; //N 是哈希表大小。

```

22. void HashTable::outputHashTable()
{
    for (int i=0; i<SIZE; i++)
    {
        Node<string> *next = hashArray[i];
        cout << "In slot " << i << endl;
        cout << " ";
        while (next != NULL)
        {
            cout << next->getData() << " ";
            next = next->getLink();
        }
        cout << endl;
    }
}

```

23. 这个代码和函数 intersection 的代码很类似, 但是它多了一项操作, 即如果元素不在 otherSet 中就增加元素:

```

template<class T>
Set<T>* Set<T>::setDifference(const Set<T>& otherSet)
{
    Set<T> *diffSet = new Set<T>();
    Node<T>* iterator = head;
    while (iterator != NULL)
    {

```

```

        if (!otherSet.contains(iterator->getData( )))
        {
            diffSet->add(iterator->getData( ));
        }
        iterator = iterator->getLink( );
    }
    return diffSet;
}

```

24. using namespace ListNodeSavitch;

using namespace QueueSavitch;

template<class T>

bool inQ(Queue<T> q, T target)

```

{
    Queue<T>::Iterator i;
    i = q.begin( );
    while ((i != q.end( )) && (*i != target))
        i++;
    return (i != q.end( ));
}

```

注意在下列语句中，return 语句不能正常工作，因为它会引起对 NULL 进行解引用操作，这是非法的。这个错误是运行时错误，而不是编译错误。

```
return (*i == target);
```

25. 需要对模板类 SearchTree 增加函数声明，这些只是定义。

```

template<class T> // 使用 iostream.
void SearchTree<T>::preorderShow( ) const
{
    preorderShow(root);
}
template<class T> // 使用 iostream.
void SearchTree<T>::preorderShow(
    TreeNode<T>* subTreeRoot) const
{
    if (subTreeRoot != NULL)
    {
        cout << subTreeRoot->data << " ";
        preorderShow(subTreeRoot->leftLink);
        preorderShow(subTreeRoot->rightLink);
    }
}
template<class T> // 使用 iostream.
void SearchTree<T>::postorderShow( ) const
{
    postorderShow(root);
}
template<class T> // 使用 iostream.
void SearchTree<T>::postorderShow(
    TreeNode<T>* subTreeRoot) const
{
    if (subTreeRoot != NULL)
    {
        postorderShow(subTreeRoot->leftLink);

```

```

        postorderShow(subTreeRoot->rightLink);
        cout << subTreeRoot->data << " ";
    }
}

```

## 编程练习

1. 编写一个 void 函数，该函数的功能是接收一个整数型链表，并将整数型链表中节点的顺序倒置。该函数有一个指向链表头节点的指针，将其作为传引用参数。函数调用后，指针指向新链表的头节点，新链表和原链表的节点都是一样的，只是和原链表节点顺序相反。注意，所编写的函数不能创建或删除任何节点。它只是简单地改变一下链表内节点的顺序。最后，编写测试程序对该函数进行验证。
2. 编写一个名为 mergeLists 的函数，该函数接收两个传引用参数，这两个参数是两个指针变量，分别指向节点类型为 int 的两个链表的头节点。假设两个链表是有序链表，即第一个节点数值最小，第二个次之，依此类推。函数返回一个指向新链表头节点的指针，新链表包含原来两个链表的所有节点。这个较长的新链表中的节点也是按照从小到大的顺序排序。注意，所编写的函数不能创建或删除任何节点。当函数调用结束时，两个指针变量参数的值为 NULL。
3. 设计和实现一个多项式类。采用链表实现多项式  $a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ ，每个节点包含一个表示  $x$  幂值的整数值和一个表示相应系数的整数值。这个类的操作包括加、减、乘以及多项式求值。重载 +、- 和 \* 运算符来进行加、减及乘运算。通过一个参数类型为 int 的成员函数来实现多项式求值。这个成员函数的返回值是通过为  $x$  代入参数并执行多项式中的运算而得到的。

所设计的类包含四个构造函数：默认构造函数，拷贝构造函数，含有一个输入参数，输入参数类型为 int 的构造函数，这个构造函数生成的多项式只包含一个常数项，常数项等于构造函数的输入参数值；含有两个输入参数，两个输入参数类型均为 int 的构造函数，这个构造函数生成的多项式的系数和幂由两个输入参数给出（由单输入参数的构造函数所生成的多项式形式简单，只包含一个常数项  $a_0$ 。由双输入参数的构造函数所生成的多项式，其形式略微复杂，为  $a_i x^i$ ）。所设计的类需要包含一个合适的析构函数，包含能够对多项式进行输入和输出的成员函数。

当用户要输入一个多项式时，可以用如下方式表示：

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

但是，如果一个系数  $a_i$  为 0，用户可以省略  $a_i x^i$  项。例如，下列多项式：

$$3x^4 + 7x^2 + 5$$

可以用如下方式输入：

$$3x^4 + 7x^2 + 5$$

也可以采用下面这种方式输入：

$$3x^4 + 0x^3 + 7x^2 + 0x^1 + 5$$

如果一个系数是负的，那么用负号替换加号即可，如下例所示：

$$3x^5 - 7x^3 + 2x^1 - 8$$

$$-7x^4 + 5x^2 + 9$$

正如上面两个例子的第二个例子所示，多项式前面的负号只作用于第一个系数，而不是整个多项式。多项式的输出格式与输入格式一致。在输出时，凡是系数为0的项一律不输出。为了简化输入，假设总是一行输入一个多项式，并且多项式总是会有一个常数项  $a_0$ 。如果没有常数项，用户可以输入0来表示常数项，如下所示：

$$12x^8 + 3x^2 + 0$$

4. a. 在示例 17.36 的注解中，我们提到真正的 `SearchTree` 模板类应该有一个拷贝构造函数、一个重载赋值运算符、其他重载运算符和其他成员函数。使用示例 17.36 的代码，增加下列函数和重载运算符的声明：

默认构造函数、拷贝构造函数、`delete`、重载运算符、`=`、`makeEmpty`、`height`、`size`、`preOrderTraversal`、`inOrderTraversal` 及 `postOrderTraversal`。函数 `preOrderTraversal`、`inOrderTraversal` 及 `postOrderTraversal` 都会去调用一个全局函数 `process` 来处理它们所遇到的节点。函数 `process` 是类 `SearchTree` 的友元函数。在本道练习题中，它只是一个桩程序。

给出这些函数的前提条件和运行结果，来描述每个函数具体有什么功能。

函数 `height` 没有输入参数，它返回树的高度。树的高度是指所有节点的最大高度。节点的高度是节点与根节点之间的链接数。

函数 `size` 没有输入参数，它返回树中所有节点的个数，

函数 `makeEmpty` 删除树中的所有节点，并释放节点所使用的内存以重用。`makeEmpty` 函数将树的根节点指针设置为 `NULL`。

- b. 成员函数、友元函数以及重载运算符的实现。注意，这里所列的一些函数已经在本书中实现了。你应该充分利用本书的代码，也应该仔细测试你所编写的这些代码。
- c. 为树类设计和实现一个迭代器类。你需要决定对于 `searchTree` 来说，`begin` 和 `end` 元素的含义是什么，运算符 `++` 所指的下一个元素是什么。

提示 1：也许你需要维护一个私有的 `size` 变量，通过插入来增加它的大小，通过删除来减小它的大小，它的值由 `size` 函数来返回。另一种方案（如果你确信调用 `size` 不会太频繁的话，可以使用这种方案）是当你需要知道 `size` 时，



通过遍历树来计算 size。也可以采用类似的一些技术方案，但可能需要更加复杂的实现细节来实现 height 函数。

提示 2：一次写几个函数作为一组。写完每组后进行编译和测试。你会很喜欢这种方式。

提示 3：在你编写运算符“=”以及拷贝构造函数前，要注意到它们有一项共同的任务，就是复制一个树。写一个 copyTree 函数来抽象出这个共同的任务，以实现代码复用。接下来，使用这个复用代码实现这两个重要的函数。

提示 4：函数 makeEmpty 和析构函数有一个销毁树的公共任务。

5. 在古代大陆，美丽的公主 Eve 有许多求婚者。她决定用如下方式来挑选这些求婚者。首先，把所有求婚者排成一行，为每个求婚者分配一个号。第一个求婚者的号为 1，第二个求婚者为 2，依此类推，直到最后一个求婚者。他的数字是  $n$ 。Eve 从第一个求婚者开始，按照顺序选出三个（因为她的名字有三个字母）求婚者，第三个求婚者将被剥夺机会，并将其从队伍中排除。Eve 继续这一过程，按顺序再选出三位，然后排除这一组中的第三位。依此类推，当她来到队伍的最后时，她将从头开始重新继续这一过程。

例如，如果有六位求婚者，那么排除过程如下所示：

123456	求婚者的初始队列，从 1 号开始。
12456	3 号求婚者被排除，继续从第四位求婚者开始选择
1245	6 号求婚者被排除，从 1 号重新继续选择
125	4 号求婚者被排除，从 5 号继续选择
15	2 号求婚者被排除，从 5 号继续选择
1	5 号求婚者被排除，1 是最终的幸运求婚者

编写一个程序，创建循环链表来决定当求婚者数目为  $n$ ，为了和公主结婚，你应该站在哪个位置。所编写的程序应当通过删除节点来模拟排除求婚者的过程，被删除的节点对应于每一轮中被排除的求婚者。注意，别写出内存泄漏的代码。

6. 修改 17.2 节所给出的 Queue 模板类，使它成为一个优先队列的实现代码。优先队列和规则队列很相似，只是优先队列的每一个元素还有一个优先权。对于这一问题，把优先权设为整数，其中，0 表示最高优先权，值越大所表示的优先权越低。

remove 函数应当返回并删除具有最高优先权的元素。例如：

```
q.add('X', 10);
q.add('Y', 1);
q.add('Z', 3);
cout << q.remove(); // 返回 Y
cout << q.remove(); // 返回 Z
cout << q.remove(); // 返回 X
```

给队列中的数据赋不同的优先权值，优先权值的大小按不同的顺序排列（例如，升序、降序和混合排序），以此来测试队列。通过在 `remove()` 函数中执行线性搜索，可以实现优先队列。在将来的课程中，可以学习一个被称为堆的数据结构，堆可以提供一种更为高效的方式来实现优先队列。

7. 为示例 17.28 和示例 17.29 的 `Set` 类增加一个 `remove` 函数、一个 `cardinality` 函数、一个迭代器。写一个合适的测试程序。
8. 示例 17.24 和示例 17.25 把一个字符串哈希为一个整数，并在哈希表中存储了相同的字符串。请修改程序，使得哈希表中不要存储字符串而是存储 `Employee` 对象。定义 `Employee` 类使得它包含私有字符串成员变量，这些成员变量为：名字和姓氏的组合变量、工作头衔，以及电话号码。程序应包含这些成员变量所对应的 `get` 和 `set` 方法。程序使用员工名作为哈希函数的输入。程序需要对链表做一些修改，因为所创建的类 `LinkedList2` 仅能用于字符串链表。为了最大化程序的普适性，请修改哈希表，使得它可以使用通用类型的数据。还需要为哈希表类增加一个 `get` 函数，该函数返回哈希表中输入的员工名的 `Employee` 对象。可能还需要为 `Employee` 类定义 `==` 和 `!=` 运算符。通过增加和获取几个名字来测试所编写的程序，包括增加一些映射到哈希表中同一元素的名字。
9. 示例 17.24 到示例 17.26 给出了拼写检查程序的开头部分。重新优化程序使得它更加实用。修改后的程序应在一个文本文件中读取内容，解析每一个单词并判断每一个单词是否在哈希表中。如果不在，输出行号以及可能拼写错误的单词。剔除文本中出现的任何标点符号。使用 `words.txt` 文件，它是哈希表字典的基础。这个文件包含 45 407 个普通单词和一些英文名。注意，一些单词是大写的。在一个短文中测试一下所编写的拼写检查程序。

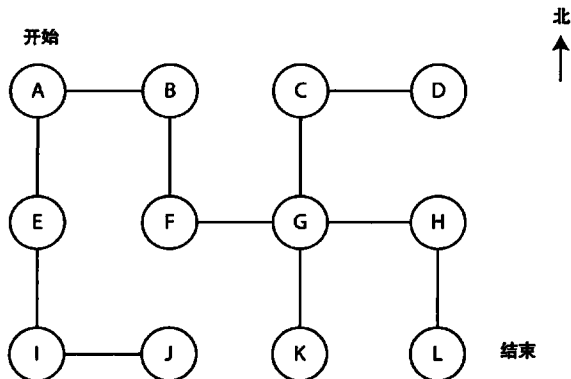
10. 改写示例 17.28 和示例 17.29 所示的 `Set<T>` 类，使它使用哈希表而不是链表存储数据。公共函数的头部保持不变，这样的话，像示例 17.30 所示的演示程序不需要做任何改变也能正常工作。增加构造函数允许新类 `Set<T>` 的用户可以指定哈希表数组的大小。

数据类型为 `T` 的类必须覆盖 `<<` 运算符。为了把 `<<` 的返回值转换为字符串，进行如下操作：

```
# include <sstream>
...
stringstream temp;
temp << instance of Class;
string s = temp.str();
```

再接受一个挑战，既使用哈希表，同时又使用链表来实现集合。添加到集合中的元素应当使用两种数据结构来存储。任何查询元素的操作应当使用哈希表，任何需要遍历整个元素的操作使用链表。

11. 下图中的结构被称为图。其中，圈被称为节点，线被称为边。边连接两个节点。你可以把图看成房间和通道形成的迷宫。节点可被认为是房间，边把一个个房间连接起来。注意下图中每个节点最多只有四个边。



编写一个程序，使用一个指向 Node 类实例的引用来实现前面所描述的迷宫。图中的每个节点对应于一个 Node 实例。边对应于连接节点之间的链接，可以把它表示为 Node 的一个实例变量，这个实例变量指向另一个 Node 类。节点 A 中的用户开始行走，该用户的目标是到达节点 L。程序应当输出可能的移动方向：北、南、东和西。执行结果样例如下所示：

```
You are in room A of a maze of twisty little passages, all alike.
You can go east or south.
E
You are in room B of a maze of twisty little passages, all alike.
You can go west or south.
S
You are in room F of a maze of twisty little passages, all alike.
You can go north or east.
E
```

12. 首先，完成编程练习 11。接着，写一个递归算法，找到从节点 A 到节点 L。算法可适用于任何一组起点和终点，而不仅仅是节点 A 和 L。算法也可以适用于环路，比如节点 E 和 F 之间连成的环路。

## 18.1 异常处理基础 696

异常处理的简单样例 697

定义自己的异常类 704

多个抛出和捕获 704

陷阱：首先捕获比较明确的异常 707

提示：异常类可以很简单 708

在函数中抛出异常 708

异常说明 710

陷阱：派生类中的异常说明 712

## 18.2 异常处理的编程技术 713

抛出异常的时机 713

陷阱：未被捕获的异常 714

陷阱：嵌套try-catch块 715

陷阱：过度使用异常 715

异常类层次结构 715

可用内存测试 716

再次抛出异常 716

# 第 18 章 异常处理

例外证明了规则。

一句格言

## 概述

有一种编写程序的方法,首先假设没有什么不正常或者不正确的事情会发生。例如,当一个程序将头节点从链表中删除时,或许你会假设链表是非空链表。一旦你的程序已经可以正常工作于主要场景时,即程序总会按照事先所设想的情况正确执行,这时候你可以增加一些代码来处理异常情况。C++ 为实现这种处理方式提供了一种方法。首先,在编写代码时并不去考虑会有什么不正常的情况发生。接着,使用 C++ 的异常处理工具来添加代码以处理那些不正常情况。

异常处理通常用来对错误情况进行处理,但把它看成是用来处理意外情况的一种方法可能更好些。毕竟,如果代码已经对“error”进行了正确处理,那它就不再是错误了。

一些函数在使用过程中可能会遇到一些特殊的情况,需要对这些特殊情况进行不同的处理,异常最重要的用途或许就是为这种函数提供一种解决方案。很多程序可能会去调用这种函数,一些程序会采用这种函数提供的某一种方式进行处理,另外一些程序可能会采用这种函数提供的其他方式进行处理。例如,如果在一个函数中有除零运算,那么可能有些程序会在调用该函数的时候终止运行;而对于其他一些调用该函数的程序,可能需要我们提供不同的处理方案。在接下来的章节中,我们将会看到,当出现特殊情况时,可以通过定义使这种函数抛出异常,异常允许在函数体外处理一些特殊的情况。这样的话就可以针对特殊情况调用不同的函数进行处理。

在 C++ 中,异常处理的过程如下:库代码或者自己所写的代码提供一种异常出现时发出信号的机制,这被称为抛出异常(throwing an exception)。在程序的另外一个地方放入处理异常的代码,这被称为处理异常(handling the exception)。这种编程方式会使得代码看起来更加简洁。接下来,我们还会继续详细解释如何在 C++ 中进行异常处理。

本章大部分内容只用到了第 1 章到第 9 章中的知识。但本章中“陷阱:派生类中的异常说明”和“异常类层次结构”这两部分使用了第 14 章的相关知识。“可用内存测试”这部分使用了第 17 章中的知识。读者可以略过上述几部分内容,而不会影响本章的连续性。“异常说明”这一部分有一段提到了派生类(见第 14 章),但这段内容也可以跳过。

## 18.1 异常处理基础

哦,我编写的程序在大部分情况下都没问题。我不知道在这种情况下也必须保证程序能正常运行。

学习计算机科学的学生,想拿成绩 A

我们应该谨慎地使用异常处理，它应该被用于一些会出现复杂情况的场合，而不应该用在我们所提供的一些简单的介绍性例子中。因此，虽然我们会通过简单的例子来交给大家一些 C++ 异常的具体细节知识，但要知道，这些简单例子通常并不会使用异常处理。虽然这种学习 C++ 异常处理的方式是合理的，但是要记住，这些最初的学习示例只是简单的样例，在实际编程项目中，对于这种简单的情形是不需要使用异常处理的。

### 异常处理的简单样例

在这个样例中，假设牛奶是人们生活中所不能或缺的一种重要食物，但是我们仍然希望我们的程序能够处理一种非常不可能发生的情形。我们离不开牛奶的基本代码如下所示：

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts / static_cast<double>(milk);
cout << donuts << " donuts.\n"
    << milk << " glasses of milk.\n"
    << "You have "<<dpg
    << "donuts for each glass of milk.\n";
```

如果没有牛奶，那么代码将会出现除零运算，这将会产生错误。为了处理没有牛奶这种特殊情况，我们可以增加一个测试。示例 18.1 给出了专门针对这种特殊情况增加了代码后的完整程序。示例 18.1 所示的程序没有使用异常处理。现在，我们来看看使用 C++ 的异常处理工具如何重写程序。

在示例 18.2 中，我们使用异常重写了示例 18.1 的程序。这只是个简单的样例，我们可能并不会在这种简单的情形下使用异常。但是，它的确是一个如何使用异常的简单样例。虽然整个程序看起来并不简洁，但至少在 try 语句和 catch 语句之间的代码块比较简洁，这也表明了使用异常的好处。看看在 try 和 catch 语句之间的代码块，它基本上和示例 18.1 所示的代码一致，除了没有使用一个大的 if-else 语句（示例 18.1 中的突出显示部分）外，新程序的 if 语句更小（加上一些简单的不带分支的语句）：

```
if (milk <= 0)
    throw donuts;
```

if 语句是说如果没有牛奶，那么就做一些异常处理。异常处理放在 catch 语句之后。它的思想是把正常情况的处理放在语句 try 代码块中，而把异常情况的处理放到了 catch 代码块中。这样我们就把正常情况和异常情况隔离了。在这个简单的样例程序中，我们不能真正体会到异常机制的优势，但是一些其他场景会证明它是很有用的。下面将介绍异常的详细机制。

#### 示例 18.1 处理特殊情况但未使用异常处理

```
1 #include <iostream>
2 using std::cin;
```

```

3  using std::cout;

4  int main( )
5  {
6      int donuts, milk;
7      double dpg;
8      cout << "Enter number of donuts:\n";
9      cin >> donuts;
10     cout << "Enter number of glasses of milk:\n";
11     cin >> milk;

12     if (milk <= 0)
13     {
14         cout << donuts << " donuts, and No Milk!\n"
15             << "Go buy some milk.\n";
16     }
17     else
18     {
19         dpg = donuts / static_cast<double>(milk);
20         cout << donuts << " donuts.\n"
21             << milk << " glasses of milk.\n"
22             << "You have " << dpg
23             << " donuts for each glass of milk.\n";
24     }

25     cout << "End of program.\n";
26     return 0;
27 }

```

### 示例运行结果

```

Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.

```

### 示例 18.2 使用异常处理同一特殊情况

```

1
2  #include <iostream>
3  using std::cin;
4  using std::cout;

5  int main( )
6  {
7      int donuts, milk;
8      double dpg;

9      try
10     {

```

这是个学习 C++ 语法的简单样例。不要把它当成是处理异常的一个很好的代表性例子。

```

11         cout << "Enter number of donuts:\n";
12         cin >> donuts;
13         cout << "Enter number of glasses of milk:\n";
14         cin >> milk;
15
16         if (milk <= 0)
17             throw donuts;
18
19         dpg = donuts / static_cast<double>(milk);
20         cout << donuts << " donuts.\n"
21             << milk << " glasses of milk.\n"
22             << "You have " << dpg
23             << " donuts for each glass of milk.\n";
24     }
25     catch (int e)
26     {
27         cout << e << " donuts, and No Milk!\n"
28             << "Go buy some milk.\n";
29     }
30
31     cout << "End of program.\n";
32     return 0;
33 }

```

### 示例运行结果1

```

Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
End of program.

```

### 示例运行结果2

```

Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.

```

在 C++ 中，处理异常的最基本的方法是由 try-throw-catch 三条语句构成的。

**try 代码块** try 代码块的语法如下：

```

try
{
    Some_Code
}

```

try 代码块包含的是基本算法的代码，它用于处理正常情况。之所以称为 try，是因为我们不能确保它 100% 顺利执行，但是想“尝试一下”。



如果确实会发生某种不正常情况，那么可以通过抛出异常的方式来表示。当我们增加一个 `throw` 后，代码的基本概况如下所示：

```
try
{
    Code_To_Try (需要尝试运行的代码)
    Possibly_Throw_An_Exception (可能抛出异常)
    More_Code (更多其他代码)
}
```

下面是一个带 `throw` 语句的 `try` 块例子（拷贝示例 18.2 的代码）：

```
try
{
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;

    if (milk <= 0)
        throw donuts;

    dpq = donuts / static_cast<double>(milk);
    cout << donuts << " donuts.\n"
        << milk << " glasses of milk.\n"
        << "You have " << dpq
        << " donuts for each glass of milk.\n";
}
```

上述代码中的下列语句**抛出** `int` 类型的值 `donuts`：

**throw 语句**  
**异常**  
**抛出异常**

```
throw donuts;
```

被抛出的值（在本例中是 `donuts`）有时被称为**异常**；而 `throw` 语句的执行被称为**抛出异常**。异常机制允许抛出任意类型的值。在本例中，抛出 `int` 类型的值。

**throw 语句**

**语法**

```
throw Expression_for_Value_to_Be_Thrown;
```

当执行 `throw` 语句时，`try` 代码块的执行将会终止。如果 `try` 代码块之后紧跟着一个正确的 `catch` 代码块，那么程序的控制流转入 `catch` 代码块的执行。`throw` 语句总是嵌入在一个分支判断语句中，比如 `if` 语句。被抛出的值可以是任意数据类型。

**示例**

```
if (milk <= 0)
    throw donuts;
```

当某种东西被“抛出”时，它从一个地方转移到了另外一个地方。在 C++ 中，从一个地方到另外一个地方的东西是控制流（还有被抛出的值）。当抛出异常时，`try` 代码块停止执行，另外一部分代码即 **catch 块**，开始执行。执行 `catch` 代码块被称为

**catch 块**

**处理异常**      **捕获异常或者处理异常。**当抛出异常后，异常最终会被某个 catch 块所处理。在示例 18.2 中，一个适当的 catch 块紧随 try 块之后。我们重复 catch 块的代码，如下所示：

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

catch 代码块看起来更像一个参数类型为 int 的函数定义。实际上它不是函数定义，但是在某些方面，catch 块看起来像是一个函数。catch 块是一个独立的代码片段，当程序遇到（执行）下列（在前面的 try 块中）代码时，它才会开始执行：

```
throw Some_int;
```

**异常处理器**      因此，throw 语句和函数调用比较相似，但是它不是去调用函数，而是调用 catch 块，并让程序执行 catch 块中的代码。catch 块通常被认为是一个**异常处理器**，异常处理器是一个特定术语，表明 catch 块有类似于函数一样的特性。

下列 catch 块代码行中的标识符 e 表示什么？

```
catch(int e)
```

**catch 块参数**      这个标识符 e 看起来像是个输入参数，而且它的使用方式也非常像参数的用法。实际上，像 e 这种在 catch 块开头部分的标识符被称为 **catch 块参数**。每个 catch 块最多只能有一个 catch 块参数。catch 块参数有两个作用：

1. catch 块参数之前有一个类型名，它指定了 catch 块能够捕获到什么类型的抛出值。
2. catch 块参数指定了所捕获的抛出值名字，这样我们可以在 catch 块中写代码来对这个值进行一些处理。

我们将以相反的顺序来讨论 catch 块参数的这两个功能。这一小节先讨论使用 catch 块参数作为被抛出和被捕获值的名字。这一小节的名字叫“多个抛出和捕获”，在本章的后面，会讨论哪个 catch 块（异常处理器）将处理所抛出的值。在我们当前的例子中只有一个 catch 块。通常情况下 catch 块参数的名字是 e，但也可以使用任何合法的标识符来替换 e。

我们来看看示例 18.2 的 catch 块是如何工作的。当程序抛出一个异常值时，try 块中的代码执行结束，控制流转到紧随 try 块之后的 catch 块（或多个块）中。我们把示例 18.2 的 catch 块拷贝到这里：

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

当程序抛出一个异常值时，被抛出值的数据类型必须是 int，这样这个 catch 块才能正常使用。在示例 18.2 中，被抛出的值是由变量 donuts 给出的；由于变量 donuts 是 int 类型的，因此，catch 块能捕获到所抛出的异常值。

正如示例 18.2 的第二个示例运行结果所示的那样，我们假设 donuts 的值是 12，milk 的值是 0。由于 milk 的值是非正数的，那么 if 语句中的 throw 语句将会被执行。在示例 18.2 中，变量 donuts 的值被抛出。当 catch 块捕获到 donuts 的值后，donuts 的值将被插入到 catch 块参数 e 中，然后 catch 块中的代码开始执行，继而产生如下输出：

```
12 donuts, and No Milk!
Go buy some milk.
```

如果 donuts 的值是正数，那么程序将不会执行 throw 语句。这种情况下，整个 try 块的代码会被执行。当 try 块的最后一行语句执行结束后，catch 代码块之后的代码将会被执行。注意，如果没有异常抛出，程序将会跳过 catch 块代码。

从上述讨论可以看出，似乎 try-throw-catch 的结构和 if-else 语句很相似。除了前者能抛出异常值外，两者几乎是等价的。try-throw-catch 的结构很像是赋予 if-else 语句给分支语句发送消息的额外能力。这样看起来的话，它和 if-else 语句本身区别不大，但实际使用表明它们之间还是有很大的不同。

为了对上述内容进行较为正式的总结，我们假设 try 块包含了某种代码，它内含 throw 语句。通常只有在异常发生时 throw 语句才能执行，当执行它时，它会抛出某种数据类型的值。当程序抛出异常时（如示例 18.2 抛出变量 donuts 的值），try 块执行结束。try 块中所有其他代码将被跳过，而且程序控制流将会转向对应的 catch 块。catch 块只能应用于和它紧挨的前面的一个 try 块所抛出的异常。如果抛出异常，那么异常对象将被插入到 catch 块参数里，接着 catch 块中的代码将会被执行。例如，如果你回顾示例 18.2 的运行结果，你会发现一旦用户输入一个非正数的数值，try 块就会终止执行，catch 块开始执行。到目前为止，我们假设每一个 try 块后跟一个正确的 catch 块。后续，我们将讨论如果 try 之后没有正确的 catch 块将会发生什么状况。

### catch 块参数

catch 块参数是出现在 catch 块开始部分的一个标识符，它充当的是一个异常抛出占位符。当在它前面的 try 块中抛出一个恰当的值时，这个值就会被插入到 catch 块参数中（抛出值的数据类型必须符合 catch 块参数类型要求，这样 catch 块的代码才能正确执行）。你可以使用任何合法（非 C++ 保留字）的标识符来作为 catch 块参数。

#### 示例

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

e 是 catch 块参数。

如果 try 块中没有抛出异常（没有值），那么 try 块的代码执行完成之后程序会从 catch 代码块之后的代码处继续执行。换句话说，如果程序没有抛出异常，程序将会跳过 catch 块。在大部分情况下，throw 语句是会被执行的，因此，大部分情况下 try 块中的代码都会被完全执行，而 catch 块中的代码都会被全部跳过。

#### try-throw-catch

抛出和捕获异常的基本机制是通过 try-throw-catch 序列来实现的。throw 语句抛出异常（一个值），catch 块捕获异常（这个值）。当异常被抛出时，try 块结束，catch 块中的代码开始执行。catch 块执行完成后，catch 块之后的代码会被接着执行（假设 catch 块中的代码并没有终止程序或执行其他特殊操作）。

（抛出异常的数据类型必须和 catch 块中的参数类型列表匹配，否则异常不会被 catch 块所捕获。关于这一点，我们会在后续的子章节“多个抛出和捕获”中进一步讨论。）

如果在 try 块之中没有抛出异常，那么 try 块执行完成后，程序会从 catch 块之后的代码处接着执行。（换句话说，如果没有抛出异常，catch 块将被跳过。）

#### 语法

```
try
{
    一些代码语句
    < 输入带 throw 语句的代码或者一个可能会抛出异常的函数调用。 >
    Some_More_Statements (其他一些代码语句)
}
catch (Type e)
{
    < 如果 try 块中抛出了 catch 参数类型值，那么此处的代码将会被执行。 >
}
```

#### 示例

例子参考示例 18.2。

## 自测练习題

### 1. 下列代码产生的输出结果是什么？

```
int waitTime = 46;

try
{
    cout << "Try block entered.\n";
    if (waitTime > 30)
        throw waitTime;
    cout << "Leaving try block.\n";
}

catch (int thrownValue)
{
}
```

```

        cout << "Exception thrown with\n"
              << "waitTime equal to " << thrownValue << endl;
    }
    cout << "After catch block" << endl;

```

2. 如果我们做了如下面代码所示的一些变化后, 请问自测练习题1的输出结果是什么? 所做的改动是把下面这行

```
int waitTime = 46;
```

改为:

```
int waitTime = 12;
```

3. 在自测练习题1所给出的代码中, throw语句是什么?
4. 当throw语句被执行时将会发生什么情况? (请进行一般性的回答, 而不是简单地针对自测练习题1或者其他样例代码进行回答。)
5. 在自测练习题1所给出的代码中, try块是什么?
6. 在自测练习题1所给出的代码中, catch块是什么?
7. 在自测练习题1所给出的代码中, catch块参数是什么?

## 定义自己的异常类

throw语句可以抛出任意类型的异常值。因此, 通常情况下, 我们可以定义一个类, 让该类的对象携带我们所需要的准确信息, 把这些信息抛给catch块。自定义异常类的一个更为重要的原因是我们能够通过不同的类型来识别每一种可能发生的异常情况。

异常类仅仅是一个类。使它成为异常类的原因在于它被应用于异常处理。尽管如此, 在选择异常类的名字和处理其他细节时还是需要注意。

示例18.3包含一个程序样例, 它带有一个编程者自定义的异常类。这只是个简单的示例程序, 用于展现C++的一些异常处理细节。虽然对这个简单的例子使用了过多的设计, 但是, 从另一方面来看, 它也是一个清晰地展现了C++异常类具体实现细节的例子。

注意下列代码中的throw语句:

```
throw NoMilk(donuts);
```

NoMilk(donuts)部分是对类NoMilk构造函数的调用。构造函数接受一个int参数(在本例中是donuts), 并创建一个类NoMilk对象。接着该对象被抛出。

## 多个抛出和捕获

一个try块有抛出任意多个异常值的可能性, 这些异常值的数据类型可能各不相同。但任意一个try块在执行时, 最多只能抛出一个异常(因为throw语句结束了try块代码的执行), 但是当try块代码被执行时, 在不同情况它会抛出不同的异常值。每个catch块只能捕获一种数据类型的异常值, 但通过在一个try块之后放置多个catch块, 我们可以捕获多个不同类型的异常值。例如, 示例18.4所示的程序, 它的try块之后有两个catch块。

### 示例 18.3 自定义异常类

```

1  #include <iostream>
2  using std::cin;
3  using std::cout;

4  class NoMilk
5  {
6  public:
7      NoMilk( ) {}
8      NoMilk(int howMany) : count(howMany) {}
9      int getCount( ) const{ return count; }
10 private:
11     int count;
12 };

13 int main( )
14 {
15     int donuts, milk;           样例程序的运行结果和示例 18.2 的运行
16     double dpq;               结果是一样的。
17     try
18     {
19         cout << "Enter number of donuts:\n";
20         cin >> donuts;
21         cout << "Enter number of glasses of milk:\n";
22         cin >> milk;

23         if (milk <= 0)
24             throw NoMilk(donuts);

25         dpq = donuts / static_cast<double> (milk);
26         cout << donuts << " donuts.\n"
27              << milk << " glasses of milk.\n"
28              << "You have " << dpq
29              << " donuts for each glass of milk.\n";
30     }
31     catch (NoMilk e)
32     {
33         cout << e.getCount( ) << " donuts, and No Milk!\n"
34              << "Go buy some milk.\n";
35     }
36     cout << "End of program.\n";
37     return 0;
38 }

```

### 示例 18.4 捕获多个异常

```

1  #include <iostream>
2  #include <string>
3  using std::cin;
4  using std::cout;
5  using std::endl;
6  using std::string;

```

异常类可能有它们自己的接口和实现文件，而且可以放在一个命名空间里。这是另一个简单的示例程序。

```

7  class NegativeNumber
8  {
9  public:
10     NegativeNumber( ){}
11     NegativeNumber(string theMessage): message(theMessage) {}
12     string getMessage( ) const{ return message; }
13 private:
14     string message;
15 };

16 class DivideByZero
17 {};

18 int main( )
19 {
20     int pencils, erasers;
21     double ppe; // 每块橡皮可以被多少支铅笔使用。

22     try
23     {
24         cout << "How many pencils do you have?\n";
25         cin >> pencils;
26         if (pencils < 0)
27             throw NegativeNumber("pencils");
28         cout << "How many erasers do you have?\n";
29         cin >> erasers;
30         if (erasers < 0)
31             throw NegativeNumber( "erasers" );

32         if (erasers != 0)
33             ppe = pencils / static_cast<double>( erasers );
34         else
35             throw DivideByZero( );
36         cout << "Each eraser must last through "
37              << ppe << " pencils.\n";
38     }
39     catch (NegativeNumber e)
40     {
41         cout << "Cannot have a negative number of "
42              << e.getMessage( ) << endl;
43     }
44     catch (DivideByZero)
45     {
46         cout << "Do not make any mistakes.\n";
47     }
48     cout << "End of program.\n";
49     return 0;
50 }

```

如果 catch 块参数没有被使用，那么就没有必要在文件开始部分给出该参数。

#### 示例运行结果 1

```

How many pencils do you have?
5
How many erasers do you have?
2

```

```
Each eraser must last through 2.5 pencils
End of program.
```

### 示例运行结果2

```
How many pencils do you have?
-2
Cannot have a negative number of pencils
End of program.
```

### 示例运行结果3

```
How many pencils do you have?
5
How many erasers do you have?
0
Do not make any mistakes.
End of program.
```

注意，在 `catch` 块中没有 `DivideByZero` 参数。如果程序不需要参数，可以不带参数，只是简单地列出参数类型名。关于这个问题，我们将会在后面的编程提示“异常类可以很简单”中做进一步的讨论。



### 陷阱：首先捕获比较明确的异常

当捕获多个异常时，`catch` 块的顺序是很重要的。当 `try` 块中抛出一个异常值时，程序流会转向紧随 `try` 块的多个 `catch` 块，按顺序依次试着调用，第一个匹配抛出异常类型的 `catch` 块将会被执行。

例如，下列代码是一种特殊的 `catch` 块，它可以捕获所抛出的任意类型的值：

`catch(...)`

```
catch(...)
{
    <在此处放置你想使用的代码。>
}
```

上述代码中的三个点并不表示省略掉一些代码。你可以在你的实际程序中输入三个点。这是一个放置在其他 `catch` 块之后的、很不错的默认 `catch` 块。例如，我们可以把它增加到示例 18.4 的 `catch` 块中。代码如下所示：

```
catch (NegativeNumber e)
{
    cout << "Cannot have a negative number of "
          << e.getMessage() << endl;
}
catch (DivideByZero)
{
    cout << "Do not make any mistakes.\n";
}
catch (...)
{
    cout << "Unexplained exception.\n";
}
```



但是需要注意，只有把默认 catch 块放到 catch 块列表中的最后才是合理的。例如，假设我们让它成为第二个 catch 块，如下所示：

```
catch(NegativeNumber e)
{
    cout << "Cannot have a negative number of "
          << e.getMessage() << endl;
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
catch(DivideByZero)
{
    cout << "Do not make any mistakes.\n";
}
```

类型为 NegativeNumber 的异常（被抛出的值）理所应当将被参数类型为 NegativeNumber 的 catch 块所捕获。但是，如果类型为 DivideByZero 的值被抛出，它将会被 catch(...) 所捕获。因此，DivideByZero 捕获块永远不会被程序调用到。幸运的是，大部分编译器遇到这种情况时都会告诉你，你的 catch 块顺序设置有问题。■

#### 提示：异常类可以很简单

在下列代码中，我们复制了示例 18.4 的异常类 DivideByZero 的定义：

```
class DivideByZero
{
};
```

这个异常类没有成员变量，也没有成员函数（除了默认构造函数）。它除了名字外，什么都没有，但这也足够了。正如示例 18.4 所示，抛出一个类 DivideByZero 的对象，可以触发适当的 catch 块的执行。

当使用一个简单的异常类时，一旦程序控制流到达 catch 块，通常我们就不能再对异常（抛出的值）做任何处理了。由于异常仅仅被用来使程序到达 catch 块，因此，我们可以忽略 catch 块参数。事实上，不管异常类型简单与否，如果我们不需要 catch 块参数，那么在任何时候我们都可以忽略它。■

#### 在函数中抛出异常

有时候对异常进行延后处理是合理的。例如，你有一个函数，函数中有一段代码，当它遇到试图除零的操作时，会抛出异常，但有可能你并不想在函数中捕获这个异常。一些程序在调用这个函数时可能会选择终止程序，而其他一些调用该函数的程序可能会做一些其他处理。因此，你可能也不清楚当捕获到异常后，在函数中到底做什么处理。针对这种情况，在函数定义中不要捕获异常，而是让调用这个函数的程序把对函数的调用放到一个 try 块中，在一个紧随该 try 块的 catch 块中捕获异常。

我们来看看示例 18.5 的程序。它有一个 try 块，但是在 try 块中并没有看到 throw 语句。在程序中，抛出异常的语句如下列代码所示：

```

        if (bottom == 0)
            throw DivideByZero( );

```

这条语句在 try 块中是看不到的。但是，它的执行与 try 块是有关系的，因为它在函数 safeDivide 的定义中，而在 try 块中对函数 safeDivide 进行了调用。

我们将在下一小节中讨论在函数 safeDivide 中声明 throw (DivideByZero) 的意义。

### 示例 18.5 在函数内部抛出异常

---

```

1  #include <iostream>
2  #include <cstdlib>
3  using std::cin;
4  using std::cout;
5  using std::endl;

6  class DivideByZero
7  {};

8  double safeDivide(int top, int bottom) throw (DivideByZero);

9  int main( )
10 {
11     int numerator;
12     int denominator;
13     double quotient;
14     cout << "Enter numerator:\n";
15     cin >> numerator;
16     cout << "Enter denominator:\n";
17     cin >> denominator;

18     try
19     {
20         quotient = safeDivide(numerator, denominator);
21     }
22     catch (DivideByZero)
23     {
24         cout << "Error: Division by zero!\n"
25              << "Program aborting.\n";
26         exit(0);
27     }

28     cout << numerator << "/" << denominator
29          << " = " << quotient << endl;

30     cout << "End of program.\n";
31     return 0;
32 }
33
34 double safeDivide(int top, int bottom) throw (DivideByZero)
35 {
36     if (bottom == 0)
37         throw DivideByZero( );

```

```

38         return top / static_cast<double>(bottom);
39     }

```

#### 示例运行结果1

```

Enter numerator:
5
Enter denominator:
10
5/10 = 0.5
End of Program.

```

#### 示例运行结果2

```

Enter numerator:
5
Enter denominator:
0
Error: Division by zero!
Program aborting.

```

## 异常说明

### 异常说明

如果一个函数不打算捕获异常，那么它至少应该警告程序员该函数的任何调用都有可能抛出异常。如果在函数定义中，有异常被抛出但却不被捕获，那么这些异常类型应该被列在异常说明中，就像示例 18.5 的函数声明所给出的那样：

```
double safeDivide(int top, int bottom) throw (DivideByZero);
```

### 抛出列表

正如示例 18.5 所示，异常说明应该既出现在函数声明中又出现在函数定义中。如果一个函数有多个函数声明，那么所有函数声明必须有相同的异常说明。函数的异常说明有时也被称为**抛出列表**（throw list）。

如果在函数定义中存在多个可能出现的异常，那么可以通过逗号把多个异常类型分开，如下列代码所示：

```
void someFunction() throw (DivideByZero, SomeOtherException);
```

在异常说明中的所有异常类型都会被正常处理。我们所说的异常被正常处理是指，它可以按我们在前面章节所描述的处理方式那样进行处理。特别是，我们可以把函数调用放到一个 try 块中，try 块之后紧随一个 catch 块来捕获那种类型的异常，如果函数抛出了异常（而且没有在函数内部捕获它），那么紧随 try 块的 catch 块将会捕获到这个异常。

如果没有异常说明（没有抛出列表），甚至连一个空类型异常都没有，那么程序运行看起来好像把所有可能的异常类型都放在了异常说明中；也就是说，任何抛出的异常都被正确处理。

如果一个函数抛出了异常，但是该异常类型并没有出现在异常列表中（而且异常未在函数内部被捕获），这将会出现什么情况呢？这既不是一个编译时错误，也不是一个运行时错误。在这种情况下，函数 unexpected() 会被调用。你可以改变函数

unexpected 的处理方式，但是该函数的默认处理方式是调用 `terminate()` 函数，这个函数会结束程序。特别需要注意，如果在函数中抛出了一个异常，但该异常类型并没有出现在异常说明中（也没有在函数内部捕获），那么它将被程序中的任何 `catch` 块所捕获。作为另一种处理方式，它会导致程序调用 `unexpected()` 函数，这个函数的默认行为是结束程序。

需要记住一点，异常说明主要是针对那些会“跳出”函数的异常类型。如果它们不是在函数外部被处理，那么就不属于异常说明中的异常类型。如果它们在函数外部处理，不管它们在什么地方出现，它们都应当在异常说明中被列出。如果在一个 `try` 块中抛出了一个异常，这个异常在函数定义内部，并且在函数内部的 `catch` 块中被捕获，那么它的类型不需要列在异常说明中。如果一个函数定义中包含了对另一个函数的调用，而且另一个函数可以抛出不被捕获的异常，那么异常类型应当放在异常说明中。

或许你会认为，抛出不被捕获的异常以及不在抛出列表中出现异常，可能应该交给编译器去检查。但是，考虑到 C++ 中异常执行的具体细节，对编译器来说执行这种检查是不可能的。因此，必须在运行时进行检查<sup>1</sup>。

如果你希望一个函数不要抛出任何在函数内部捕获不到的异常，那么你可以使用空异常说明，如下列代码所示：

```
void someFunction() throw ( );
```

我们对各种异常说明进行归纳，如下所示：

```
void someFunction() throw (DivideByZero, SomeOtherException);  
// 类型为 DivideByZero 或者 SomeOtherException 的异常被正常处理。  
// 所有其他异常的处理使用函数 unexpected()。
```

```
void someFunction() throw ( );  
// 空异常列表，所有异常处理都会调用函数 unexpected()。
```

```
void someFunction() ;  
// 所有类型的异常都能正常处理。
```

函数 `unexpected()` 的默认行为是结束程序。不需要使用任何特别的 `include` 或者 `using` 指令即得到函数 `unexpected()` 的默认行为。通常，我们不需要对函数 `unexpected()` 的默认行为进行重新定义，但是，可以使用函数 `set_unexpected` 来改变函数 `unexpected()` 的行为。如果需要使用函数 `set_unexpected`，那请参考一本更高级的书来查询相关具体细节。

我们要记住一个派生类对象同时也是其基类的对象。因此，如果 `D` 是类 `B` 的派生类，`B` 在异常说明中，那么函数抛出的类 `D` 对象也可以被正常处理，因为它是类 `B` 的对象。但是，在这个过程中，异常机制并不会对类型进行自动转换。如果 `double` 类型在异常说明中，它并不能用于处理 `int` 类型的异常。还是需要把 `int` 和 `double` 都放到异常说明中。

1 对于所有编程语言来说，这并不是一定的。对于某种特定的编程语言来说，这主要取决于在该语言中，异常说明的具体细节是如何定义的。

**警告!**

最后一条警告：不是所有编译器都会按我们所期望的方式来处理异常说明。一些编译器把异常说明当成注释处理。对于这些编译器来说，异常说明不会对程序代码产生任何影响。这也是为什么要把函数所有可能抛出的异常都放到异常说明中的另一个原因。采用这种方式，所有编译器会以相同的方式来处理异常。当然，也可以让函数不带异常说明，这样会使所有编译器的处理都一样，但是这样做的话对程序的文档化不好，而且程序也不会像使用了异常说明的程序那样得到编译器所提供的错误检查。如果编译器支持异常说明，那么程序一旦抛出我们不期望的异常时，它就会立即结束（注意这是一个运行时行为——但是这个运行时行为取决于你所使用的编译器）。

**陷阱：派生类中的异常说明**

当在派生类中重定义或覆盖一个函数定义时，它的异常说明应该和基类的异常说明相同，或者它的异常说明是基类异常说明的子集。换句话说，当重定义或者覆盖一个函数定义时，不能在异常说明中增加任何异常（但可以删除一些不需要的异常）。这是合理的，因为派生类对象可以用在任何基类对象可以使用的地方，因此，函数重定义或者重写必须满足任何为基类对象所编写的代码。■

**自测练习题**

8. 下列程序所产生的输出结果是什么？

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

void sampleFunction(double test) throw (int);

int main( )
{
    try
    {
        cout << "Trying.\n";
        sampleFunction(98.6);
        cout << "Trying after call.\n";
    }
    catch(int)
    {
        cout << "Catching.\n";
    }

    cout << "End program.\n";
    return 0;
}

void sampleFunction(double test) throw (int)
{
    cout << "Starting sampleFunction.\n";
    if (test < 100)
        throw 42;
}
```

9. 当我们进行如下改动时，自测练习题 8 所产生的输出结果是什么？改动如下，在 try 块中把

```
sampleFunction(98.6);
```

改为：

```
sampleFunction(212);
```

## 18.2 异常处理的编程技术

仅在异常情况下使用这个。

沃伦·皮斯，《中尉的工具》

到目前为止，我们已经给出了许多代码来说明 C++ 中的异常处理是如何工作的，但是我们还没有给出一个设计良好的具有实际使用价值的异常处理程序实例。既然大家已经了解了异常处理的工作机制，本节将继续介绍使用异常处理的编程技术。

### 抛出异常的时机

我们已经给出了一些非常简单的代码来解释异常处理的基本概念，但我们之前给出的例子都过于简单、不太切合实际。一个更为复杂但却更具指导意义的方式是把抛出异常和捕获异常隔离到不同的函数中。在大多数情况下，应该把 throw 语句放到函数定义中，在函数的异常说明中列出异常，并且把每个 catch 语句放到不同的函数中。因此，对于 try-throw-catch 三段体的首选用法可以用如下代码来表示：

```
void functionA( ) throw (MyException)
{
    .
    .
    .
    throw MyException( <可能是个参数> );
    .
    .
    .
}
```

然后，在某个其他函数（甚至可能是某个其他文件中的某个其他函数）中，可能会有如下代码：

```
void functionB( )
{
    .
    .
    .
    try
    {
        .
        .
        .
    }
```

```

        functionA( );
        .
        .
    }
    catch (MyException e)
    {
        <处理异常>
    }
    .
    .
}

```

即使是上面这种 throw 语句的用法，也只应该在不可避免的情况下谨慎地使用。如果能以某种其他方式较为容易地处理一个问题，那就不要抛出异常。当异常条件的处理依赖于函数如何被使用以及在哪里被使用时，我们就应该谨慎使用 throw 语句。如果异常条件被处理的方式依赖于函数如何被使用以及在哪里被使用，那么最好的办法是让调用函数的程序员来处理异常。对于其他所有情况，最好避免抛出异常。为了说明这一问题，我们来设计一个关于这种问题的使用场景。

假设你正在为医院中的病人监控系统编写一个函数库。库中有一个函数可能用于计算病人每日的体温，它的值是通过访问某个文件中病人的记录并使用多次测量的体温值之和除以测量次数而得到的。现在，假设这些函数被用来创建服务于不同情况的不同系统。如果从未对病人的体温进行过测量，那么求平均值会导致除零运算，这将会发生什么情况？对于一个重症监护室来说，这表示有地方出错了，比如病人丢了（众所周知这是会发生的）。因此，对于这个系统来说，当这个潜在的除零运算发生时，系统应当发出紧急信号。但是，当系统用在某个不是很紧急的场合下，比如门诊室或者甚至是一些非重危病房，可能就没什么意义了，在病人记录上写一个简单的通知就够了。在这种场景下，求病人体温均值的函数应当在发生除零运算时抛出一个异常，在异常说明中列出这个异常，并让每一个系统以自己的方式去处理这个异常。

### 抛出异常的时机

在大多数情况下，应该在函数内部使用 throw 语句，并且应该在函数中列出异常说明。此外，当异常条件的处理依赖于函数如何被使用以及在哪里被使用时，应该谨慎使用 throw 语句。如果异常条件被处理的方式依赖于函数如何被使用以及在哪里被使用，那么最好的办法是让调用函数的程序员来处理异常。对于其他情况，最好避免抛出异常。



### 陷阱：未被捕获的异常

代码所抛出的每个异常都应在程序的某个地方被捕获。如果程序抛出了一个异常，但在程序的任何地方都没去捕获它，那么程序将终止。

从技术上讲,如果程序抛出了一个异常,但异常没被捕获,那么,函数 `terminate()` 会被调用。函数 `terminate()` 的默认行为是终止程序。我们可以改变函数 `terminate()` 的默认行为,但一般不需要这么做,在这里就不再详细展开了。

一个函数抛出了异常,但在函数内部以及外部都没捕获该异常,那么可能会出现两种情况。如果在异常说明中没有列出该异常,那么程序会调用函数 `unexpected()`。如果在异常说明中列出了该异常,那么程序会调用函数 `terminate()`。除非对函数 `unexpected()` 做了改动,否则它还是会去调用函数 `terminate()`。因此,对于这两种情况而言,结果是一样的。如果一个函数抛出了异常,但在函数内部以及外部都没捕获该异常,那么程序将会终止。■



### 陷阱：嵌套 try-catch 块

可以把一个 try 块及紧随其后的 catch 块放到一个大的 try 块中,或者放到一个大的 catch 块中。在很少的情况下,这么做可能是有用的。但是,如果你打算使用这种方式,那么你应该先考虑一下是否有更好的方式来组织你的程序。一般情况下,把内部的 try-catch 块放到函数定义中,把对函数的调用放在外层的 try 块或 catch 块中(或许仅仅是为了完全消除一个或更多的 try 块),这样编写程序总是会更好一些。

如果把一个 try 块及紧随其后的 catch 块放在一个更大的 try 块中,异常在内层 try 块中抛出,但却没有在内层的任何 catch 块中捕获,那么异常会被抛到外层的 try 块中,并且可能会被紧随外层 try 块的 catch 块所捕获。■



### 陷阱：过度使用异常

通过使用异常,我们可以写出控制流异常复杂的程序,这种复杂性使得对程序的理解会变得相当困难。而且,使用异常很容易写出复杂的程序。抛出异常可以使得程序的控制流从程序的一个地方转到另外一个地方。在早期的编程工作中,使用 `goto` 语句可以实现这种无限制的控制流。现在,编程专家们都认为这种无限制控制流的编程方式是一种非常不好的程序风格。异常又使得我们能回归到这种老式的、糟糕的无限制控制流的编程时代。因此,应当尽可能少地使用异常,应该以某种比较好的方式来使用异常。有一种不错的规则可以判断是否要使用异常,具体如下:如果你想把一个 `throw` 语句放到程序中,那请先考虑一下如果没有 `throw` 语句你会如何写程序或者定义类。如果你能找到一个替代的、合理的代码编写方案,那可能你就不再想把 `throw` 语句引入到你的程序了。■

## 异常类层次结构

定义异常类的层次结构是非常有用的。例如,可能我们有一个 `ArithmeticError` 异常类,接着我们定义了一个 `DivideByZeroError` 异常类,它是异常



类 `ArithmeticError` 的派生类。由于每个 `DivideByZeroError` 对象也是一个 `ArithmeticError` 对象，因此，对象 `ArithmeticError` 的每个 `try` 块能捕获到对象 `DivideByZeroError` 所抛出的异常。如果在异常说明中列出了异常类型 `ArithmeticError`，那么，不管你是否通过手动方式把 `DivideByZeroError` 添加到了异常说明中，异常类型 `DivideByZeroError` 实际上已经被自动添加到了异常说明中。

## 可用内存测试

在第 17 章中，我们用下面类似的代码创建了新的动态变量：

```
struct Node
{
    int data;
    Node *link;
};
typedef Node* NodePtr;
...
NodePtr pointer = new Node;
```

只要内存充足就可以创建新节点，那么，上述这段代码就可以正常工作。但是，如果内存不足的话，会发生什么情况呢？事实上，如果没有充足的内存可以用来创建新节点，那么程序会抛出 **bad\_alloc** 异常。

考虑到当内存不足时，使用 `new` 操作创建新节点会抛出 `bad_alloc` 异常，我们可以使用如下代码来检测内存是否已经耗尽：

```
try
{
    NodePtr pointer = new Node;
}
catch (bad_alloc)
{
    cout << "Ran out of memory!";
}
```

当然，在上述代码中，除了简单地给出一个警告消息外，也可以做一些其他处理。到底做哪些细节处理，主要依赖于程序任务本身。

`bad_alloc` 的定义在函数库的头文件 `<new>` 中给出，它被放在 `std` 命名空间中。因此，当使用 `bad_alloc` 时，程序应当包含下列代码（或其他类似代码）：

```
#include <new>
using std::bad_alloc;
```

## 再次抛出异常

在 C++ 语言中，`catch` 块中再次抛出异常是合法的。可能只有在很少的情况下，我们需要在捕获异常之后，再根据具体情况，以决定再次抛出相同的或不同的异常来给其他异常处理模块做进一步处理。

## 自测练习题

10. 当异常没有被任何 catch 块捕获时，会发生什么情况？
11. 可以把一个 try 块内嵌到另一个 try 块中吗？

## 本章小结

- 异常处理机制可以使你在对程序进行设计和编码时，将对正常情况的处理和对意外情况的处理隔离开。
- 可以在 try 块中抛出异常。作为选择，也可以在一个没有 try 块的函数中抛出异常（或者在一个没有能够捕获这个异常类型的 catch 块中抛出异常）。在这种情况下，可以把函数调用放到一个 try 块中。
- 异常是在 catch 块中被捕获的。
- try 块之后可能跟随多个 catch 块。在这种情况下，总是把捕获特定异常类的 catch 块放在捕获一般异常类的前面。
- 异常处理的方式会随调用函数的变化而变化，但无论如何，在一个函数中抛出异常（但不在函数中捕获异常）是使用异常的最好方式。几乎没有什么应用场景能够从抛出异常中获益。
- 如果在函数中抛出了异常，但没有在函数中捕获它，那么应该在函数的异常说明中列出该异常的类型。
- 如果被抛出的异常没有被任何 catch 块捕获，那么默认情况是程序终止运行。
- 不要过度使用异常。

## 自测练习题答案

1. Try block entered.  
Exception thrown with  
waitTime equal to 46  
After catch block.
2. Try block entered.  
Leaving try block.  
After catch block.
3. throw waitTime;

注意，下列代码是一个 if 语句，而不是 throw 语句，尽管它里面包含了一句 throw 语句：

```
if (waitTime > 30)
    throw waitTime;
```

4. 当执行 `throw` 语句时, `try` 块代码执行结束。`try` 块中的其他代码将被跳过不再被执行, 程序控制流转向随后的一个或多个 `catch` 块。当我们说控制流转到了 `catch` 块, 我们的意思是说抛出的异常值赋给了 `catch` 块参数, 并且程序开始执行 `catch` 块的代码了。

```
5. try
{
    cout << "Try block entered.";
    if (waitTime > 30)
        throw waitTime;
    cout << "Leaving try block.";
}
```

```
6. catch(int thrownValue)
{
    cout << "Exception thrown with\n"
        << "waitTime equal to" << thrownValue << endl;
}
```

7. `thrownValue` 是一个 `catch` 块参数。

```
8. Trying.
   Starting sampleFunction.
   Catching.
   End of program.
```

```
9. Trying.
   Starting sampleFunction.
   Trying after call.
   End of program.
```

10. 如果所有 `catch` 块都没有捕获异常, 那么你的程序将会终止运行。从技术上讲, 如果程序抛出了一个异常, 但异常没被捕获, 那么, 函数 `terminate()` 会被调用, 而函数 `terminate()` 的默认行为是终止程序的运行。
11. 是的, 你可以把一个 `try` 块和它对应的一个 `catch` 块放到一个大的 `try` 块中。但是, 如果把内层的 `try` 和 `catch` 块放到一个函数定义中, 把对它的调用放到一个大的 `try` 块中可能会更好。

## 编程练习

1. 从第 10 章的示例 10.11 获取类 `PFArrary` 的源代码。修改重载运算符 `[]` 的定义, 使得它抛出 `OutOfRangeException` 异常, 能够处理数组索引越界问题, 以及处理当数组已满时试图增加元素的问题。`OutOfRangeException` 类是一个自定义的异常类。异常类应该有一个私有的 `int` 成员和 `string` 成员, 以及一个公共的带 `int` 和 `string` 类型参数的构造函数。异常对象应该存储错误的索引值和相关的消息。你可以选择消息来描述发生的异常情况。编写一个测试程序来测试修改后的类 `PFArrary`。

2. (对于 Stroustrup 问题, 见 *C++ Programming Language*, 第 3 版) 编写一个程序, 要求程序中包含一个函数调用另一个函数的代码, 这样的调用关系有 10 层深度。给每个函数一个参数来表示所抛出异常的层次。main 函数提示输入调用深度参数, 并接收所输入的参数, main 函数接下来会调用第一个函数。main 函数捕获异常并显示所抛出的是第几层的异常。别忘了抛出深度为 0 的情况, 在这种情况下, main 必须抛出并捕获异常。

提示: 你可以使用 10 个不同的函数或者同一个函数的 10 份拷贝来彼此调用, 但是不建议这么做。为了减少代码量, 可以使用 main 函数调用另外一个函数, 该函数对它自身进行递归调用。假如你用这种方式做了, 再对调用深度进行限制还有必要吗? 如果你这么做的话, 那就可以不用给函数附加任何参数, 但是如果你不会使用递归的方式, 那还是给函数增加一个多余的参数吧。

3. 修改第 17 章中从示例 17.17 到示例 17.19 的源代码。原有的程序是, 如果类的使用者试图从一个空栈中做弹出数据的操作, 程序会打印出错误消息并退出。修改程序, 使程序可以抛出名为 `PopEmptyStackException` 的异常。

写一个 main 函数来测试新编写的 `Stack` 类, 当它试图从一个空栈中执行弹出操作时, 就捕获异常类型 `PopEmptyStackException`。

4. 下列代码使用两个数组, 一个存储产品, 另一个存储产品 ID 号 (更好的组织方式是使用一个类数组或者结构体数组, 但这不是本道编程练习题所关注的主題)。函数 `getProductID` 接收的输入参数包括两个数组、两个数组的长度以及要搜索的目标产品。接着, 它开始从产品名数组开始循环, 如果找到了目标产品, 它返回对应的目标产品的 ID:

```
int getProductID(int ids[], string names[],
                 int numProducts, string target)
{
    for (int i=0; i < numProducts; i++)
    {
        if (names[i] == target)
            return ids[i];
    }
    return -1; // 没找到目标产品。
}
```

int main() // 测试 getProductID 函数的运行程序样例。

```
{
    int productIds[] = {4, 5, 8, 10, 13};
    string products[] = {"computer", "flash drive",
                        "mouse", "printer", "camera"};

    cout << getProductID(productIds, products, 5, "mouse") << endl;
    cout << getProductID(productIds, products, 5, "camera")
         << endl;
    cout << getProductID(productIds, products, 5, "laptop")
         << endl;
}
```

```

    return 0;
}

```

在实现函数 `getProductID` 时可能会有一个问题：如果没找到目标产品，函数会返回一个特殊的错误码 `-1`。函数调用者可能会忽略 `-1`，或者实际上我们可能想把 `-1` 作为一个产品 ID 号。请重新编写程序，使得当程序未找到目标产品时，不返回错误码 `-1`，而是抛出一个异常。

5. 对一个函数来说，用返回异常来代替返回某个特殊错误码通常是一种更好的方案。下列类用来维护一个账户结余。

```

class Account
{
private:
    double balance;
public:
    Account()
    {
        balance = 0;
    }
    Account(double initialDeposit)
    {
        balance = initialDeposit;
    }
    double getBalance()
    {
        return balance;
    }
    // 返回新的结余，如果出现错误则返回 -1。
    double deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
        else
            return -1; // 该代码表明有错误。
        return balance;
    }
    // 返回新的结余，如果是无效账户则返回 -1。
    double withdraw(double amount)
    {
        if ((amount > balance) || (amount < 0))
            return -1;
        else
            balance -= amount;
        return balance;
    }
};

```

重写类使得它抛出异常而不是返回错误码 `-1`。编写测试代码来对试图提取和存储无效数额的情况进行测试并捕获所抛出的异常。

## 19.1 迭代器 723

迭代器基础 723

陷阱：编译器问题 726

迭代器的种类 728

常量迭代器和可迭代器 731

反向迭代器 732

其他种类的迭代器 733

## 19.2 容器 734

连续容器 734

陷阱：迭代器和删除元素 738

提示：容器中的类型定义 739

容器适配器栈和队列 739

陷阱：底层容器 740

关联式容器set和map 742

效率 747

## 19.3 泛型算法 748

运行时间和大- $O$ 表示法 749

容器访问运行时间 752

不改变序列的算法 753

改变序列的算法 757

集合算法 758

排序算法 760

# 第 19 章 标准模板库

图书馆不是建起来的；它是自然增长起来的。

奥古斯丁·比勒尔

## 概述

在第 17 章中，我们创建了自己定义的数据结构：栈和队列。事实上，存在大量的标准数据结构集可以用来存储数据。由于这些数据结构集非常规范，因此，对这些数据结构集进行标准化的、可移植的定义是有意义的。标准模板库 (STL) 包含了很多关于这种数据结构的定义。它包含栈、队列和其他标准数据结构的实现。当我们在 STL 中讨论这些数据结构时，它们通常被称为容器类，这是因为它们是用来保存数据集的。在第 7 章中，通过对向量类的描述，使我们对 STL 有了初步的认识。向量类是 STL 中的一种容器类。本章将对 STL 中的一些基本类进行简要介绍。由于 STL 非常大，我们无法在这里给出 STL 所有内容的介绍，但是我们将尽力让大家开始学会使用一些基本的 STL 容器类以及 STL 中的一些其他内容。

STL 是由 Hewlett-Packard 公司的 Alexander Stepanov 和 Meng Lee 开发的，它是基于 Stepanov、Lee 和 David Musser 的研究成果。它是用 C++ 语言实现的一套库集合。尽管 STL 不是 C++ 的核心部分，但它是 C++ 标准的一部分，因此，任何遵守这套标准的 C++ 实现都包含 STL。在实际应用中，我们可以把 STL 看成是 C++ 语言的一部分。

正如其名所示，STL 中的类是模板类。在 STL 中，一个典型的容器类都有类型参数来表示存储在容器类的数据的类型。

STL 容器类广泛地使用了迭代器，它是用于遍历容器中所有数据的对象。第 17 章的 17.3 节对迭代器的基本概念进行了介绍。尽管本章假定你并没有学习过第 17 章的内容，但大多数读者还是会发现在学习本章之前学习第 17 章的内容是很有帮助的。正如 STL 所定义的那样，迭代器应用非常广泛，它不仅仅可以用来对我们将提到的几个容器内的数据进行遍历。在本章中，我们对迭代器的讨论主要集中在如何利用迭代器简化本章所提及的几个容器类的使用。在一个具体环境中去理解迭代器概念，会使得迭代器的概念更为生动，可以给读者更充分的理解，使得读者在阅读更深入的 STL 知识时更容易理解和接受（有很多专门讨论 STL 的书籍）。

在 STL 中，还包括很多重要的通用的算法实现，比如搜索和排序。算法是作为模板函数实现的。讨论完容器类后，我们将会讨论这些算法的实现。

STL 和 C++ 的一些其他库（比如 `<iostream>`）有所不同，STL 中的类和算法都是泛型的，换句话说，它们都是模板类和模板函数。

如果你还没有使用过 STL，那你应该先读一下第 7 章的 7.3 节，它涵盖了 STL 中向量 (vector) 模板类的知识。第 17 章给出了一些 STL 内在的抽象概念的具体实现样例，因此，尽管本章没有使用任何第 17 章的知识，但大多数读者将会发现在学习本章前学习第 17 章的知识有助于对本章的理解。

## 19.1 迭代器

人会去进行迭代，程序员是人。

佚名

如果你还没有阅读第 10 章关于指针和数组的内容以及第 7 章的 7.3 节关于向量的内容，那请先阅读这两章的相关内容。向量是 STL 中的容器模板类之一。迭代器是一个广义的指针。本节将向读者展示如何在向量中使用迭代器。19.2 节所介绍的其他容器模板类使用迭代器的方式和在向量中使用的方式是完全相同的。因此，在本节所学到的迭代器的用法将不仅仅适用于向量，它也广泛地适用于其他容器类。这反映了 STL 哲学中的一条原则：迭代器用法中的语义、命名和语法对于不同的容器类型来说都应当是一致的。

### 迭代器基础

#### 迭代器

迭代器是一个广义的指针，事实上，其代表性的实现方式就是通过使用指针实现的。但是，如果设计一个抽象的迭代器，就可以使我们从迭代器具体的实现细节中解脱出来，这个抽象的迭代器可以给我们提供一个统一的接口，它可以用在不同的容器类中。每一个容器类都有自己的迭代器类型，就像每一个数据类型都有它自己的指针类型一样。但是，正如所有类型的指针对于该类型的动态变量的行为在本质上都是一样的，每个类型的迭代器的行为也是一样的，但每个迭代器只能用在它自己的容器类型中。

迭代器不是指针，但如果认为它是个指针，并把它当作指针使用，也不算什么大错。像指针变量一样，迭代器变量定位于（或者说它指向）容器中的一条数据项。可以通过下列应用于迭代器对象的重载运算符来操作迭代器：

- 前置和后置自增运算符（++）使得迭代器指向后一个数据项。
- 前置和后置自减运算符（--）使得迭代器指向前一个数据项。
- 等于和不等于运算符（== 和 !=）用来测试两个迭代器是否指向相同位置的数据项。
- 解引用运算符用来表示：如果 *p* 是一个迭代器变量，那么，\**p* 用于对指针 *p* 所指的数据进行访问。访问可能是只读或只写，或者也有可能是既允许读数据也允许对数据进行更改，这主要依赖于具体的容器类。

并不是所有迭代器都有上述运算符。但作为一个容器例，向量模板类具有上述所有运算符，它还包括其他运算符。

容器类中包含一个能够启动迭代器进程的成员函数。毕竟，一个新的迭代器变量在刚开始时并不会处于（指向）容器中任何数据所在的位置。许多容器类，包括向量模板类，都有下列成员函数，它返回一个指向数据结构中某个特定元素的迭代器对象（迭代器值）：

- `c.begin()` 返回容器 *c* 的迭代器对象，该迭代器指向容器 *c* 的“第一个”元素。
- `c.end()` 返回容器 *c* 的迭代器对象，它被用于测试是否迭代器已经越过了容



器 `c` 的最后一个元素。我们完全可以把迭代器 `c.end()` 类比为 `NULL`，在第 17 章中，`NULL` 被用来测试是否一个指针已经越过了链表的最后一个元素。因此，迭代器 `c.end()` 所处的位置并不在容器的数据元素所在位置中，实际上，它是一种结束标识符。

对于许多容器类来说，通过使用这些工具，我们可以写出一个 `for` 循环来遍历容器对象 `c` 中的所有元素，如下所示：

```
//p 是用于容器变量 c 的迭代器变量。
for (p = c.begin(); p != c.end(); p++)
    process *p // *p 表示当前的数据元素。
```

上面是关于迭代器的基本描述。现在，我们来详细看看向量模板容器类的具体设置。

示例 19.1 以图形的方式表示了向量模板类中使用迭代器的情况。记住，尽管 STL 中的每一个容器类型使用迭代器的方式基本相同，但每一个容器类型都有自己的迭代器类型。用于 `int` 类型向量的迭代器类型表示如下：

```
std::vector<int>::iterator
```

另外一个容器类是链表模板类。`int` 类型链表的迭代器类型表示如下：

```
std::list<int>::iterator
```

在示例 19.1 的程序中，我们对类型名 `iterator` 做了具体的指定，使它可以应用于 `int` 型的向量，把它作为 `int` 型向量的迭代器。我们在向量模板类中定义了示例 19.1 要用到的迭代器类型名。因此，如果把向量模板类的类型指定为 `int` 类型，当我们需要一个指向 `vector<int>` 的迭代器类型时，我们可以使用如下代码来表示：

```
vector<int>::iterator;
```

### 示例 19.1 用于向量的迭代器

---

```
1 // STL 迭代器的演示程序。
2 #include <iostream>
3 #include <vector>
4 using std::cout;
5 using std::endl;
6 using std::vector;

7 int main()
8 {
9     vector<int> container;

10    for (int i = 1; i <= 4; i++)
11        container.push_back(i);

12    cout << "Here is what is in the container:\n";
13    vector<int>::iterator p;
14    for (p = container.begin(); p != container.end(); p++)
15        cout << *p << " ";
16    cout << endl;
```

```

17     cout << "Setting entries to 0:\n";
18     for (p = container.begin(); p != container.end(); p++)
19         *p = 0;

20     cout << "Container now contains:\n";
21     for (p = container.begin(); p != container.end(); p++)
22         cout << *p << " ";
23     cout << endl;

24     return 0;
25 }

```

### 示例运行结果

```
Here is what is in the container:
```

```
1 2 3 4
```

```
Setting entries to 0:
```

```
Container now contains:
```

```
0 0 0 0
```

示例 19.1 给出了向量（或其他任何容器类）迭代器的基本用法：

```

vector<int>::iterator p;
for (p = container.begin(); p != container.end(); p++)
    cout << *p << " ";

```

我们回想一下在程序中，`container` 的类型是 `vector<int>`，它的迭代器类型实际上可以用代码表示为：`std::vector<int>::iterator`。

我们可以把向量 `v` 看成是一个对其内在元素进行线性存储的类。它的第一个元素是 `v[0]`，第二个是 `v[1]`，依此类推。迭代器 `p` 是一个定位于容器中某个元素（或者指向某个元素）的对象。迭代器可以从容器中某个元素的位置移到另外一个元素的位置。比如说，如果对象 `p` 定位于 `v[7]`，那么 `p++` 会把 `p` 移到 `v[8]`。这样可以使得迭代器从向量中的第一个元素移动到最后一个元素，但它需要首先找到第一个元素，而且需要知道什么时候它会移到最后一个元素。

通过使用运算符“==”，可以辨别出一个迭代器和另一个迭代器是否处于容器中的同一位置。因此，当一个迭代器指向容器中的第一个、最后一个或者其他位置的元素时，我们可以测试另外一个元素是否处于第一个、最后一个或者其他位置。

如果 `p1` 和 `p2` 是两个迭代器，那么当且仅当 `p1` 和 `p2` 处于容器中的同一位置时，`p1==p2` 为真（这可以和指针进行类比。如果 `p1` 和 `p2` 都是指针对象，那么当 `p1` 和 `p2` 指向同一对象时，`p1` 和 `p2` 的比较结果为真）。通常情况下，`!=` 正好是 `==` 的逆操作，那么当 `p1` 和 `p2` 不处于容器中的同一位置时，`p1!=p2` 为真。

`begin()`

成员函数 `begin()` 用于把迭代器定位于容器中的首个元素位置处。对于向量容器以及许多其他容器类，成员函数 `begin()` 返回一个指向容器中第一个元素的迭代器对象（对于向量 `v`，第一个元素是 `v[0]`）。因此，

```
vector<int>::iterator p = v.begin();
```

表示初始化变量 `p`，使它指向容器中的第一个元素。继而，遍历向量 `v` 中所有元素的

循环语句的代码如下：

```
vector<int>::iterator p;
for (p = v.begin( ); Boolean_Expression ; p++)
    Action_At_Location p;
```

所期望的循环终止条件为：

```
p = v.end( )
```

**end( )** 成员函数 **end( )** 返回一个标记值，它用来检查是否迭代器已经越过了最后一个元素的位置。如果 **p** 定位于最后一个元素的位置，那么 **p++** 之后的 **p=v.end( )** 测试值将由 **false** 变为 **true**。因此，正确的布尔表达式和 **for** 循环语句的终止条件正好相反：

```
vector <int>::iterator p;
for (p = v.begin( ); p != v.end( ); p++)
    Action_At_Location p;
```

注意，除非对象 **p** 所处的位置已经超过了最后一个元素，否则表达式 **p != v.end( )** 不会从 **true** 变为 **false**。因此，**v.end( )** 并非定位于容器中的一个元素。**v.end( )** 的值是一个特殊值，用来表示一个结束标志。它不是迭代器，但是我们可以使用 **==** 和 **!=** 来对 **v.end( )** 和迭代器进行比较。**v.end( )** 的值可以类比为值 **NULL**。正如我们在第 17 章中所描述的那样，**NULL** 用来表示一个链表的结束。

下列代码摘录于示例 19.1 的 **for** 循环，它对名为 **container** 的向量，使用了相同的处理方式：

```
vector<int>::iterator p;
for (p = container.begin( ); p != container.end( ); p++)
    cout << *p << " ";
```

在迭代器 **p** 所处的位置采取如下动作：

```
cout << *p << " ";
```

解引用运算符 **\*** 已经被重载，用来对 STL 容器的迭代器进行操作，可以使用 **\*p** 得到位置 **p** 的数据元素。特别是，对于一个向量容器，使用 **\*p** 会得到迭代器 **p** 所指位置的元素。接着，前面的 **cout** 语句输出迭代器 **p** 所指向的元素。因此，整个 **for** 循环会输出向量容器中的所有元素。

使用解引用运算符 **\*p** 总会得到迭代器 **p** 所指向的元素。在某些情况下，**\*p** 只有读访问权限，不允许改变元素。而在其他一些情况下，它允许访问元素，也允许对元素做更改。正如下面摘自示例 19.1 的代码所示，**\*p** 允许改变 **p** 所指向的元素：

```
for (p = container.begin( ); p != container.end( ); p++)
    *p = 0;
```

**for** 循环遍历向量容器中的所有元素，并且把所有元素的值改为 0。

## 陷阱：编译器问题

一些编译器在声明迭代器时会存在一些问题。我们可以通过多种不同的方式来声明迭代器。例如，我们已经使用过如下这种声明方式：

```
using std::vector;
...
vector<char>::iterator p;
```

也可以使用如下代码代替上面的代码：

```
using std::vector<char>::iterator;
...
iterator p;
```

可能你也使用过下面这种不太好的声明方式：

```
using namespace std;
...
vector<char>::iterator p;
```

还有其他类似的声明方式。

编译器应当接受这些不同的声明方式。但是，我们发现一些编译器只接受其中的一部分声明方式。如果你的编译器不支持某种声明方式，那就请换其他声明方式。■

## 迭代器

迭代器是一个可以用于操作容器的对象。通过它可以对容器中的元素进行访问。迭代器是指针概念的推广。迭代器中的运算符 ==、!=、++ 以及 -- 的行为方式和它们在指针中的使用方式是一样的。一个迭代器遍历容器中所有元素的基本要点如下列代码所示：

```
STL_container<datatype>::iterator p;
for (p = container.begin(); p != container.end(); p++)
    Process_Element_At_Location p;
```

STL\_container 是容器类的名字，（例如，vector），datatype 是存储在容器中的元素的数据类型。成员函数 begin（）返回一个指向容器中第一个元素的迭代器对象。成员函数 end（）返回一个标记值，它指向越过容器中最后一个元素的位置。

## 解引用

当使用解引用运算符对迭代器对象 p 进行操作时，\*p 会得到迭代器 p 所指向的元素。在某些情况下，\*p 只有读访问权限，不允许改变元素。而在其他情况下，它允许访问元素，也允许对元素做更改。

## 自测练习题

1. 如果 v 是一个向量，那么 v.begin（）的返回值是什么？v.end（）的返回值又是什么？
2. 如果 p 是向量对象 v 的迭代器，那么 \*p 表示什么？
3. 假设 v 是 int 类型的向量。写一个 for 循环语句来输出 v 中除了第一个元素外的其

他所有元素。

## 迭代器的种类

不同的容器具有不同种类的迭代器。可以根据迭代器所支持的各种具体操作来对迭代器进行分类。向量迭代器是最为普通的一种,也就是说,向量迭代器支持所有操作。接下来,我们将再次使用向量容器来对迭代器进行说明。在本例中,我们使用向量来说明迭代器的自减操作和随机访问操作。示例 19.2 给出了另外一个程序例子,这个程序使用名为 `container` 的向量对象以及一个迭代器对象 `p`。

示例 19.2 的第 29 行使用了自减运算符。正如你所料想的那样, `p--` 操作使迭代器 `p` 移向它的前一个位置。自减运算符 “`--`” 和自加运算符 “`++`” 类似,但是它们使得迭代器的移动方向恰好相反。

自增和自减运算符既可以作为前缀操作 (`++p`),也可以作为后缀操作 (`p++`)。除了改变了 `p` 对象外,这两个运算符还可以返回一个值。返回值的具体情况与自增和自减运算符作用于 `int` 变量的情况完全类似。对于前缀形式,首先改变的是变量的值,然后再返回被改变的变量值。对于后缀形式,先返回变量的原有值,再改变变量的值。我们建议不要使用自增和自减运算符来作为返回值的表达式,我们仅用它们来改变变量的值。

下面是摘自示例 19.2 的一些代码,这些代码表明采用向量迭代器可以对向量中的元素进行随机访问 (random access),比如对于 `container`:

```
vector<char>::iterator p = container.begin( );
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
cout << "The third entry is " << *(p + 2) << endl;
```

### 随机访问

随机访问意味着只需要一步就可以直接访问任何一个特定的元素。我们使用 `container[2]` 作为向量的随机访问示例。这就是访问元素时使用的一种简单的方法:方括号运算符,方括号运算符是数组和向量用于访问元素的标准运算符。这里我们要介绍一点新知识,那就是也可以使用方括号来操作迭代器以获得对元素的随机访问。表达式 `p[2]` 是获取索引号为 2 的元素的方法。

表达式 `p[2]` 和 `*(p+2)` 是完全等价的。根据指针算法类推(见第 10 章), `(p+2)` 表示在 `p` 的前两个位置。在前面的代码中,由于 `p` 处于第一个位置(索引号为 0),因此, `(p+2)` 在第三个位置(索引号为 2)。表达式 `(p+2)` 返回一个迭代器。表达式 `*(p+2)` 对迭代器进行解引用操作。当然,也可以使用一个非负整数来替换 2 以获得指向其他元素的指针。

## 示例 19.2 双向随机访问迭代器的使用

```
1 // 双向随机访问迭代器的演示程序。
2 #include <iostream>
3 #include <vector>
4 using std::cout;
5 using std::endl;
6 using std::vector;
```

```

7  int main()
8  {
9      vector<char> container;

10     container.push_back('A');
11     container.push_back('B');
12     container.push_back('C');
13     container.push_back('D');

14     for (int i = 0; i < 4; i++)
15         cout << "container[" << i << "] == "
16             << container[i] << endl;

17     vector<char>::iterator p = container.begin();
18     cout << "The third entry is " << container[2] << endl;
19     cout << "The third entry is " << p[2] << endl;
20     cout << "The third entry is " << *(p + 2) << endl;

21     cout << "Back to container[0].\n";
22     p = container.begin();
23     cout << "which has value " << *p << endl;

24     cout << "Two steps forward and one step back:\n";
25     p++;
26     cout << *p << endl;

27     p++;
28     cout << *p << endl;
29     p--;
30     cout << *p << endl;

31     return 0;
32 }

```

同一变量的三种不同表示方式。

这种表示方式只适用于向量和数组。

这两种表示方式适用于任何随机访问迭代器。

这适合任何双向迭代器。

### 示例运行结果

```

container[0] == A
container[1] == B
container[2] == C
container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B

```

一定要注意不管是 `p[2]` 还是 `(p+2)` 都没有改变迭代器变量 `p` 的值。`(p+2)` 表达式返回指向另一个位置的迭代器，但是它也离开了原来所处的位置。在 `p[2]` 操作的背后也会出现类似的情况。而且也要注意，`p[2]` 和 `(p+2)` 所表示的内容也依赖于

迭代器  $p$  所处的位置。例如,  $(p+2)$  表示不管  $p$  的位置在哪里, 它代表的是当前位置  $p$  的后两个位置处的元素。

例如, 假设采用下面的代码来替换前面讨论的示例 19.2 的代码 (注意新添加的  $p++$  这行代码):

```
vector<char>::iterator p = container.begin( );
p++;
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
cout << "The third entry is " << *(p + 2) << endl;
```

这三个 `cout` 的输出, 将不再是下面的结果:

```
The third entry is C
The third entry is C
The third entry is C
```

而是被替换成:

```
The third entry is C
The third entry is D
The third entry is D
```

$p++$  的位置从 0 移到了 1, 因此,  $(p+2)$  现在是处于位置 3 的迭代器, 不在位置 2 上。因此,  $*(p+2)$ 、 $p[2]$  和 `container[3]` 是等价的, 而不再是和 `container[2]` 等价了。

现在, 我们应该很清楚如何根据迭代器所支持的具体操作来对迭代器进行分类了。迭代器主要分为下面这几种类型。

- 前向迭代器 (forward iterator): 可对迭代器进行 “++” 操作。
- 双向迭代器 (bidirectional iterator): 可对迭代器进行 “++” 和 “--” 操作。
- 随机访问迭代器 (random-access iterator): 可对迭代器进行 “++”、“--” 和随机访问操作。

注意上面这三种迭代器类型的操作能力依次增强: 每个随机访问迭代器也是一个双向迭代器, 每个双向迭代器也是一个前向迭代器。

正如我们将会看到的那样, 不同模板容器类有不同种类的迭代器。向量模板类的迭代器是随机访问迭代器。

注意上面提到的几个迭代器: 前向迭代器、双向迭代器和随机访问迭代器都是指迭代器的种类, 但不是类型名。真正的迭代器的类型名看起来应该像这样: `std::vector<int>::iterator`, 在本例中它碰巧是一个随机访问迭代器。

### 迭代器的种类

不同容器具有不同种类的迭代器。下面是几个主要的迭代器种类。

- 前向迭代器 (forward iterator): 可对迭代器进行 “++” 操作。
- 双向迭代器 (bidirectional iterator): 可对迭代器进行 “++” 和 “--” 操作。
- 随机访问迭代器 (random-access iterator): 可对迭代器进行 “++”、“--” 和随机访问操作。

## 自测练习题

4. 假设向量 `v` 按先后顺序包含字母 'A'、'B'、'C' 和 'D', 那么下列代码的输出是什么?

```
vector<char>::iterator i = v.begin( );
i++;
cout << *(i + 2) << " ";
i--;
cout << i[2] << " ";
cout << *(i + 2) << " ";
```

## 常量迭代器和可迭代器

对于前向迭代器、双向迭代器和随机访问迭代器这三种迭代器来说, 每一种都可以被划分为两种类型——常量 (*constant*) 迭代器和可变 (*mutable*) 迭代器, 它是依据解引用运算符对迭代器操作行为的不同而进行划分的。如果是常量迭代器, 那么解引用运算符对它操作后所产生的元素是只读的。例如, 对于一个常量迭代器 `p`, 可以把 `*p` 赋值给一个变量, 或者将其输出到屏幕上, 但是不能改变容器中的元素, 例如, 通过给 `*p` 赋值来改变容器中的元素是不允许的。对于一个可迭代器 `p` 来说, 可以对 `*p` 进行赋值, 这样将会改变容器中对应的元素。换句话说, 对于一个可迭代器 `p`, `*p` 返回一个左值。向量迭代器是可迭代器, 如下列摘自示例 19.1 的代码所示:

```
cout << "Setting entries to 0:\n";
for (p = container.begin( ); p != container.end( ); p++)
    *p = 0;
```

如果一个容器只有常量迭代器, 那对于该容器来说, 就不可能得到一个可迭代器。但是, 如果一个容器有可迭代器, 而你想要获得这个容器的常量迭代器, 那么你是可以得到常量迭代器的。当你不想让程序去改变容器中元素的值时, 你可能会想把常量迭代器作为一种错误检查手段。例如, 下列代码将会为一个名为 `container` 的向量容器生成一个常量迭代器:

```
std::vector<char>::const_iterator p = container.begin( );
```

或者等价于:

```
using std::vector<char>::const_iterator;
const_iterator p = container.begin( );
```

采用这种方式来声明 `p`, 下列代码将会产生错误消息。

```
*p = 'Z';
```

如果你对示例 19.2 中的代码做了如下替换, 效果是完全一样的。将:

```
vector<char>::iterator p;
```

替换为:

```
vector<char>::const_iterator p;
```

但是, 如果对示例 19.1 做类似的替换, 这将导致程序不能正常工作。这是因为在示例 19.1 中有下面这行代码:

常量迭代器

可迭代器



```
*p = 0;
```

注意, `const_iterator` 是一个类型名, 而 `constant iterator` 却是迭代器种类的名字。然而, 每个类型名为 `const_iterator` 的迭代器都将会是一个常量迭代器。

### 常量迭代器

常量迭代器是一种不允许改变其所指向元素的值的一种迭代器。

### 反向迭代器

有时可能需要反向遍历容器中所有元素的值。如果有一个带双向迭代器的容器, 我们可能很想使用下面的代码来试试:

```
vector<int>::iterator p;
for (p = container.end(); p != container.begin(); p--)
    cout << *p << " ";
```

上面的代码在编译时是会通过的, 这段程序或许也会在某些系统上正常运行, 但是, 这段代码中存在一个根本性的错误: `container.end()` 并不是一个常规的迭代器, 它只是一个结束标志。但是, `container.begin()` 却是一个迭代器, 而不是一个标志。

幸运的是, 可以用一个简单的方法去实现你想要做的事情。对于一个带双向迭代器的容器来说, 有一种方法可以对容器中的所有元素进行反向访问, 那就是使用所谓的反向迭代器。下列代码可以很好地对容器中的元素进行反向输出:

#### 反向迭代器

```
vector<int>::reverse_iterator rp;
for (rp = container.rbegin(); rp != container.rend(); rp++)
    cout << *rp << " ";
```

#### `rbegin()`

成员函数 `rbegin()` 返回一个指向容器中最后一个元素的迭代器。成员函数

#### `rend()`

`rend()` 返回一个标志, 它标记着反向序列中元素的“结尾”。注意, 对一个类型为 `reverse_iterator` 的迭代器, 自增运算符“++”使得迭代器从后向前移动。换句话说, “--”和“++”的操作功能互换了。示例 19.3 的程序向我们演示了一个反向迭代器的例子。

反向迭代器也有面向常量的类型, 它被命名为 `const_reverse_iterator`。

### 示例 19.3 反向迭代器

```
1 // 反向迭代器演示程序。
2 #include <iostream>
3 #include <vector>
4 using std::cout;
5 using std::endl;
6 using std::vector;

7 int main()
8 {
9     vector<char> container;
```

```

10     container.push_back('A');
11     container.push_back('B');
12     container.push_back('C');

13     cout << "Forward:\n";
14     vector<char>::iterator p;
15     for (p = container.begin( ); p != container.end( ); p++)
16         cout << *p << " ";
17     cout << endl;

18     cout << "Reverse:\n";
19     vector<char>::reverse_iterator rp;
20     for (rp = container.rbegin( ); rp != container.rend( ); rp++)
21         cout << *rp << " ";
22     cout << endl;

23     return 0;
24 }

```

### 示例运行结果

```

Forward:
A B C
Reverse:
C B A

```

### 反向迭代器

如果一个容器中包含双向迭代器，那么反向迭代器可以用来以相反的方向遍历容器中的所有元素。通常的用法如下：

```

STL_container<datatype>::reverse_iterator rp;
for (rp = c.rbegin( ); rp != c.rend( ); rp++)
    Process_At_Location p;

```

对象 *c* 是一个包含双向迭代器的容器类对象。

当使用 `reverse_iterator` 时，需要使用某种 `using` 声明或者其他某种与之等价的声明。例如，如果 *c* 是一个 `vector<int>`，那么使用下列代码就可以了：

```
vector<int>::reverse_iterator rp;
```

### 其他种类的迭代器

还有一些其他种类的迭代器，我们在本书中不会深入讨论。我们会简单地提一下两种迭代器，可能你之前见过它们的名字：输入迭代器和输出迭代器。输入迭代器本质上是一种可以用于处理输入流的前向迭代器。输出迭代器本质上是一种可以处理输出流的前向迭代器。如果你需要了解更多的细节，请查阅一些在这方面讲述更为深入的参考书。

## 自测练习题

5. 假设向量 `v` 顺序包含字母 'A'、'B'、'C' 和 'D'。下列代码的输出结果是什么？

```
vector<char>::reverse_iterator i = v.rbegin();
i++;
i++;
cout << *i << " ";
i--;
cout << *i << " ";
```

6. 假设你想运行下列代码，其中 `v` 是一个 `int` 类型的向量：

```
for (p = v.begin( ); p != v.end( ); p++)
    cout << *p << " ";
```

下列两行代码哪个可以用来声明 `p`？

```
std::vector<int>::iterator p;
std::vector<int>::const_iterator p;
```

## 19.2 容器

你可以把所有的鸡蛋放在一个篮子里，但是一定要确保这个篮子没问题。

沃尔特·沙维奇，《完美 C++》。

### 容器类

STL 的容器类是用来保存数据的，它可以是各种不同的数据结构，比如链表、队列和栈。每一个都是一个模板类，每个模板类都有一个参数用来指定存储于模板中的元素的数据类型。因此，你可以把一个链表指定为 `int` 型、`double` 型、`string` 型或其他任何你希望的类或结构体类型。每个容器模板类可能有它自己特殊的取值函数，也可能有一些用来从容器中添加数据和移除数据的赋值函数。不同容器类可能有不同种类的迭代器。例如，一个容器类可能有双向迭代器，而另一个容器类可能只有前向迭代器。但是，不管这些容器类是在什么时候被定义的，对于 STL 的所有容器类来说，迭代器运算符和成员函数 `begin()` 以及 `end()` 都具有相同的含义。

### 连续容器

#### 单向链表

#### 双向链表

连续容器把它内部的数据项组织成一个链表形式，比如说有第一个元素，第二个元素，依此类推，一直到最后一个元素。我们在第 17 章中所讨论的链表就是连续容器的一个例子；这种链表有时也被称为**单向链表**，因为每个元素只有一个链接用来指向另外一个元素的位置。STL 中没有对应于这种单链表的容器，尽管有一些的确提供了单链表的实现，一个典型的例子是 `slist`<sup>1</sup>。STL 中最简单的链表是**双向链表**，它是名

<sup>1</sup> Silicon Graphics 版本的 STL 包含 `slist`，并且它被发布在 g++ 编译器中。SGI 提供了一个针对 STL 的非常有用的参考文档，它几乎适用于每一个 STL 版本。

为 `list` 的模板类。示例 19.4 展示了这两种链表的区别，它们之间更详细的区别参见 17.1 节。

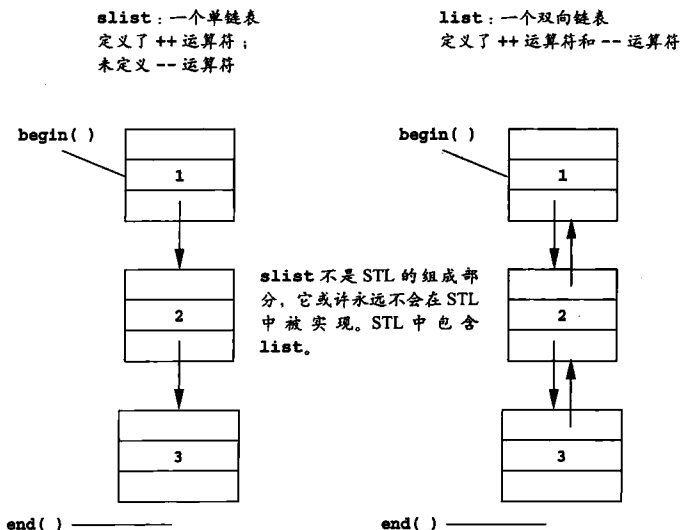
`slist<int>`  
和  
`list<int>`

示例 19.4 中的链表按顺序包含三个整数值：1、2 和 3。两个链表类型分别是 `slist<int>` 和 `list<int>`。该示例还对迭代器 `begin()` 和 `end()` 的位置进行了图示说明。在图中，我们还没向读者说明如何把这些整数输入到链表中。

在示例 19.4 中，我们把单向链表和双向链表画成在第 17 章中所讨论的节点和指针的形式。STL 类 `list` 和非标准类 `slist` 可能会（也可能不会）采用这种方式来实现。但是，当使用 STL 模板类时，它帮我们屏蔽了这些实现细节。因此，我们只需要简单地考虑数据（可能是节点，也可能不是节点）的位置和迭代器（不是指针）。你可以把示例 19.4 中的箭头看作是使用 “++” 运算符时迭代器的移动方向（它朝下）和使用 “--” 运算符时迭代器的移动方向（它朝上）。

我们给出了模板类 `slist` 来帮助读者理解连续容器。它和我们在第 17 章中所讨论的一些内容相关。对于大部分程序员来说，当提及链表时第一反应就是单向链表。但是，由于模板类 `slist` 不是一个标准模板类，因此，我们将不再对此展开进一步的详细讨论。如果在你的实现中包括了模板类 `slist`，而且你想使用它，那么它的具体使用细节和我们将要讨论的模板类 `list` 很相似。模板类 `slist` 与模板类 `list` 最大的不同是：`slist` 没有定义自减运算符（前缀和后缀）。

#### 示例 19.4 两种链表



**push\_back**

示例 19.5 给出了使用 STL 模板类 `list` 的一个简单程序。函数 `push_back` 向链表末尾增加一个元素。注意，对于链表模板类来说，通过使用解引用运算符我们可以

对数据进行读取和更改。也要注意,在使用 `list` 模板类、所有其他模板类以及 STL 中的迭代器时,要把所有相关定义都放在 `std` 命名空间中。

注意,如果我们把示例 19.5 的 `list` 和 `list<int>` 分别用 `vector` 和 `vector<int>` 替换,它们的编译和运行几乎完全一样。这种一致的使用方法也是 STL 语法的一个关键部分。

然而,在向量和链表容器之间还是有些区别的。一个主要的区别是向量容器有一个随机访问迭代器,而链表只有一个双向迭代器。例如,如果使用示例 19.2 所示的程序,它使用的是随机访问,我们把程序中所有出现的 `vector` 和 `vector<char>` 分别换成 `list` 和 `list<char>`,接下来再去编译程序,这时我们将会得到一个编译错误(即使删除了包含 `container[i]` 或 `container[2]` 的语句,程序还是会得到一个错误消息)。

示例 19.6 列出了 STL 中的一些基本的连续容器模板类。对于其他容器类,比如栈和队列,我们可以通过学习子章节“容器适配器栈和队列”中所讨论的技术来掌握它们。示例 19.7 给出了连续容器类的一些成员函数的样例。所有这些连续容器类都有一个析构函数,析构函数返回所占用的存储,以便内存回收。

内存管理

deque

`deque` 的发音是“d-queue”或者“deck”,它表示“双向队列”。双向队列是一种超级队列。对一个队列来说,可以在数据序列的末尾增加一个元素,在另一端移除一个元素。而对于一个双向队列,可以在队列的任何一端增加一个数据或者移除一个数据。双向模板类 `deque` 表示的是一个双向队列,它包含一个参数用来表示所要存储数据的数据类型。

### 连续容器

连续容器把它内部的数据项以一个链表的形式进行组织,比如说有第一个元素,第二个元素,依此类推,直到最后一个元素。我们已经讨论的连续容器模板类包括 `slist`、`list`、`vector` 以及 `deque`。

### 示例 19.5 链表模板类的用法

```
1 // 演示 STL 模板类 list 用法的程序。
2 #include <iostream>
3 #include <list>
4 using std::cout;
5 using std::endl;
6 using std::list;

7 int main( )
8 {
9     list<int> listObject;

10     for (int i = 1; i <= 3; i++)
11         listObject.push_back(i);

12     cout << "List contains:\n";
```

```

13     list<int>::iterator iter;
14     for (iter = listObject.begin( ); iter != listObject.end( );
           iter++)
15         cout << *iter << " ";
16     cout << endl;

17     cout << "Setting all entries to 0:\n"
18     for (iter = listObject.begin( ); iter != listObject.end( );
           iter++)
19         *iter = 0;

20     cout << "List now contains:\n"
21     for (iter = listObject.begin( ); iter != listObject.end( );
           iter++)
22         cout << *iter << " ";
23     cout << endl;

24     return 0;
25 }

```

#### 示例运行结果

```

List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0

```

### 示例 19.6 STL 的基本连续容器类

模板类名	迭代器类名	迭代器种类	头文件
<b>list</b> (警告: list 并非 C++ 标准库的一部分)	<b>list&lt;T&gt;::iterator</b> <b>list&lt;T&gt;::const_iterator</b>	可变双向 常量双向	<b>&lt;list&gt;</b> (依赖于具体实现)
<b>list</b>	<b>list&lt;T&gt;::iterator</b> <b>list&lt;T&gt;::const_iterator</b> <b>list&lt;T&gt;::reverse_iterator</b> <b>list&lt;T&gt;::const_reverse_iterator</b>	可变双向 常量双向 可变双向 常量双向	<b>&lt;list&gt;</b>
<b>vector</b>	<b>vector&lt;T&gt;::iterator</b> <b>vector&lt;T&gt;::const_iterator</b> <b>vector&lt;T&gt;::reverse_iterator</b> <b>vector&lt;T&gt;::const_reverse_iterator</b>	可变随机访问 常量随机访问 可变随机访问 常量随机访问	<b>&lt;vector&gt;</b>
<b>deque</b>	<b>deque&lt;T&gt;::iterator</b> <b>deque&lt;T&gt;::const_iterator</b> <b>deque&lt;T&gt;::reverse_iterator</b> <b>deque&lt;T&gt;::const_reverse_iterator</b>	可变随机访问 常量随机访问 可变随机访问 常量随机访问	<b>&lt;deque&gt;</b>

## 示例 19.7 一些连续容器的成员函数

成员函数 (n 是一个容器对象)	含义
<code>c.size( )</code>	返回容器中元素的数量
<code>c.begin( )</code>	返回位于容器中第一个元素位置的迭代器
<code>c.end( )</code>	返回位于容器中最后一个元素位置之后的迭代器
<code>c.rbegin( )</code>	返回位于容器中最后一个元素位置的迭代器。与 <code>reverse_iterator</code> 一起使用。不是 <code>slist</code> 的成员函数
<code>c.rend( )</code>	返回位于容器中第一个元素位置之前的迭代器。与 <code>reverse_iterator</code> 一起使用。不是 <code>slist</code> 的成员函数
<code>c.push_back(Element)</code>	在数据序列末尾插入 <code>Element</code> 。不是 <code>slist</code> 的成员函数
<code>c.push_front(Element)</code>	在数据序列前面插入 <code>Element</code> 。不是 <code>vector</code> 的成员函数
<code>c.insert(Iterator, Element)</code>	在 <code>Iterator</code> 之前插入一个 <code>Element</code> 元素的拷贝
<code>c.erase(Iterator)</code>	删除 <code>Iterator</code> 位置处的元素。返回一个紧随其后的迭代器。如果最后一个元素被移除，则返回 <code>c.end( )</code>
<code>c.clear( )</code>	删除容器中的所有元素，函数返回 <code>void</code>
<code>c.front( )</code>	返回指向数据序列头部元素的引用。与 <code>c.back( )</code> 类似
<code>c1 == c2</code>	如果 <code>c1.size( ) == c2.size( )</code> 并且 <code>c1</code> 中的每一个元素等于 <code>c2</code> 中对应的元素，则返回真
<code>c1 != c2</code>	<code>!(c1 == c2)</code>

本节所讨论的所有连续容器都有一个默认构造函数、一个拷贝函数以及其他各种构造函数来初始化容器的默认值或指定元素的值。每一个连续容器都有一个析构函数，返回程序所占用的存储，使得内存可以回收。此外，每个连续容器还有一个重载赋值运算符。

## 陷阱：迭代器和删除元素

给一个容器增加一个元素或者删除一个元素能够影响到其他迭代器。一般而言，增加或删除元素后，并不能保证迭代器在执行增加和删除操作前后指向相同的元素。但是，一些迭代器的确可以保证执行增加和删除操作后迭代器的位置并不会移动；当然，如果迭代器的位置正好处于被删除的元素位置处，这种情况例外。

到目前为止，在我们所看到的模板类中，`list` 和 `slist` 保障在执行增加和删除操作时，它们的迭代器不会移动；当然，如前面所述，如果迭代器的位置正好处于被删除的元素位置处，这种情况除外。模板类 `vector` 和 `deque` 对此不做保障。■

### 提示：容器中的类型定义

当用 STL 容器类进行编程时，容器类内包含了一些可以方便使用的类型定义。我们已经看到 STL 容器类可能包含类型名：`iterator`、`const_iterator`、`reverse_iterator` 以及 `const_reverse_iterator`（因此，在这些容器类的内部必然包含这些类型名的类型定义）。通常，也包含一些其他的类型定义。

类型 `value_type` 是存储在容器中元素的数据类型，`size_type` 是一个无符号整数类型，它是成员函数 `size` 的返回类型。例如，`list<int>::value_type` 和 `int` 是一样的，它是 `int` 的另外一种名字。到目前为止，我们所讨论的所有模板类都包含已经定义好的 `value_type` 和 `size_type`。■

### 自测练习题

7. `vector` 和 `list` 的主要区别是什么？
8. 在模板类 `slist`、`list`、`vector` 以及 `deque` 中，哪一个包含成员函数 `push_back`？
9. 在模板类 `slist`、`list`、`vector` 以及 `deque` 中，哪一个包含随机访问迭代器？
10. 在模板类 `slist`、`list`、`vector` 以及 `deque` 中，哪一个可能有可迭代器？

### 容器适配器栈和队列

容器适配器是在其他类之上实现的模板类。例如，默认情况下，栈（`stack`）模板类的实现基于双向队列模板类，这意味着栈的底层实现是一个双向队列，数据存放在双向队列上。但是，我们看不到实现细节，我们看到的只是一个栈，一个简单的后进先出的数据结构。

其他容器适配器类还有队列模板类和优先队列（`priority_queue`）模板类。第 17 章讨论了栈和队列。相对于队列，**优先队列**有额外的属性，当元素加入到优先队列时，它给每一个元素分配一个优先权值。如果所有数据有相同的优先权值，那么从一个优先队列中移除一个元素的方式和从一个队列中移除元素的方式完全一样。如果每个数据项有不同的优先权值，那么，优先队列会保障优先权值高的元素在优先权值低的元素之前移出队列。在这里，我们不会对优先队列的细节展开讨论，只是为那些熟悉这一概念的读者简要介绍一下。

尽管适配器模板类是构建在默认的容器类之上的，但是出于高效性或者其他某些原因（这主要根据应用的需求而定），我们也可以选择为适配器模板类指定一个特定的底层容器类。例如，任何一个连续容器都可能充当栈模板类的底层容器，任何非 `vector` 的连续容器都可以充当队列模板类的底层容器。在 STL 中，栈和队列的默认底层数据结构是双向队列 `deque`。对于一个优先队列来说，默认的底层容器是向量容器类。如果你对默认的底层容器类型比较满意，那么对你来说，一个容器适配器看起来和其他任何模板容器类是一样的。例如，对于一个 `int` 类型的栈，当它使用默认的底层容器时，栈模板类的类型名是 `stack<int>`。如果你希望指定底层容器，把它替

优先队列

警告！



换为向量模板类，你将会使用 `stack<int, vector<int> >` 作为类型名。一定要确保两个连续的尖括号 “>” 中要插入一个空格。在本书中，我们总会使用默认的底层容器。

示例 19.8 给出了栈模板类的成员函数和其他一些具体细节。示例 19.9 给出了队列模板类的具体细节。示例 19.10 给出了一个使用栈模板类的简单例子。



陷阱：底层容器

如果你为适配器类指定了底层容器，注意不要在类型表达式中连续输入两个尖括号 “>”，它们之间一定要有空格，否则编译器将无法识别。使用 `stack<int, vector<int> >` 时，注意两个连续的尖括号 “>” 之间有一个空格。不要使用 `stack<int, vector<int>>`。■

自测练习题

- 11. 栈模板适配器类包含哪种迭代器（前向、双向或者随机访问）类型？
- 12. 队列模板适配器类包含哪种迭代器（前向、双向或者随机访问）类型？
- 13. 如果 `s` 是一个 `stack<char>` 对象，那么 `s.pop()` 的返回值是什么类型？

示例 19.8 栈模板类

栈适配器模板类详细说明

类型名：`stack<T>` 或者 `stack<T, Sequence_Type>`，`T` 是栈中元素的类型。  
库的头文件：`<stack>`，它的定义放在 `std` 命名空间中。  
类中定义的类型：`value_type`，`size_type`。  
栈适配器中没有迭代器。

成员函数样例

成员函数（ <code>s</code> 是一个栈对象）	含义
<code>s.size()</code>	返回栈中元素的数目
<code>s.empty()</code>	如果栈是空的，返回 <code>true</code> ，否则，返回 <code>false</code>
<code>s.top()</code>	返回栈顶元素的可变引用
<code>s.push(Element)</code>	在栈顶插入一个 <code>Element</code> 元素的拷贝对象
<code>s.pop()</code>	删除栈顶元素，注意函数返回为 <code>void</code> ，而不是返回被移除的元素
<code>s1 == s2</code>	如果 <code>s1.size() == s2.size()</code> ，而且 <code>s1</code> 中的每一个元素等于 <code>s2</code> 中对应的每个元素，那么返回 <code>true</code> ，否则返回 <code>false</code>

栈模板类也有一个默认构造函数、拷贝构造函数和一个以连续类对象为参数的构造函数，这个连续类对象负责初始化栈中的元素。栈模板类也有一个析构函数，用来返回程序所占的所有存储，以便内存回收，此外它还有一个重载的赋值运算符。

## 示例 19.9 队列 (queue) 模板类

### queue 适配器模板类详细说明

类型名: `queue<T>` 或者 `queue<Sequence_Type, T>`, `T` 是队列中元素的类型。出于程序高效性的考虑, `Sequence_Type` 不能是向量类型。

库的头文件: `<queue>`, 它的定义放在 `std` 命名空间中。

类中定义的类型: `value_type`, `size_type`。

队列适配器中没有迭代器。

### 成员函数样例

成员函数 ( <code>q</code> 是一个栈对象)	含义
<code>q.size( )</code>	返回栈中元素的数目
<code>q.empty( )</code>	如果队列是空的, 返回 <code>true</code> ; 否则, 返回 <code>false</code>
<code>q.front( )</code>	返回队列头部元素的可变引用
<code>q.back( )</code>	返回队列的最后一个元素的可变引用
<code>q.push(Element)</code>	在队列尾部插入一个 <code>Element</code> 元素的拷贝对象
<code>q.pop( )</code>	删除队列头部元素, 注意函数返回为 <code>void</code> , 而不是返回被移除的元素
<code>q1 == q2</code>	如果 <code>q1.size( ) == q2.size( )</code> , 而且 <code>q1</code> 中的每一个元素等于 <code>q2</code> 中对应的每个元素, 那么返回 <code>true</code> , 否则返回 <code>false</code>

队列模板类也有一个默认构造函数、一个拷贝构造函数和一个以任意连续类为参数的构造函数, 这个连续类对象负责初始化队列中的元素。队列模板类也有一个析构函数, 用来返回程序所占的所有存储, 以便内存回收, 此外它还有一个重载的赋值运算符。

## 示例 19.10 使用栈模板类的程序

```

1 // 演示来自 STL 中栈模板类用法的程序。
2 #include <iostream>
3 #include <stack>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7 using std::stack;

8 int main( )
9 {
10     stack<char> s;
11     cout << "Enter a line of text:\n";
12     char next;
13     cin.get(next);
14     while (next != '\n')
15     {
16         s.push(next);
17         cin.get(next);
18     }

```

```

19     cout << "Written backward that is:\n";
20     while ( ! s.empty() )
21     {
22         cout << s.top();
23         s.pop();
24     }
25     cout << endl;

26     return 0;
27 }

```

成员函数 `pop` 删除了一个元素，但是它不会返回被删除的元素。`pop` 是一个 `void` 函数。因此，我们需要使用 `top` 来访问被删除的元素。

### 示例运行结果

```

Enter a line of text:
straw
Written backward that is:
warts

```

## 关联式容器 `set` 和 `imap`

实际上，可以把关联式容器看成是一个非常简单的数据库。关联式容器用于存储数据，比如存储结构体和其他任何类型的数据。每一个数据元素有一个关联值，我们将其称为键 (key)。例如，如果有一个数据，它是表示雇员记录的结构体，那么，或许可以使用雇员的社会安全号作为键。通过键可以用来获取记录项。虽然键的类型和待存储数据的类型彼此之间通常是有联系的，但实际上并不要求它们之间有任何的关联。一个非常简单的例子是，可以把每个数据项作为它自己的键。例如，在一个集合 (set) 中，每个元素是它自己的键。

从某种意义上来说，集合 (set) 模板类是我们想象到的一个最简单的容器。它存储没有重复的元素。当一个元素第一次被插入到集合中后，相同元素的再次插入将不起任何作用，这样在集合中同一个元素只会出现一次。每一个元素都有自己的键。因此，对于集合来说，基本上我们只需要在集合中进行三种操作：加入和删除元素的操作，以及查询一个元素是否在集合中已经出现了的操作。像所有 STL 类一样，集合模板类也把效率作为代码实现的目标。为了高效地工作，集合对象以特定顺序存储值。我们可以用以下代码来指定用以存储数据的顺序：

```
set<T, Ordering > s;
```

其中的 `Ordering` 表示的是一个顺序关系，它接收两个类型为 `T` 的参数，返回一个布尔值<sup>2</sup>。`T` 是所存储的元素的类型。如果没有指定顺序，集合会假设排序使用 “<”

2 次序关系必须是严格弱次序。一般而言，大部分用于实现运算符<的次序都是严格弱次序。对于想了解严格弱次序具体细节的人来说，他只需要知道严格弱次序一定是下面几种形式之一：

(非反射) `Ordering(x, x)` 总是错误的；

(反对称) `Ordering(x, y)` 必然推出 `!Ordering(y, x)`；

(传递性) `Ordering(x, y)` 且 `Ordering(y, z)` 必然推出 `Ordering(x, z)`；

(等价传递性) 如果 `x` 等价于 `y` 且 `y` 等价于 `z`，那么 `x` 等价于 `z`。

如果 `Ordering(x, y)` 和 `Ordering(y, x)` 都不成立，那么两个元素 `x` 和 `y` 是等价的。

关系运算符。示例 19.11 给出了关于集合模板类的一些基本的使用细节。示例 19.12 给出了一个简单例子，它向我们展示了模板类 `set` 的一些成员函数的用法。

### 映射

**映射 (map)** 本质上是一个按顺序排列的对 (pair) 集合函数。对于每一个出现在对中的值 `first`，最多只有一个值 `second`，比如 `map` 中的键值对 (`first`, `second`)。在 STL 中，模板类 `map` 是 STL 中的 `map` 对象的具体实现。例如，如果你想为每一个字符串名分配一个特定的值，那就声明一个 `map` 对象，如下列代码所示：

```
map<string, int> numberMap;
```

`string` 值就是我们所称的键，`numberMap` 对象可以为 `string` 类型的键关联一个唯一的 `int` 值。

### 关联数组

还有另外一种方法，就是把映射看作是一个关联数组。一个传统的数组可以把一个索引号映射成为一个值。例如，`a[10]=5` 表示在索引号为 10 的数组元素位置处存储数值 5。一个关联数组允许我们使用自己选择的数据类型来定义数组的索引号。例如，`numberMap["c++"]=5` 将会把整数 5 和字符串 "c++" 关联起来。为了方便起见，在 `map` 中定义方括号运算符 "`[]`"，它允许我们使用像数组一样的符号来访问 `map` 对象。当然，如果我们想使用插入和查找方法，那也是可以的。

和 `set` 对象类似，`map` 对象根据键来对存储的元素进行排序。你可以在尖括号符号 "`<>`" 中增加一项，作为第三项，用来指定键的顺序。如果没有指定顺序，将会使用默认顺序。`map` 所能使用的排序的限制条件和集合模板类对排序的限制条件是相同的。注意，排序仅针对键的值而言。上面声明中的第二个类型可以是任意类型，而且不需要和排序有什么关系。和集合对象一样，对 `map` 对象中被存储元素进行排序也是出于程序效率的原因。

从映射对象中增加和获取数据的一种最简单的方法就是使用 "`[]`" 运算符。给定一个映射对象 `m`，表达式 `m[key]` 将返回一个和 `key` 关联的数据元素的引用。如果映射中没有 `key` 这条记录，那么映射会为此 `key` 创建一条新的记录，并使用默认值作为这条记录的数据元素。通过这种方式，可以为映射增加一条新的记录或者替换已有的一条记录。例如，`m[key] = newData`；这条语句将会在 `key` 和 `newData` 之间建立一个新的关联。注意，操作映射时必须非常仔细，不要错误地为映射创建一条记录。例如，如果你执行了一条语句 `val = m[key]`；，你希望的是得到与 `key` 关联的值，但是你不小心把 `key` 值输错了，这个输入错误的 `key` 并不在映射中，那么映射将会为此 `key` 创建一条新的记录，并给它分配一个默认值，接着会把这个值赋给变量 `val`。

## 示例 19.11 集合 (set) 模板类

### 集合模板类的详细说明

类型名：`set<T>` 或 `set<T, Ordering>`，`T` 是集合中元素的类型。`Ordering` 用来排序所存储的元素。如果没有给出 `Ordering`，就是用二元运算符 "`<`"。

库的头文件：`<set>`，它的定义在 `std` 命名空间中。

类中定义的类型：`value_type`, `size_type`。

迭代器：`iterator`, `const_iterator`, `reverse_iterator`, `const_reverse_iterator`。所有迭代

器都是双向迭代器，不包括 `const_` 的那些迭代器都是可迭代器。`begin()`、`end()`、`rbegin()` 和 `rend()` 的行为方式和前面的相同。增加或者删除元素不会影响迭代器，当然，如果迭代器正好处于被删除的元素上，这种情况除外。

### 成员函数样例

成员函数 ( <i>s</i> 是一个集合对象)	含义
<code>s.insert(Element)</code>	在集合中插入一个 <i>Element</i> 的拷贝。如果 <i>Element</i> 已经在集合中了，那么该操作不会对集合有任何影响
<code>s.erase(Element)</code>	从集合中删除元素。如果集合中没有该元素，那么该操作不会对集合产生任何影响
<code>s.find(Element)</code>	返回一个指向集合中 <i>Element</i> 的拷贝的迭代器。如果 <i>Element</i> 不在集合中，返回 <code>s.end()</code> 。它的实现与迭代器是否是可迭代器无关
<code>s.erase(Iterator)</code>	删除 <i>Iterator</i> 位置处的元素
<code>s.size()</code>	返回集合中元素的数目
<code>s.empty()</code>	如果集合是空，返回 <code>true</code> ；否则，返回 <code>false</code>
<code>s1 == s2</code>	如果两个集合包含相同的元素，则返回 <code>true</code> ；否则，返回 <code>false</code>

集合模板类也包含一个默认构造函数、一个拷贝构造函数，以及其他此处未提及的特定构造函数。它也有一个析构函数，返回所有占用的存储供内存回收，另外，它还包括一个重载的赋值运算符。

### 示例 19.12 使用集合模板类的程序

```

1 // 集合模板类用法的演示程序。
2 #include <iostream>
3 #include <set>
4 using std::cout;
5 using std::endl;
6 using std::set;
7 int main()
8 {
9     set<char> s;
10
11     s.insert('A');
12     s.insert('D');
13     s.insert('C');
14     s.insert('C');
15     s.insert('B');
16
17     cout << "The set contains:\n";
18     set<char>::const_iterator p;
19     for (p = s.begin(); p != s.end(); p++)
20         cout << *p << " ";
21     cout << endl;

```

在集合中，无论一个元素被添加多少次，集合中只能保留一份这个元素。

```

21     cout << "Set contains 'C': ";
22     if (s.find('C')==s.end( ))
23         cout <<" no " << endl;
24     else
25         cout <<" yes " << endl;

26     cout << "Removing C.\n";
27     s.erase('C');
28     for (p = s.begin( ); p != s.end( ); p++)
29         cout << *p << " ";
30     cout << endl;

31     cout << "Set contains 'C': ";
32     if (s.find('C')==s.end( ))
33         cout << " no " << endl;
34     else
35         cout << " yes " << endl;

36     return 0;
37 }

```

### 示例运行结果

```

The set contains:
A B C D
Set contains 'C': yes
Removing C.
A B D
Set contains 'C': no

```

示例 19.13 给出了 map 模板类的一些基本的说明。为了理解这些具体细节，**pair** 我们需要了解一些关于 pair 模板类的详细知识。

### 示例 19.13 map 模板类

#### map 模板类详细说明

类型名：map<KeyType, T> 或 map<KeyType, T, Ordering> 用于 map 对象，它关联 ("maps") 中类型为 KeyType 的元素和类型为 T 的元素。出于存储效率的考虑，Ordering 主要用于根据键的值对元素进行排序。如果没有给出 Ordering，默认使用二元运算符 "<" 来进行排序。

库的头文件：<map>，它的定义在 std 命名空间中。

定义的类型：引入 key\_type 作为键的数据类型，mapped\_type 作为映射值的数据类型，以及 size\_type。（因此，所定义的数据类型 key\_type 就是我们上面所说的 KeyType。）

迭代器：iterator、const\_iterator、reverse\_iterator 和 const\_reverse\_iterator。所有迭代器都是双向迭代器，不包括 const\_ 的那些迭代器既不是常量迭代器也不是可变速代器，是处于它们二者之间的一种迭代器。例如，如果 p 的类型是迭代器，那么它可以改变 key 的值，但是不能改变类型 T 的值。把所有的迭代器看成常量，这可能是最好的，或者至少是首选的。begin()、end()、rbegin() 和 rend() 的行为方式和前面的相同。增加或者删除元素不会影响迭代器。如果迭代器正好处于被删除的元素上，这种情况除外。

成员函数样例

成员函数 (m 是一个 map 对象)	含义
<code>m.insert(Element)</code>	在 map 中插入一个 Element 元素。Element 类型为 const KeyType, T, 任何类型为 pair<KeyType, T> 的键值。如果插入成功，返回 m 对象的第二个属性为真，迭代器位于被插入的元素位置。
<code>m.erase(Target_Key)</code>	删除键为 Target_Key 的元素
<code>m.find(Target_Key)</code>	返回一个指向键为 Target_Key 的迭代器。如果没找到待搜索的键，返回 m.end()。
<code>m[Target_Key]</code>	返回键 Target_Key 所关联的对象的引用。如果 map 中还没有包含这个对象，那么插入一个类型为 T 的默认对象
<code>m.size()</code>	返回 map 中键值对 (pair) 的数目
<code>m.empty()</code>	如果 map 是空，返回 true，否则，返回 false
<code>m1 == m2</code>	如果两个 map 包含相同的键值对，则返回 true，否则，返回 false

map 模板类同样也包含一个默认构造函数、一个拷贝构造函数，以及其他一些在此处未提及的特定构造函数。它也有一个析构函数，返回所占用的存储供内存回收，另外，它还包括一个重载的赋值运算符。

STL 模板类 pair <T1, T2> 中的对象为一对值，比如第一个元素的类型为 T1，第二个元素的类型为 T2。如果 aPair 是一个类型为 pair <T1, T2> 的对象，那么 aPair.first 是第一个元素，它的数据类型为 T1，aPair.second 是第二个元素，它的数据类型为 T2。成员变量 first 和 second 都是公共成员变量，因此，不需要取值函数和赋值函数。

pair 模板的头文件是 <utility>。因此，使用 pair 模板类时，需要把下列代码（或者其他类似代码）引入到文件中：

```

#include <utility>
using std::pair;

```

map 模板类使用 pair 模板类来存储键和数据元素的关联。例如，下面这行定义：

```

map<string, int> numberMap;

```

如果我们增加下面这行代码所表示的映射：

```

numberMap["c++"] = 10;

```

那么，当我们使用迭代器访问这个键值对时，iterator->first 表示键 "C++"，而 iterator->second 表示数据的值 10。

示例 19.14 给出了一个简单的例子来说明如何使用模板类 map 的成员函数。我们将会介绍其他四个关联容器，但我们并不会对这四个关联容器展开详细的讨论。模板类 multiset 和 multimap 在本质上分别与 set 和 map 相同，不同的是 multiset 允许有多个重复的元素，而 multimap 允许每个键关联多个值。在 STL 中，还包含 hash\_set 类和 hash\_map 类的实现。这些模板类在本质上和 set 及 map 是相同的，不同的是它们使用的是哈希表来实现的。我们在第 17 章中讨论了哈希表。

set 和 map 类的大部分实现都是使用平衡二叉树而不是哈希表。在一个平衡二叉树中，根节点左侧的所有节点数和其右侧的节点数大致相等。关于二叉搜索树的讨论也是在第 17 章中展开的，但是我们并没有对如何平衡一棵二叉树的问题进行详细讨论。

## 效率

STL 的实现一直追求最佳效率。例如，set 和 map 中的元素按顺序存储，这样在它们内部搜索元素的效率可以变得更加高效。

每一个模板类的每一个成员函数都有一个保证的最大运行时间。这些最大运行时间使用一个称为 big- $O$  (大- $O$ ) 的符号来表示，我们在 19.3 节中将会对此展开讨论 (19.3 节针对我们已经讨论过的一些容器成员函数，给出了一些保证运行时间。关于这些时间，我们将会“容器访问运行时间”小节中给出)。本章的其余部分将会描述某些函数的最大保证运行时间。

### 示例 19.14 使用 map 模板类的程序

```

1 // 使用 map 模板类的演示程序。
2 #include <iostream>
3 #include <map>
4 #include <string>
5 using std::cout;
6 using std::endl;
7 using std::map;
8 using std::string;

9 int main( )
10 {
11     map<string, string> planets;

12     planets["Mercury"] = "Hot planet";
13     planets["Venus"] = "Atmosphere of sulfuric acid";
14     planets["Earth"] = "Home";
15     planets["Mars"] = "The Red Planet";
16     planets["Jupiter"] = "Largest planet in our solar system";
17     planets["Saturn"] = "Has rings";
18     planets["Uranus"] = "Tilts on its side";
19     planets["Neptune"] = "1500 mile-per-hour winds";
20     planets["Pluto"] = "Dwarf planet";

21     cout << "Entry for Mercury - " << planets["Mercury"]
22         << endl << endl;

23     if (planets.find("Mercury") != planets.end( ))
24         cout << "Mercury is in the map." << endl;
25     if (planets.find("Ceres") == planets.end( ))
26         cout << "Ceres is not in the map." << endl << endl;

27     cout << "Iterating through all planets: " << endl;
28     map<string, string>::const_iterator iter;
29     for (iter = planets.begin( ); iter != planets.end( ); iter++)

```

迭代器将按键的排序结果输出映射对象。在本例中输出将按照 planet 字母顺序列出。



```

30     {
31         cout << iter->first << " - " << iter->second << endl;
32     }
33     return 0;
34 }

```

### 示例运行结果

```

Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.
Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile-per-hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid

```

---

### 自测练习题

- 为什么 set 模板类中的元素要按照某种顺序进行存储?
- set 可以有类类型的元素吗?
- 假设 s 的类型是 set<char>。如果 'A' 在 s 中,那么 s.find('A') 的返回值是什么? 如果 'A' 不在 s 中,那么 s.find('A') 的返回值是什么?
- 执行下列代码后,在映射 mymap 中将会有多少个元素?

```

map<int, string> mymap;
mymap[5] = "c++";
cout << mymap[4] << endl;

```

## 19.3 泛型算法

“如果你从 365 个中取出一个,还剩多少个?”

“当然还剩下 364 个。”

矮胖子好像有点不相信,说:“我倒要看看在纸上是怎么算的。”

刘易斯·卡罗尔,《爱丽丝镜中奇遇记》

本节讨论 STL 中的一些基本函数模板。在此,我们并不会对所有函数模板展开全面的讨论,但是我们将会向读者展示大量的样例,使读者能够理解 STL 中所包含的那些内容,也会向读者充分展示具体的实现细节,使读者能够慢慢开始使用这些模板函数。

## 泛型算法

有时候, 我们也把这些模板函数称为泛型算法。使用术语算法是有原因的。回顾一下之前的内容, 算法只是执行一个任务的一套指令集。算法可以使用任何编程语言来表达, 包括 C++ 编程语言。但是, 当使用“算法”这个词时, 程序员通常想到的是使用英语或者伪代码来书写一个非正式的算法表示。这种表示方法通常被认为是函数定义程序代码的抽象表示。它给出了算法中重要的、具体的实现细节, 但它不会给出具体详细的实现代码。STL 指定了模板函数底层算法的某些重要的、具体的实现细节, 这就是为什么它们有时候被称为泛型算法的原因。这些 STL 函数模板不仅仅能够以函数实现者所期望的方式来提供值, 它们还可以实现许多其他功能。如果能让 STL 中函数模板的实现能够满足标准的话, 那么这些实现必须满足函数模板的最小实现要求。在大部分情况下, 它们的实现必须满足保证运行时间。这就为函数接口增加了一个全新的度量标准。在 STL 中, 接口不仅告诉程序员函数是做什么的, 而且还包括如何使用函数, 以及任务多快会被完成。在一些案例中, 甚至指定了具体使用哪个算法, 尽管并没有给出精确的、详细的编码。此外, 在 STL 中可能会为某些函数指定具体的算法, 之所以这么做, 是因为它清楚所指定的算法的效率。对代码高效性保证进行具体说明是 STL 中一个很关键的、新增的知识点。在本章中, 我们将使用术语泛型算法、泛型函数和 STL 函数模板, 这些术语的意思都是一样的。

为了使用一些术语来讨论这些模板函数或者泛型算法的效率, 我们首先给出一些通常如何对算法效率进行度量的背景知识。

## 运行时间和大-O表示法

如果你问一个程序员他或她的程序有多快时, 你可能期望得到一个具体的答案, 比如“两秒”。然而, 一个程序的速度并不能使用一个数字来给出。如果程序的输入比较大, 那么程序运行所花的时间会比那些相对较小的输入所花的时间更长。我们会期望一个存储数值的程序存储 10 个数值所花的时间比存储 1000 个所花的时间要更少。可能, 存储 10 个数值需要 2 秒, 但是存储 1000 个数值需要 10 秒。程序员如何来回答“你的程序运行得有多快”这个问题? 程序员将不得不给出一个表, 表中的值给出对于不同大小的输入程序分别运行了多长时间。例如, 表的形式可能如示例 19.15 所示。这张表并没有给出一个单个时间, 它给出了程序在不同输入大小下不同的运行时间。

## 数学函数

这张表是对一个在数学上被称为函数(function)的概念的描述。就像一个(非空) C++ 函数接收输入参数并返回一个值一样, 这个函数也接收一个程序输入大小的参数, 并返回一个值, 表示在对应的输入大小下程序的运行时间。如果我们调用这个函数  $T$ , 那么  $T(10)$  的运行时间是 2 秒,  $T(100)$  的运行时间是 2.1 秒,  $T(1000)$  的运行时间是 10 秒,  $T(10,000)$  的运行时间是 2.5 分钟。这张表只是函数  $T$  的一部分样本值。对于每个输入大小, 程序都会有一个对应的运行时间值。因此, 尽管并没有在表中列出这些值, 但同样存在  $T(1)$ ,  $T(2)$ ,  $\dots$ ,  $T(101)$ ,  $T(102)$  等的值。对于任何正整数  $N$ ,  $T(N)$  是程序排序  $N$  个数所花费的时间。函数  $T$  被称为程序的运行时间。

## 运行时间

到目前为止, 我们一直假设这个排序程序对于任意  $N$  个输入数所运行的时间都是相同的。实际情况并非如此。如果列表已经排好序了, 那么程序会花费更少的运行时间。在这种情况下,  $T(N)$  被定义为对“最困难”的列表进行排序所花费的时间, 这个

**最坏情况运行时间** 时间是指对  $N$  个元素进行排序时程序运行的最长时间。这被称为“最坏情况运行时间”。在本章中，当我们给出一个算法或者一段代码的运行时间时，我们都是指最坏情况运行时间。

一个程序或者一个算法所花费的时间通常是由一个公式给出的，比如： $4N + 3$ ， $5N + 4$ ，或  $N^2$ 。如果运行时间  $T(N)$  是  $5N + 5$ ，那么，对于大小为  $N$  的输入来说，程序的运行时间将为  $5N + 5$  个时间单元。

下列代码用来对  $N$  个元素的数组进行搜索，它的目的是判断一个特定值 `target` 是否在数组中：

```
int i = 0;
bool found = false;
while (( i < N ) && !(found))
    if (a[i] == target)
        found = true;
    else
        i++;
```

示例 19.15 一些函数的运行时间值

输入大小	运行时间
10 numbers	2 seconds
100 numbers	2.1 seconds
1000 numbers	10 seconds
10,000 numbers	2.5 minutes

我们想计算出执行这段代码将会花费多长时间的预估值。我们想要得到一个预估值，它不依赖于所使用的计算机。这是有原因的，因为要么我们不知道将使用哪个计算机，要么有可能我们会在不同的时间段使用几个不同的计算机来运行这个程序。

**操作** 得到预估值的一种可能的办法是计算出程序的执行“步骤 (steps)”数，但是决定什么是一个步骤却不是一件容易的事。在这种情况下，我们通常计算操作 (operations) 数来代替步骤数。术语“操作”几乎和术语“步骤”一样很模糊，但在实际中，至少我们对“操作”有一定的共识。对于上一段代码，每使用一次接下来的任何一个运算符都将算是一次操作：`=`、`<`、`&&`、`!`、`[]`、`==` 和 `++`。除了执行这些操作外，计算机还必须做其他事情，但和这些运算符相关的操作似乎是计算机正在做的主要工作，我们假设这些主要工作占据了运行这段代码的大部分时间。因此，实际上我们对时间的分析是假设除了上面列举的几个操作外，其他任何操作根本不花费时间。程序运行的总时间等于执行这些主要操作所需要花费的时间。尽管很明显这是理想化的计算方式，不是完全真的、实际的运行时间，但实践证明这种简化假设在实际中是可以取得良好效果的，因此经常用它来分析程序和算法的效率。

即使我们对假设进行了简化,但还是必须考虑两种情况:数组中存在搜索值 `target`, 数组中没有搜索值 `target`。我们首先考虑搜索目标不在数组中的情况。我们知道,程序所要执行的操作数依赖于待搜索的数组中元素的个数  $N$ 。在循环 (loop) 执行之前,程序将会执行两次关于运算符 “=” 的操作。由于我们假设搜索目标不在数组中,那么程序将会执行  $N$  次循环,数组中的每个元素都会在循环过程中出现一次。每执行一次循环,下列运算符都会被执行一次: `<`、`&&`、`!`、`[]`、`==` 和 `++`。对于每一次循环迭代都将增加六个操作。最后,  $N$  次迭代后,程序将再次检查布尔表达式,并会发现表达式的值为 `false`。这又会增加最后的三次操作 (`<`、`&&`、`!`)<sup>3</sup>。如果我们对所有操作次数进行统计可以得出,当搜索目标不在数组中时,总共执行了  $6N + 5$  次操作。如果搜索目标在数组中,那么操作数是  $6N + 5$  或者更少,我们将会把关于这一点的证明作为练习题留给读者完成。因此,对于任何  $N$  个元素组成的数组以及待搜索的目标值,程序在最坏情况下的运行时间是  $T(N) = 6N + 5$  次操作。

我们刚刚已经确定了搜索代码的最差运行时间是  $6N + 5$  次操作。但是,并不是每次操作都是一个传统意义上的时间单位,比如纳秒、秒或分钟。如果我们想知道这个算法在某个特定计算机上的运行时间,我们必须知道这个计算机执行一次操作需要多长时间。如果一次操作在 1 纳秒内完成,那么程序执行时间将是  $6N + 5$  纳秒。如果一次操作在 1 秒内完成,那么程序执行时间将是  $6N + 5$  秒。如果我们使用一个比较慢的计算机,它每执行一次操作需要花费 10 秒,那么代码的总共运行时间是  $60N + 50$  秒。一般而言,如果执行一次操作花费计算机  $c$  纳秒,那么实际的运行时间将是大约  $c(6N + 5)$  纳秒 (我们之所以说大约,那是因为我们做了些简单的假设,因此,这个时间不会是绝对精确的实际运行时间)。这意味着我们的运行时间  $6N + 5$  是一个非常粗略的估计值。为了使得运行时间能够用纳秒来表示,我们还必须乘一个常量,常量的大小取决于你所使用的具体计算机。我们给出的估计值只有加上这个倍乘常量后才会准确。

## 大- $O$ 记法

对于刚刚所讨论的运行时间估计值,我们通常会使用大- $O$  记法来表示 ( $O$  代表的是 “Oh”, 而不是数字零)。假设我们估计运行时间是  $6N + 5$  次操作,另外,假设我们也知道不管每个不同的操作实际上到底花费了多长时间,总会存在一个常量因子  $c$  使得程序真正的运行时间小于或等于

$$c(6N + 5)$$

在这两个假设前提下,我们可以说代码 (或者说程序或算法) 运行时间为  $O(6N + 5)$ 。这个表达通常读作 “ $6N + 5$  的大  $O$ ”。这种表达实际上并不需要我们知道常量  $c$  具体是多少。毫无疑问,这个值会随着计算机的不同而不同,但是我们必须知道对于任何一个合理的计算机总会有一个具体的  $c$  值。如果计算机非常快,  $c$  可能比 1 小——比如说,可能是 0.001。如果计算机非常慢,  $c$  可能会很大,比如说 1000。此外,改变时间单位 (比如说从纳秒到秒) 仅仅涉及常量的改变,因此没必要给出具体的时间单位。

一定要注意大- $O$  估计是一个上限估计。我们总是会取实际计数中的高值而不是低

3 由于我们简化了评估,并没有把 `!(found)` 这个操作计算在内,因此,实际上我们只有两次操作,而不是三次操作。但是,对我们来说,重要的是要得到运行时间的上界。因此,如果我们增加一个多余的操作并不会产生什么影响。

值来作为估计值。而且还要注意,当我们在进行一个大- $O$ 估计时,我们没有必要确定一个非常准确的操作执行次数。我们只需要使得新估计的值乘以倍乘常量后接近正确值即可。如果我们的估计值比实际值大两倍,那这也是一个不错的估计值。

#### 任务大小

运行时间数量级的估计表达式包含一个表示任务大小的参数,这个任务通过算法(或者是程序,或者是代码段)进行求解。在我们所举的样例中,参数 $N$ 是待搜索数组的元素个数。毫无疑问,搜索一个具有大量元素的数组比搜索一个小的数组所花费的时间要长得多。大- $O$ 运行时间估计值的表达式总是一个关于问题规模的函数。在本章中,我们所有的算法将会和某个容器的值的范围相关。在任何情况下, $N$ 都表示这个范围内的元素个数。

下面是大- $O$ 估值的另外一种实用的、可替换的方式:

只关注估值表达式中的最高指数,不用注意表示倍乘关系的常量。

例如,下列所有估值表达式的大- $O$ 均为 $O(N^3)$ :

$$N^3 + 2N + 1, 3N^2 + 7, 100N^2 + N$$

而下面所有估值表达式的大- $O$ 均为 $O(N^3)$ :

$$N^3 + 5N^2 + N + 1, 8N^3 + 7, 100N^3 + 4N + 1$$

我们承认这些大- $O$ 运行时间估值有点粗略,但它们确实包含了一些信息。这种估值方式不会区分运行时间 $5N + 5$ 和运行时间 $100N$ 的区别,但它们的确实我们能够区分不同算法在运行时间的区别,继而能够判别出一些算法比另一些要快。我们看示例19.16所给出的图中的这些运行时间线,可以看到所有函数表达式的大- $O$ 为 $O(N)$ 的线最终都会逐渐位于函数 $0.5N^2$ 所代表的曲线的下面。这样的结果是必然的:假设 $N$ 的值足够大,那么 $O(N)$ 算法总是比 $O(N^2)$ 算法要快。尽管处理问题的规模不大时, $O(N^2)$ 算法可能会比 $O(N)$ 算法快,但事实上,编程者们已经发现对于大部分直观上看起来比较“大”的实际应用来说, $O(N)$ 算法总是要比 $O(N^2)$ 算法表现更好。对于示例19.16图中的其他两个不同的大- $O$ 运行时间来说,情况也是类似的。

#### 线性运行 时间

#### 平方运行 时间

一些专业术语会有助于我们描述泛型算法的运行时间。线性运行时间是指运行时间为 $T(N) = aN + b$ 的表达式。线性运行时间的大- $O$ 运行时间总是 $O(N)$ 。平方运行时间是指运行时间的高阶次数项为 $N^2$ 。平方运行时间的大- $O$ 运行时间总是 $O(N^2)$ 。有时运行时间公式也会出现对数的形式。这种情况下一般不会给出对数的底数,因为底数的变化只会引起一个常量倍乘因子的变化。如果你看到的是 $\log N$ ,那么你可以把它当作是底数为2的表达式,但如果你喜欢把它看成是底数为10的表达式也没有问题。对数函数是一种增长非常缓慢的函数。因此, $O(\log N)$ 运行时间非常快。

在许多情况下,我们估计的运行时间要比大- $O$ 估计时间更准确。特别是,当我们讨论一个线性运行时间,它是紧凑上界的,你可以认为确切的运行时间就是 $T(N) = cN$ ,尽管并没有指定 $c$ 。

#### 容器访问运行时间

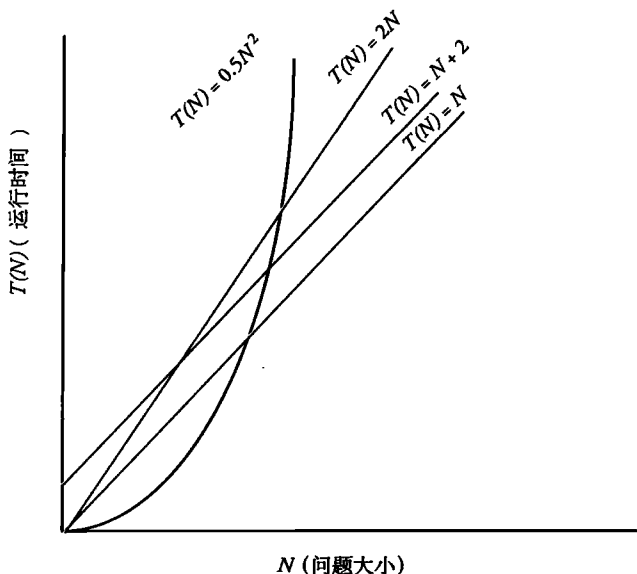
既然我们已经了解了大- $O$ 表示法,那我们就可以使用该方法来表示19.2节所讨论的那些容器类取值函数的效率。在`vector`的后面插入(`push_back`),在`deque`的

前面或者后面插入 (push\_back 和 push\_front), 在一个列表的任何地方进行插入 (insert), 所有这些操作都是  $O(1)$  (也就是说, 它是一个运行时间的常量上界, 和容器的大小无关)。在 vector 或者 deque 中插入或者删除任意元素的运行时间估价值是  $O(N)$ , 其中  $N$  是容器中元素的数目。对于一个 set 或者 map 的查找操作, (find) 的大- $O$  运行时间是  $O(\log N)$ , 其中  $N$  是容器中元素的数目。

### 自测练习题

18. 说明运行时间  $T(N) = aN + b$  的大- $O$  运行时间是  $O(N)$ 。(提示: 唯一问题是加  $b$ , 假设  $N$  至少为 1。)
19. 对于对数函数来说任意两个基数  $a$  和  $b$ , 如果  $a$  和  $b$  都是大于 1 的数, 那么存在一个常量  $c$ , 使得  $\log_a N \leq c (\log_b N)$ 。因此, 没必要在  $O(\log N)$  中指定基数。也就是说, 对于大- $O$  运行时间来说,  $O(\log_a N)$  和  $O(\log_b N)$  是一样的。

### 示例 19.16 运行时间比较



### 不改变序列的算法

本节讨论用于操作容器但又不会以任何方式对容器内容进行修改的模板函数。泛型函数 find 是一个很好的、简单的、典型的例子。

泛型函数 `find` 与 `set` 模板类的成员函数 `find` 类似，但却是不同的 `find` 函数。任何 STL 顺序容器类都可以使用泛型 `find` 函数。示例 19.17 给出了一个样例。该例展示了 STL 顺序容器类 `vector<char>` 使用泛型函数 `find` 的用法。如果我们把示例 19.17 中的所有 `vector<char>` 替换为 `list<char>`，或者我们把所有 `vector<char>` 替换为其他任何顺序容器类，我们会看到，泛型函数 `find` 在这些容器类中的执行方式都是相同的。这就是为什么函数被称为泛型的原因之一：对 `find` 函数进行定义后，它可以在很多种容器中执行。

### 示例 19.17 泛型函数 `find`

---

```

1  // 演示泛型函数 find 用法的程序。
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  using std::vector;
9  using std::find;

10 int main( )
11 {
12     vector<char> line;

13     cout << "Enter a line of text:\n";
14     char next;
15     cin.get(next);
16     while (next != '\n');
17     {
18         line.push_back(next);           如果 find 没有找到目标值，
19         cin.get(next);                 它返回第二个参数。
20     }

21     vector<char>::const_iterator where;
22     where = find(line.begin( ), line.end( ), 'e');
23     //where 定位于 v 中第一次出现 'e' 的地方。

24     vector<char>::const_iterator p;
25     cout << "You entered the following before you"
26         << "entered your first line:\n";
27     for (p = line.begin( ); p != where; p++)
28         cout << *p;
29     cout << endl;
30     cout << "You entered the following after that:\n";
31     for (p = where; p != line.end( ); p++)
32         cout << *p;
33     cout << endl;

34     cout << "End of demonstration.\n";
35     return 0;
36 }
```

## 示例运行结果1

```

Enter a line of text
A line of text.
You entered the following before you entered your first e:
A lin
You entered the following after that:
e of text.
End of demonstration.

```

## 示例运行结果2

```

Enter a line of text
I will not!
You entered the following before you entered your first e:
I will not! ←
You entered the following after that:
End of demonstration.

```

如果 find 没有找到目标值，它返回 line.end()。

如果 find 函数没有找到所要查找的元素，它返回该函数的第二个迭代器参数，这个参数不一定像示例 19.17 那样必须等于 end()。示例运行结果 2 给出了 find 函数没找到搜索目标时的程序运行结果。

那么，find 函数是否一定可以在所有容器类中正常工作呢？答案是否定的。要使用 find 函数，就必须把迭代器作为参数，但是诸如 stack 之类的一些容器本身并没有迭代器。要使用 find 函数，容器必须有迭代器，容器中的元素必须是线性顺序存储，这样的话 ++ 运算符才能遍历容器，而且必须可以对元素使用 == 运算符来进行比较判断。换句话说，容器必须有前向迭代器（或者必须有更强的迭代器，比如双向迭代器）。

在将要介绍的泛型函数模板中，我们使用泛型函数所需要的迭代器种类名作为函数的类型参数名，以这种方式来描述迭代器类型参数。因此，在实际调用泛型函数时，应该使用某种前向迭代器来替换 ForwardIterator，比如说使用 list、vector 或其他容器模板类所包含的迭代器来替换 ForwardIterator。记住，双向迭代器也是一个前向迭代器，而且随机访问迭代器也是一个双向迭代器。因此，不管是双向迭代器、随机访问迭代器还是一个普通老式的前向迭代器类型都可以使用类型名 ForwardIterator。在某些情况下，当指定迭代器种类为 ForwardIterator 时，我们甚至可以使用更为简单的迭代器种类——即输入迭代器或者输出迭代器，来替换 ForwardIterator。考虑到本书并没有对输入迭代器和输出迭代器进行专门介绍，因此我们不会在函数模板声明中提及这两种迭代器。

请记住，前向迭代器、双向迭代器和随机访问迭代器指的是迭代器的种类，而不是类型名。实际的类型名看起来类似于：std::vector<int>::iterator，此处的迭代器是一个随机访问迭代器。

示例 19.18 给出了一些 STL 中的不变泛型函数例子。示例 19.18 使用了一些讨论容器类迭代器的常用表示法（符号）。从迭代器 first 位置到迭代器 last 位置，但不包括迭代器 last，这个范围被称为一个区间（range）。例如，下列 for 循环输出所有在 [first,last) 区间中的元素：

区间

[first,last)



```
for (iterator p = first; p != last; p++)
    cout << *p << endl;
```

注意, 当给出两个区间时, 这两个区间不一定要在同一个容器中, 或者说甚至不需要在相同类型的容器中。例如, 对于 `search` 函数来说, 区间 `[first1, last1)` 和 `[first2, last2)` 可能在同一个或者不同的容器中。

注意示例 19.18 中含有三个搜索函数: `find`、`search` 和 `binary_search`。函数 `search` 搜索一个子序列, 而 `find` 和 `binary_search` 搜索单个值。那么, 当搜索单个元素时, 你是如何决定到底是用 `find` 还是 `binary_search`? 一个函数返回一个迭代器, 而另外一个只返回一个 `Boolean` 值, 但这还不是最大的区别。`binary_search` 函数要求被搜索的范围是排好序的(升序使用符号“<”), 而且它的运行时间为  $O(\log N)$ 。但是, `find` 函数不要求搜索的范围是排好序的, 但它只能保证运行时间仅仅能达到线性时间, 没有 `binary_search` 的运行时间快。如果你的元素本身是排好序的, 或者你可以让它们排好序, 那就请使用 `binary_search`, 这会使搜索变得更快。

### 区间 `[first, last)`

迭代器从位置 `first` (通常是 `container.begin()`) 移动到位置 `last` (通常是 `container.end()`), 但不包含位置 `last`, 这种操作非常普遍, 以致我们给它起了一个特殊的名字: 区间 `[first, last)`。例如, 下列代码输出区间 `[c.begin(), c.end())` 中的所有元素, 其中 `c` 是某个容器对象, 比如 `vector`:

```
for (iterator p = c.begin(); p != c.end(); p++)
    cout << *p << endl;
```

请注意在使用函数 `binary_search` 时, 它可以保障你的搜索程序是使用二分查找算法实现的, 我们在第 13 章中讨论过二分查找算法。使用二分查找算法的重要性在于, 它可以保障一个非常快的运行时间,  $O(\log N)$ 。如果你还没阅读第 13 章的内容, 没有了解过关于二分查找的相关知识, 那把它看作是一个对排好序的元素序列进行搜索的高效搜索算法即可。这是在二分查找法中仅有的与本章相关的内容。

### 示例 19.18 一些不变泛型函数

这些函数都适用于前向迭代器, 这意味着它们同样也适用于双向迭代器以及随机访问迭代器 (在一些应用场合, 它们甚至可以适用于其他种类的迭代器, 在本书中, 我们没有对这些迭代器展开详细介绍)。

```
template <class ForwardIterator, class T>
ForwardIterator find(ForwardIterator first, ForwardIterator last,
                    const T& target);
// 遍历区间 [first, last), 返回一个迭代器, 迭代器指向第一次出现搜索目标的元素位置。
// 如果没有找到待搜索的目标, 则返回第二个参数。
// 时间复杂度: 与区间 [first, last) 的元素个数成线性关系。

template <class ForwardIterator, class T>
```

```

int4 count(ForwardIterator first, ForwardIterator last, const T& target);
// 遍历区间 [first, last), 返回和搜索目标相等的元素个数。
// 时间复杂度: 与区间 [first, last) 的元素个数成线性关系。

template <class ForwardIterator1, class ForwardIterator2>
bool equal(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2);
// 从 first2 开始, 如果存在一个序列和区间 [first1, last1) 的序列相等, 那么返回 true,
// 否则, 返回 false。
// 时间复杂度: 与区间 [first1, last1) 的元素个数成线性关系。

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2);
// 检查区间 [first2, last2) 是否是区间 [first1, last1) 的子区间。
// 如果是, 它返回一个迭代器, 这个迭代器位于区间 [first1, last1) 中第一次匹配 [first2, last2)
// 的起始位置处。
// 如果不是, 则返回 last1。
// 时间复杂度: 和区间 [first1, last1) 的元素个数成平方关系。

template <class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& target);
// 前提条件: 区间 [first, last) 中的元素使用 "<" 按升序排列。
// 使用二分查找算法来判断是否搜索目标在区间 [first, last)。
// 时间复杂度: 对于随机访问迭代器, 它的复杂度是  $O(\log N)$ 。对于非随机访问迭代器, 时间复杂度与
//  $N$  的大小成线性关系, 其中  $N$  是区间 [first, last) 的大小。

```

## 自测练习题

20. 用标识符 `list` 替换示例 19.17 中出现的所有标识符 `vector`。编译并运行程序。
21. 假设 `v` 是类 `vector<int>` 的对象。使用泛型函数 `search` (示例 19.18) 来判定 `v` 中是否包含数字 42, 其后紧随数字 43。不必写出整个程序, 但要给出程序中所要使用到的 `include` 和 `using` 指令。(提示: 再使用一个 `vector` 可能会有帮助。)

## 改变序列的算法

示例 19.19 包含对 STL 中一些泛型算法的描述, 这些算法通过某种方式对容器的内容进行了修改。

记住, 增加一个元素到容器中或者从容器中删除一个元素都会对其他迭代器产生影响。除非容器模板类做了保证, 否则无法保证增加或者删除元素后迭代器还处于增加或删除之前的位置。在我们所看到的这些模板类中, `list` 和 `slist` 保证执行增加和删除操作后, 迭代器位置不变。当然, 如果迭代器位置碰巧处于被删除元素的位置, 这种情况除外。模板类 `vector` 和 `deque` 未对此做保障。示例 19.19 中的一些函数模板保证一些特定迭代器的值不变; 当然, 你可以相信这些保证对于任何容器都是适用的。

<sup>4</sup> 实际返回类型是我们没讨论过的整数类型, 但是返回值应当被赋值给一个 `int` 类型的变量。

### 示例 19.19 一些可变泛型函数

```
template <class T>
void swap(T& variable1, T& variable2);
// 交换变量1和变量2的值。
```

迭代器类型参数的名字告诉了函数可以正常工作的迭代器种类。需要记住这些是函数所需要的最低要求的迭代器。例如，ForwardIterator 适用于前向迭代器、双向迭代器和随机访问迭代器。

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2);
// last2 使得区间 [first1, last1) 和区间 [first2, last2) 中的元素个数相同。
// 行为：拷贝位于区间 [first1, last1) 中的所有元素到区间 [first2, last2) 中。
// 返回 last2。
// 时间复杂度：和区间 [first1, last1) 的元素个数成线性关系。
```

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                      const T& target);
// 将区间 [first, last) 中所有等于 target 的元素删除。
// 容器大小不变。等于 target 的那些被删除的元素被移到区间 [first, last) 的末尾。
// 那么，在这个区间内会有一个迭代器 i，使得所有元素值都不等于 target 的元素位于区间 [first,
// i) 中。
// 函数返回 i。
// 时间复杂度：和区间 [first1, last) 的元素个数成线性关系。
```

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
// 将区间 [first, last) 中所有元素的位置倒转。
// 时间复杂：和区间 [first, last) 的元素个数成线性关系。
```

```
template <class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first,
                   RandomAccessIterator last);
// 使用伪随机数生成器来将区间 [first, last) 中的元素随机地重新排列。
// 时间复杂度：和区间 [first1, last) 的元素个数成线性关系。
```

### 自测练习题

22. 模板函数 random\_shuffle 可以应用于 list 容器吗?
23. 模板函数 copy 要求前向迭代器，而 vector 具有随机访问迭代器。那么，模板函数 copy 可以应用于 vector 容器吗?

### 集合算法

示例 19.20 给出了一个在 STL 中定义的关于集合操作的泛型函数样例。注意，这个泛型算法假设容器里的元素是按顺序存储的。事实上，容器 set、map、multiset 和 multimap 确实是按顺序存储它们的元素；因此，示例 19.20 的所有函数都可以在这些模板类容器中正常工作。但是，诸如 vector 之类的一些其他容器，其内部元素

并不是按顺序存储的；这样的话，一些函数就不适用于这些模板类容器了。要求元素按顺序存储的原因是：对于按顺序存储的元素序列，这些算法会变得更加高效。

### 示例 19.20 集合操作

下面这些函数适用于 set、map、multiset 和 multimap（以及一些其他容器），但是并不是对所有容器都适用。例如，对于 vector、list 及 deque 这三个容器，这些函数不能正常工作（除非这三个容器内的元素是按顺序存储的）。对于这些函数来说，它们的适用条件是容器内部的元素必须按顺序排列。所有这些函数都适用于前向迭代器，这也意味着它们同样适用于双向迭代器和随机访问迭代器。（在一些应用案例中，它们也适用于我们未提到的一些其他种类的迭代器。）

```
template <class ForwardIterator1, class ForwardIterator2>
bool includes(ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2);
// 如果区间 [first2, last2) 中的所有元素也出现在区间 [first1, last1) 中，函数返回 true。
// 否则，函数返回 false。
// 时间复杂度：区间 [first1, last1) 中的元素个数加区间 [first2, last2) 中的元素个数总和成
// 线性关系。

template <class ForwardIterator1, class ForwardIterator2,
          class ForwardIterator3>
void5 set_union(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2,
               ForwardIterator3 result);
// 创建两个区间 [first1, last1) 和 [first2, last2) 的并集。新创建的并集中的元素是按顺序排列的。
// 新创建的并集元素的存储起始于 result 的位置。
// 时间复杂度：区间 [first1, last1) 中的元素个数加区间 [first2, last2) 中的元素个数总和成
// 线性关系。

template <class ForwardIterator1, class ForwardIterator2,
          class ForwardIterator3>
void5 set_intersection(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      ForwardIterator3 result);
// 创建两个区间 [first1, last1) 和 [first2, last2) 的交集。新创建的交集元素是按顺序排列的。
// 新创建的交集元素的存储起始于 result 的位置。
// 时间复杂度：与区间 [first1, last1) 中的元素个数加区间 [first2, last2) 中的元素个数总和成
// 线性关系。

template <class ForwardIterator1, class ForwardIterator2,
          class ForwardIterator3>
void5 set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    ForwardIterator3 result);
// 创建两个区间 [first1, last1) 和 [first2, last2) 的差集。新创建的差集中的元素是按顺序排列
// 的。新创建的差集的元素是那些在第一个区间中出现，而第二个区间没有出现的元素。
// 新创建的差集元素的存储起始于 result 的位置。
// 时间复杂度：区间 [first1, last1) 中的元素个数加区间 [first2, last2) 中的元素个数总和成线
// 性关系。
```

---

<sup>5</sup> 函数返回一个类型为 ForwardIterator3 的迭代器，但也可以把它看成是一个返回值为 void 的函数来使用。

## 自测练习题

24. 数学课中所说的集合并不要求集合内的元素是有序的，它有一个并集操作。为什么 `set_union` 模板函数要求容器中的元素是有序的呢？

## 排序算法

示例 19.21 给出了两个模板函数的声明以及文档：一个函数用来对一个区间的元素进行排序，另一个函数对两个排好序的区间的元素进行合并。注意，排序函数 `sort` 保证运行时间为  $O(N \log N)$ 。尽管这超出了本书的讨论范畴，但它说明我们不可能写出一个运行时间比  $O(N \log N)$  还快的排序函数。因此，这个函数保证排序算法尽可能的快，接近线性运行时间。

### 示例 19.21 一些泛型排序算法

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
// 对区间 [first, last) 中的元素按照升序排列。
// 时间复杂度:  $O(N \log N)$ , 其中  $N$  是区间 [first, last) 中的元素个数。

template <class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator3>
void merge(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           ForwardIterator3 result);
// 前提条件: 区间 [first1, last1) 和 [first2, last2) 中的元素都是有序的。
// 行为: 把两个区间合并成一个区间, 合并后的区间 [result, last3) 中的元素是有序的。
// 其中 last3 = result + (last1 - first1) + (last2 - first2)。
// 时间复杂度: 区间 [first1, last1) 中的元素个数加区间 [first2, last2) 中的元素个数总和成
// 线性关系。

排序使用 "<" 运算符, 因此必须对这个运算符进行定义。还有另外一种没有给出的排序函数版本, 这个
函数允许你指定排序方式。这里 "Sorted" 的意思是说元素是按照升序排序的。
```

## 本章小结

- 迭代器是一个广义的指针。可以用它来遍历容器中某个区间中的元素。迭代器一般都定义了运算符 `++`、`--` 及解引用运算符 `*`。
- 包含迭代器的容器类都有成员函数 `end()` 和 `begin()`，这两个函数返回迭代器值，通过返回值可以处理容器中的所有数据，如下例所示：

```
for (p = c.begin(); p != c.end(); p++)
    process *p // *p 是当前的数据项。
```

- 通常迭代器主要有以下几种。

前向迭代器：可以对迭代器进行 ++ 操作。

双向迭代器：既可以对迭代器进行 ++ 操作，也可以进行 -- 操作。

随机访问迭代器：可以对迭代器进行 ++ 操作、-- 操作以及随机访问操作。

- 对于一个常量迭代器 p，使用解引用运算符 \*p，会产生一个只读元素。而对于一个可迭代器 p，那就可以对 \*p 进行赋值操作。
- 双向迭代器包含反向迭代器，它可以使你的代码以反向顺序来遍历容器中的元素。
- 在 STL 中，主要的容器模板类是 list、vector 以及 deque，list 包含一个可变的双向迭代器；模板类 vector 和 deque 都包含一个可变随机访问迭代器。
- stack 和 queue 都是容器适配器类，这意味着它们构建在其他容器类之上。stack 类是一个后进先出的容器类。而 queue 是一个先进先出容器类。
- 为了使搜索算法更为高效，set、map、multiset 及 multimap 容器模板类中的元素是有序的。set 是一个简单的元素集合。map 允许根据键的值进行排序和获取元素值。multiset 类允许有重复的数据项。multimap 类允许一个键与多个数据项关联。
- STL 提供了一些实现了泛型算法的模板函数，这些泛型算法都有一个最大运行时间保证。

## 自测练习题答案

1. v.begin( ) 返回一个指向 v 中第一个元素的迭代器。v.end( ) 返回一个值，这个值处于 v 中所有元素的最后一个，充当结束标志。
2. \*p 是应用于 p 的一个解引用操作。\*p 是一个索引，它指向位于 p 的元素。
3. 

```
using std::vector<int>::iterator;
...
iterator p;
for (p = v.begin( ), p++; p != v.end( ); p++)
    cout << *p << " ";
```
4. D C C
5. B C
6. 两种方式都可以正常工作。
7. 主要的区别是 vector 容器包含随机访问迭代器，而 list 只包含双向迭代器。
8. 除了 slist 外，都包含。
9. vector 和 deque。
10. 它们都可以有可迭代器。
11. stack（栈）模板适配器类没有迭代器。
12. queue（队列）模板适配器类没有迭代器。

13. 不会返回任何值。pop 函数的返回类型是 void。
14. 为了便于对元素进行高效搜索。
15. 是的，它们的元素可以是任意类型，尽管每个对象只有一个类型。模板类中的类型参数就是它所存储的元素的数据类型。
16. 如果 'A' 在 s 中，那么 s.find('A') 返回一个指向元素 'A' 的迭代器。如果 'A' 不在 s 中，那么 s.find('A') 返回 s.end()。
17. mymap 将包含两个数据项。一个是从 5 指向 "c++" 的映射，另一个是从 4 指向默认字符串的映射，默认字符串是一个空格。
18. 只需注意只要  $1 \leq N$ ，那么  $aN + b \leq (a + b)N$ 。
19. 下面的讨论是基于数学的概念，而不是 C++。因此，“=” 表示等于，而不是赋值。首先要注意的是  $\log_a N = (\log_a b)(\log_b N)$ 。  
要证明上述等式成立。只要我们能注意到，如果对 a 求  $\log_a N$  的次方，那么，它的值是 N。而对 a 求它的  $(\log_a b)(\log_b N)$  次方，它的值也为 N。  
如果令  $c = \log_a b$ ，那么将会得到  $\log_a N = c(\log_b N)$ 。
20. 程序的运行结果应该完全一致。
21. 

```
#include <iostream>
#include <vector>
#include <algorithm>
using std::cout;
using std::vector;
using std::search;
...
vector<int> target;
target.push_back(42);
target.push_back(43);
vector<int>::const_iterator result = search(v.begin(), v.end(),
                                           target.begin(), target.end());
if (result != v.end())
    cout << "Found 42, 43.\n";
else
    cout << "42, 43 not there.\n";
```
22. 不能。你必须有一个随机访问迭代器，只有 list 模板类才有双向迭代器。
23. 可以，一个随机访问迭代器也是一个前向迭代器。
24. set\_union 模板函数要求容器中的元素按顺序存储，这么要求的目的是为了更为高效的方式来实现函数模板。

## 编程练习

1. 这道编程练习题的重点是向大家演示：如果一个对象的行为像一个迭代器，那么它就是一个迭代器。更加准确地说，如果一个对象访问某个容器，而且它的行为在某种程度上像是一个迭代器，那么就可以把这个对象当成是一个迭代器，继而利用适用于这种迭代器的泛型函数来对容器进行操作。但是，尽管可以对

容器使用这些泛型算法，一些诸如 `begin` 和 `end` 之类的成员函数将不会在容器中出现了（例如，在一个数组中），除非在容器中已经显式定义了这些成员函数。这道编程练习还是仅针对 `double` 类型的数组，但是对于其他基本类型情况也是一样的。

- a. 根据随机访问迭代器的特性，说明一个指向数组元素的指针，其行为方式和一个随机访问迭代器的行为方式完全相同。
  - b. 进一步说明
    - i) 数组名是一个 `double` 类型的指针，它指向数组的第一个元素，因此，数组名可以充当“`begin`”迭代器，而且
    - ii) (数组名) + (数组大小) 可以作为“`end`”指针（当然，这个指针指向数组最后一个元素之后的位置）。
  - c. 写一个小程序，在这个程序中声明一个 `double` 类型的大小为 10 的数组，用 10 个 `double` 类型的数来填充这个数组。接着，调用 `sort` 泛型算法，算法的输入参数为这些指针值，显示程序的执行结果。
2. 这个问题想说明如何使用 `remove` 泛型函数对一个容器中的几个特定数据项的实例进行删除操作。另一个附带的作用是检查编译器处理泛型函数 `remove` 时的行为（我们已经看到 `remove` 的行为随着编译器的不同而会产生不同的变化）。在开始编程练习之前，先在你喜欢的 STL 文档或者 Web 站点上查询 `remove` 泛型函数的行为（例如，通过浏览器访问站点 <http://www.sgi.com/tech/stl/remove.html>。这个网站自发布之日起一直有效）。
- a. 修改前面编程练习 1 中的数组声明，让它包含几个相同的值（比如说，4.0，但是在本练习题中也可以使用其他任何值）。确保不是所有值都相同。使用可变泛型函数 `remove`（见示例 19.19）来测试删除所有值为 4.0 的元素（也就是，在构建数组时填写的那几个值相同的元素）。
  - b. 使用 a 部分提供的 `double` 类型数组来构建和该数组具有相同元素的 `list` 和 `vector` 容器。使用包含两个迭代器参数的容器构造函数来创建这两个容器。不管是 `vector` 还是 `list` 都有一个包含两个迭代器参数的构造函数，它对 `vector` 或 `list` 中两个迭代器区间内的元素进行初始化。这里的迭代器可以定位于任何容器，当然也包括数组。在构建 `vector` 和 `list` 时，使用数组名作为 `begin` 迭代器，使用数组名和数组长度作为 `end` 迭代器，`begin` 和 `end` 迭代器作为 `vector` 和 `list` 构造函数的输入参数。使用可变泛型算法 `remove`（示例 19.19）来从 `vector` 和 `list` 中删除所有值等于 4.0 的元素（或者删除在构建数组时输入的那些值相同的元素）。显示 `vector` 和 `list` 的内容，并解释程序最终的运行结果。
  - c. 修改上述 b 部分的代码，将泛型函数 `remove` 返回的迭代器值分配给一个适当类型的迭代器变量。参考 `remove` 函数的相关文档，明确这个返回的迭



代器值的意义。输出数组、vector 和 list 的内容，即从 `begin()` 到 `end()`，使用我们在前面所讨论的“begin”和“end”来输出数组的内容。输出两个容器 vector 和 list 中的一段内容，该段内容从 `remove` 函数所返回的迭代器所处位置开始到容器的 `end()` 位置处结束。

3. 质数是一个大于 1，而且只能被它自己和 1 整除的整数。如果存在一个整数  $z$ ，使得  $x = y * z$ ，那么我们说整数  $x$  可以被整数  $y$  整除。希腊数学家 Erathosthenes 给出了一个找到小于某个整数  $N$  的所有质数的算法。这个算法被称为 Erathosthenes 算法。它的工作方式如下：输入一个从整数 2 到  $N$  的列表。数字 2 是第一个质数（考虑一下为什么是这样，这是有益的）。所有和 2 有倍乘关系的数字如 4、6、8 等都不是质数。我们把这些数从列表中删除。接着，2 之后的第一个未被删除的数是 3，它是第二个质数。所有和 3 有倍乘关系的数都不是质数，从列表中删除这些数。注意，6 已经被删除了，9 和 12 也已经离开队列了，还有 15 等。剩下的没有被删除的第一个数是接下来的一个质数。算法以这种方式继续运行，直到达到最后一个数  $N$ 。队列中剩下的所有未被删除的数都是质数。

- a. 根据这个算法编写一个程序来找出小于用户所指定的整数  $N$  的所有质数。使用 vector 容器来存储这些整数。使用一个布尔类型的数组来追踪所有被删除的元素，这个数组的所有元素初始值都设置为 true。当元素被删除后，把数组元素的值由 true 改为 false。
- b. 令  $N$  分别等于 10、30、100 和 300，对所编写的程序进行测试。

进一步提升：

- c. 实际上，我们不需要对所有值进行测试。我们可以在  $N/2$  处终止测试。尝试这种方式，并测试你所编写的程序。在  $N/2$  处终止测试是可以的，程序会更高效，但它不是我们所能使用的最小终止条件。请说明，为了得到介于 1 和  $N$  之间的所有质数，最小的终止条件是  $N$  的平方根。
  - d. 修改 a 部分的代码，使用  $N$  的平方根作为测试的上限值。
4. 假设你有一个关于学生记录的集合。记录的数据类型用一个结构体来表示，如下所示：

```

        StudentInfo
    {
        string name;
        grade;
    };

```

记录存储在一个向量 `vector<StudentInfo>` 中。编写一个程序提示输入数据，获得输入的数据后构建一个学生记录的向量，接着根据名字对向量中的记录进行排序，计算出班级中的最高和最低分数以及平均分数，最后，将这些汇总数据、班级名册以及分数都打印出来（我们不关心谁的分数最高、谁的分数

最低，我们只关注全班的最高分、最低分以及平均分这几个统计值)。测试你所编写的程序。

5. 继续编程练习 4，编写一个函数把 `StudentInfo` 向量拆分成两个向量，一个包含考试通过的学生，一个包含未通过的学生（使用 60 分或者 60 分以上作为考试是否通过的标准）。

要求使用两种方法来编写上述程序，并给出运行时间估计。

- a. 继续使用所创建的这个向量。重新生成第二个向量用于存储那些通过考试的学生记录。使用第三个向量用于存储那些考试未通过的学生记录。这么做会产生一些重复记录，因此不要采取这种方式。可以创建一个向量来存储考试未通过的学生记录。接着，使用 `push_back` 把未通过考试的学生记录增加到考试未通过学生记录向量中，最后，使用 `erase`（它是一个成员函数）从初始的向量中删除考试未通过学生记录。使用这种方法编写你的程序。
  - b. 考虑方案的高效性。你很可能从向量的中间删除  $O(N)$  个记录，而且在这种情况下，必须移动许多记录。从向量中间删除元素需要花费  $O(N)$  次操作。给出这个程序的大  $O$  运行时间估值。
  - c. 如果你使用一个链表 `list<StudentInfo>` 来存储学生记录，那么删除和插入函数的运行时间是多少？考虑在链表中执行删除操作的时间效率对整个程序的运行时间会有什么样的影响。代替向量，使用链表来重新编写程序。记住，链表不会提供索引，也不会提供随机访问。
6. a. 下面是一段伪代码，表示从用户端输入一个值  $n$ ，接着插入  $n$  个随机数字，保证没有重复值：

```
Input n from user
Create vector v of type
Loop i = 1 to n
    r = random integer between 0 and n-1
    Linearly search through v for value r
    if r is not in vector v then add r to the end of v
End Loop
Print out number of elements added to v
```

使用你自己编写的线性搜索函数来实现程序，增加一个封装程序的代码来记录程序运行了多长时间。输入不同的  $n$  值来对程序进行测试。可能你需要输入一个较大的  $n$  值来使程序的运行时间至少超过 1 秒，当然，运行时间也取决于你所使用的系统。这是一个样例程序，表明如何计算时间差（`time.h` 是一个库文件，你的 C++ 版本应该支持这个头文件）。

```
#include <time.h>

time_t start, end;
... dif;

time (&start); // 记录起始时间
// 程序剩余部分在此
```

```
time (&end); // 记录终止时间
dif = difftime(end,start);
cout << "It took " << dif << " seconds to execute. " << endl;
```

- b. 接下来，创建第二个程序，它和第一个程序几乎完全一样，除了它使用的是 STL 中的 `set` 来存储元素，而不是使用向量 `vector` 存储：

```
Input n from user
Create set s of type
Loop i = 1 to n
    r = random integer between 0 to n-1
    Use s.find(r) to search if r is already in the set
    if r is not in set s then add r to s
End Loop
Print out number of elements added to s
```

用向量版本的程序中的  $n$  来记录程序运行时间。对 `vector` 中的线性搜索和 `set` 中的 `find()` 函数的大  $O$  运行时间进行比较，看看结果是什么？注意，`find()` 函数实际上是多余的，因为如果元素已经在集合 `set` 中了，插入不会产生影响。但是，无论如何，使用 `find()` 函数所创建的程序可以和 `vector` 算法进行比较。

7. 修改你在编程练习 6 中的 a 部分所编写的程序，使得泛型函数 `find` 可以用来在向量中搜索一个已经存在的值。你可以用一些样本数据来测试你的程序，确保程序能够正常工作。
8. 信息检索领域主要涉及根据一个查询来查找相关的电子文档。例如，给出一组关键值，一个搜索引擎获取到一些网页（文档），并根据文档的相关性，把相关性强的文档放到前面，按顺序显示这些文档。这项技术需要一种方法来对输入的查询和检索的文档进行比较，找出对查询来说哪个是最相关的文档。

比较相关性的一种简单方法是计算二元余弦系数。这个系数是位于 0 和 1 之间的值，其中 1 表示查询与文档非常相似，0 表示文档中没有与查询关键字相同的内容。这种方法把每个文档看成是很多词的集合。例如，考虑下列样本文档：

“Cows are big. Cows go moo. I love cows.”

这个文档将被解析为多个关键字的集合，忽略词的大小写，并去掉标点符号，所解析的集合包含下列单词：“{cows, are, big, go, moo, i, love}”。在查询时，程序会执行相同的过程。

如果我们把查询  $Q$  表示为一个单词集合，把一个文档  $D$  表示为一个单词集合，查询和文档的相关性可以通过下列表达式来计算：

$$Sim = \frac{|Q \cap D|}{\sqrt{|Q|} \sqrt{|D|}}$$

例如，如果  $D = \{\text{cows, are, big, go, moo, i, love}\}$ ， $Q = \{\text{love, holstein, cows}\}$ ，那么，

$$Sim = \frac{|\{\text{love, cows}\}|}{\sqrt{|Q|} \sqrt{|D|}} = \frac{2}{\sqrt{3} \sqrt{7}} = 0.436$$

编写一个程序，允许用户输入一个表示文档的字符串集合，输入一个表示查询的字符串集合（如果你还有更大的雄心，可以写一个程序来解析实际的文本文件，计算出唯一字符串集合）。用 STL 字符串集合 `set` 来表示文档和查询。接着，使用二元余弦系数公式计算出查询和文档的相似性，打印出结果。`sqrt` 函数在 `cmath` 中。使用泛型函数 `set_intersection` 计算  $Q$  和  $D$  的交集。

下面是一个 `set_intersection` 的例子，用来对集合  $A$  和  $B$  求交集，并把结果存储在集合  $C$  中，其中所有集合都是字符串集合：

```
#include <iterator>
#include <algorithm>
#include <set>
#include <string>
...
using std::insert_iterator;

set<string> A,B,C;
// 下面的代码假设字符串已经被插入到了集合 A 和 B 中。
// 注意下面这行中，>> 之间的空格。
insert_iterator<set<string> > cIterator(C, C.begin( ));
set_intersection(A.begin( ), A.end( ),
                 B.begin( ), B.end( ),
                 cIterator);
// 集合 C 现在包含集合 A 和 B 的交集。
```

9. 重做第 5 章中的编程练习 8，当然，也有可能你是第一次做这道题目。这道编程练习题请你通过程序模拟，估计出在同一个房间中存在两个或更多个人具有相同生日的概率，对同一房间的 2 ~ 50 个人进行估计。

但是，请不要使用数组作为程序的编码方案，请使用映射（`map`）作为编码方案。通过许多次试验（比如说，5000 次），随机地为房间中的每个人分配生日（也就是说，从数字 1 到数字 365，假设每个数字都有相同的概率）。使用 `map<int, int>` 映射生日（1 ~ 365）来统计每个生日出现的次数。每个生日所映射的初始值为 0。当随机生成一个生日时，在 `map` 中增加该生日对应的统计次数。如果发现某个生日出现了重复，那么增加一次试验次数。做完所有试验后，这个统计次数应当能表明出现重复生日的试验次数。用这个统计次数除以总的试验次数就能够计算出在一个给定大小的房间内，两个或更多人具有相同生日的估计概率。程序输出结果应该和第 5 章编程练习 8 中的结果看起来是相同的。

10. 假设你已经收集了一个对电影进行评论的文件，评分值为 1（很差的电影）~ 5（很好的电影）。文件的第一行是一个数字，它表示该文件内一共有多少个评价。每个评价包含两行：电影名以及 1 ~ 5 的一个评分值。下面是一个评价样本文件，它包含四个电影、七条评论：

```
7
Happy Feet
4
```

```

Happy Feet
5
Pirates of the Caribbean
3
Happy Feet
4
Pirates of the Caribbean
4
Flags of Our Fathers
5
Gigli
1

```

编写一个程序，它以样本的格式读取文件，计算出每个电影的平均评分值，输出每个电影的平均评分值以及评论个数。程序对样本的输出结果如下所示：

```

Happy Feet: 3 reviews, average of 4.3 / 5
Pirates of the Caribbean: 2 reviews, average of 3.5 / 5
Flags of Our Father: 1 review, average of 5 / 5
Gigli: 1 review, average of 1 / 5

```

使用一个 `map` 或者多个 `map` 来产生输出。`map` 应该以表示每个电影名的字符串作为索引，映射到一个整数对上，其中一个整数用于表示某个电影的评论数，另一个整数表示这个电影的评论总分值。

11. 编写一个程序输出一个班级学生成绩的直方图。程序应当输入一个整数来表示每个学生的成绩，并把这些成绩存入到一个向量中。当用户输入 -1 时，表示成绩输入结束。使用一个从 `int` 到 `int` 的 `map` 来计算这个直方图。其中，`map` 中的第一个整数表示一个成绩，第二个整数表示这个成绩出现的次数。把这个直方图输出到屏幕上。参考第5章编程练习7来获取如何计算直方图的相关信息。本编程练习题不应该对成绩的最小值和最大值进行限制。

12. 考虑一个关于名字的文本文件，在这个文件中每个名字一行。下面是一个样本文件：

```

Brooke Trout
Dinah Soars
Jed Dye
Brooke Trout
Jed Dye
Paige Turner

```

在这个文件中，有重复的名字。我们想生成一个邀请表，但是不希望给同一个人多次发送邀请。通过使用集合模板类来消除重名现象。从文件中读取每个名字，把它添加到集合中，接着，输出集合中的所有名字来生成一个没有重名的邀请列表。

13. 逆波兰表示法（RPN）或者后缀表示法是一种数学表达式格式。在 RPN 中，运算符出现于操作数之后，它不像一般的方式，运算符介于两个操作数之间（这种表示法被称为中缀表示法）。可以使用一个空栈，利用如下规则来实现一个 RPN 计算器：

- 如果输入的是一个数字，把它压入栈中。
- 如果输入的是 +，那么把最后两个操作数弹出栈，对它们进行相加，并把结果压入到栈中。
- 如果输入的是 -，那么把 value1 弹出栈，再把 value2 弹出栈，然后把 value2-value1 压入到栈中。
- 如果输入的是 \*，那么把最后两个操作数弹出栈，对它们进行相乘，并把结果压入到栈中。
- 如果输入的是 /，那么把 value1 弹出栈，再把 value2 弹出，最后把 value2/value1 压入到栈中。
- 如果输入的是 q，那么停止输入值，打印栈顶元素，并退出程序。

使用栈模板类来实现 RPN 计算器。当给出了一个运算符，而栈中却没有两个操作数时，输出一个恰当的错误提示消息。这是一个输入样本以及结果的输出，它等价于表达式  $((10 - (2 + 3)) * 2) / 5$ ：

```
10
2
3
+
-
2
*
5
/
q
```

The top of the stack is: 2