

Practical C++ Programming

实用C++ 编程大全



Steve Oualline 著

O'REILLY™



幸运伟 吴拉朵 译

电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

实用C++编程大全

Practical C++ Programming

[美] Steve Oualline 著

幸运伟 吴拉朵 译



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

内 容 简 介

本书是实用C++编程大全，是为学习和使用C++的读者而编写的。

本书内容共有五部分，共二十九章，既介绍C++的基本语法，又介绍面向对象程序设计中的新概念、新技术、新方法。内容深入浅出，语言流畅通顺，同时配有大量的示例程序供读者参考。

本书着重体现实用特性，书中提供的大量实例可以帮助读者尽快熟悉面向对象的程序设计方法，是编程人员必备的参考书。

Chinese Edition Copyright ©1997 by Publishing House of Electronics Industry. Authorized Translation of the English Edition Copyright © 1995 by O'Reilly & Associates, Inc.

This translation is published and sold by permission of O'Reilly & Associates, Inc. The owner of all rights to publish and sell the same.

All rights reserved including the right of reproduction in whole or in part in any form.

本书中文专有翻译出版权由美国O'Reilly & Associates, Inc 公司授予电子工业出版社。未经许可，不得以任何手段和形式复制或抄袭本书内容。版权所有，侵权必究。

原 书 名: Practical C++ Programming

书 名: 实用C++编程大全

著 者: [美]Steve Oualline

译 者: 辛运伟 吴拉朵

责任编辑: 潘玉颖

排版制作: 周银玉

印 刷 者: 北京市牛山世兴印刷厂印刷

出版发行: 电子工业出版社出版、发行

北京市海淀区万寿路173信箱 邮编100036 发行部电话 68214070

URL: <http://www.phei.co.cn>

经 销: 各地新华书店经销

开 本: 787×1092 1/16 印张: 34 字数: 827千字

版 次: 1997年8月第1版 1997年8月第1次印刷

印 数: 3300册

书 号: ISBN 7-5053-4237-1
TP·1908

定 价: 53.00元

著作权合同登记号 图字: 01-97-0254

凡购买电子工业出版社的图书，如有缺页、倒页、脱页者，本社发行部负责调换
版权所有·翻印必究

中文版序

这本书在中国由电子工业出版社出版，我感到特别骄傲。我极荣幸地被邀请为本书的中文版作序。

一百多年前，美国决定修建一条横贯东西的铁路大动脉。但在铁路的西端萨克拉门托市出了点问题。雇佣的美国工人，在骑马到达工作地点后，相继潜逃，作他的淘金梦去了。

铁路工程的总负责人斯坦福先生向工程监督蒙塔古先生建议，可以从旧金山的华人聚居地雇佣华人民工。

“中国人？他们那么矮小，能完成这么繁重的工作吗？”他表示怀疑。

“当然，他们建设了中国的长城。”回答是肯定的。

长久以来，中国人就以能够建造长城及宏大的建筑、城市而闻名。这些不仅仅是建筑物，它们还是精美的艺术品，凝聚了工匠们的心血。工匠们用成百上千的砖，靠熟练的技能，创造出精美的传世之作。

言归正传。一个好的程序也是一件艺术品，它由成百上千行、乃至百万行的代码组成。

遗憾的是，并不是所有的程序都是完美的。世上还有许许多多不太尽善尽美的代码存在。现在，我的工作就是维护它们。

本书专为想编写优美代码的读者而著。通过本书的学习，编程就变成了一项既慎密又有趣味的艺术工作，再也不是枯燥乏味、令人困惑的麻烦事。

感谢本书的出版商中国电子工业出版社，感谢本书的中文翻译辛运伟女士、吴拉朵先生，感谢本书的责任编辑潘玉颖小姐，感谢本书的美国编辑Andy Oram小姐。没有他们的努力，本书便不可能与读者见面。本书的出版得益于他们无私的奉献。

最后，我还要感谢你，购买本书的尊敬的读者。程序员的工作并不是一帆风顺的，希望本书能助您一臂之力，编制出更完善的程序。

Steve Oualline

1997年6月

译者序

C++是目前最流行的程序设计语言之一，它在C语言的基础上，增加了类的概念，使用全新的面向对象的方法来设计程序。所以有人把它称为“带类的C”。为了帮助广大程序设计人员从C语言尽快过渡到C++语言，熟练掌握这一有力的编程工具，我们特向大家推荐这本在美国非常畅销的书。

学习一种计算机编程语言，掌握它的语法只是第一步，更重要的是要灵活地运用它，掌握正确的编程方法，体现良好的编程风格。从某种意义上来说，编程是一门艺术。和艺术大师们各具风采一样，程序设计人员也各有各的风格。好的程序员能够设计出正确、简洁、高效、易懂、流畅的程序，他们的程序成为其他人争相效仿的经典之作。虽然我们并不可能都成为程序设计大师，但是程序设计风格也是影响我们所设计的程序质量的重要因素。正如书中所说的：编码技巧与程序设计技巧相结合才能产生好的程序。

本书的作者Steve Oualline先生是美国著名的C++专家，他有多年的编程之丰富经验。他以清晰的思路、简洁优美的语言写就了这本书。Steve Oualline先生在百忙之中，欣然应允为本书作序，这使我们非常感动。感谢他对我们的帮助，感谢他为我们带来的这本好书。

本书确可称作实用C++编程大全，它不仅向你讲授语言技巧，还讲授编程的风格以及程序的调试。本书介绍了程序的整个编制过程，包括概念、设计、书写、调试、交付、文档、维护以及修正，内容全面，深入浅出，并有大量的例题贯穿其中，为读者的学习提供了最大的方便。本书在讲授语法的同时，着重于编程风格的培养，这是本书有别于其它书的地方。良好的设计风格是日积月累的结果，但愿本书能在这方面对您有所帮助。

本书由辛运伟、吴拉朵翻译。另外，还有许多同志为本书的顺利出版付出了艰辛的努力，在此一并表示感谢。这些默默无闻的幕后英雄是：本书的责任编辑潘玉颖，以及周银玉、刘丰、许宁、孟辉、李品对、由鑫、张励、郭震海、孟淑杰、贾德燕。没有他们的积极参与和鼎力相助，就不会有本书。

由于译者水平有限，译文中难免有错误之处，敬请广大读者批评指正。

译者

1997年6月

前言

本书是实用C++编程大全。它不仅向你讲授语言技巧，还讲授编程的风格以及程序的调试。本书将介绍程序的整个编制过程，包括概念、设计、书写、调试、交付、文档、维护以及修正。

我们强调编程风格。编写一个好的程序不仅仅是键入代码，它是一门艺术。编码技巧与程序设计技巧相结合才能产生好的程序。好的程序不仅要求功能正确，还应该是简单的，并且易于理解。程序员可以把他们对程序的描述作为注释放在程序中。书写清楚、注释明了的程序才是好程序。

一个程序应该尽可能地简单。应避免使用技巧。这些技巧和复杂性只会扼杀程序。本书强调简单实用这条原则。例如，C++中的15条运算符优先级规则可以简化为下面的2条：

1. 乘除法优先于加减法。
2. 括号优先于其它一切运算。

考虑两个程序。一个是由一位聪明的程序员编写的，其中使用了所有的技巧。程序中没有注释，但它能运行。另一个有很好的注释，并且结构很好，但不能运行。哪个程序更有用？从长远的观点来看，“失败”的程序更有用，因为它可以被修改，维护也容易。虽然前一个程序现在可以运行，但早晚它要被修改。最艰巨的工作就是不得不修改一个充满技巧的程序。

本书的读者对象

本书是为下列人员准备的：没有任何编程经验的读者、了解C语言并想再进一步学习C++的程序员、初步了解C++并想改进编程风格和可靠性的读者。读者应该接触过计算机，并知道基本的使用方法，比如会使用文本编辑器和文件系统。

通过书写及调试程序学习计算机语言是最佳途径。辛辛苦苦地调试一个运行失败的程序，到半夜两点才发现原来在应该键入==的地方键入了=，这就是一个很有说服力的例证。本书自始至终有许多示例，其中的大多数故意留有一些小错误。我们鼓励读者把这些示例输入到计算机中，运行并调试它们。我们在一些较短的程序中向读者介绍常见的错误，当读者编制自己较大的程序时，就知道如何发现并修正这样的错误了。（本章最后将介绍获取本书所提供程序拷贝的方法。）

下面是C++中的一些约定：

- “通用的”UNIX编译器是指能在大多数UNIX系统中使用的编译器
- GNU C++编译器名为g++（可在大多数UNIX系统中使用*）
- Borland的Turbo C++编译器用于MS-DOS环境
- Borland C++用于MS-DOS/Windows环境
- Microsoft的Visual C++用于MS-DOS/Windows环境

不同版本的编译器编译标准C++时没有多少差别。凡是涉及到因编译器不同而影响程序员编程的地方，本书会明确指出。我们会罗列出使用每个编译器建立并运行程序所需的指令。本书还给出使用编程实用工具make的示例，这些工具有助于程序的自动生成。

本书的组织方式

走之前先要学会爬。第一部分：基础，在这一部分中你将学习如何爬。本章只讲授如何编制一个非常简单的程序。读者从编程技巧和编程风格入手。下一步，将学习如何使用变量和非常简单的判定及控制语句。

这时你学过了如何编制非常简单的程序，在第七章“程序设计过程”中，我们才将编制真正程序的完整过程展示给读者。

第1章“什么是C++”，向读者介绍C++的概况，叙述它的发展历史和使用方法，解释语言是如何组织的。

第2章“编程基础”，解释基本的编程过程，读者据此可以编写非常简单的程序。

第3章“风格”，讨论编程风格。介绍如何为一个程序加注释，以及如何书写清楚简单的代码。

第4章“基本说明和表达式”，介绍简单的C++语句。本章详细介绍基本变量和赋值语句，另外还将介绍算术运算符：+、-、*、/和%。

第5章“数组、修饰符和读数”，介绍数组和更复杂的变量。本章还将介绍几个快捷运算符：++、--、*=、/=、+=、-=和%=。

第6章“判定和控制语句”，介绍简单的判定语句，包括if、else和for。还将讨论==与=的区别。

第7章“程序设计过程”，介绍编制一个简单程序的各个步骤，从程序的规划到交付。

* GNU g++编译器可从匿名FTP的prep.ai.mit.edu得到，也可以与Free Software Foundation公司联系，该公司地址为675 Massachusetts Avenue, Cambridge, MA 02139，电话为(617)876-3296。

另外还将介绍结构化程序设计方法、快速原型程序设计方法和程序的调试。

第二部分：简单的程序设计，介绍编程时所用的其它所有的简单语句和运算符。读者还将学习如何把这些语句组织为简单函数的方法。

第8章“再次讨论控制语句”，介绍另外一些控制语句。包括while、break和continue。除此之外，还将介绍switch语句。

第9章“变量作用域和函数”，介绍局部变量、函数和参数。

第10章“C++预处理器”，介绍C++预处理器。本章介绍的内容为你的编程提供了更大的灵活性；另外，还将为读者介绍各种各样的方法，以使读者能够把C++的各个部分组合在一起。使用预处理器所要遵从的有关规则也在本章中加以叙述。

第11章“位运算”，介绍对位进行运算的C++逻辑运算符。

第三部分：高级类型和类，读者将学习如何用基本说明和语句来构造高级类型的方法，如构造结构、联合和类。这一部分中还将向读者介绍指针的概念。

第12章“高级类型”，解释结构和其它高级类型。本章还将介绍sizeof运算符和enum类型。

第13章“类的初步讨论”，介绍类的概念。这是C++最强大的特色之一。类允许你把数据和对数据施加的操作组织为一个对象。

第14章“类的进一步讨论”，介绍可以对类施加的操作。

第15章“简单指针”，介绍C++指针变量以及它们的一些用法。

第四部分：高级编程概念，研究高级编程技术。在这一部分，我们研究C++的一些特色。运用这些特色，读者可以创建复杂但易用的对象或是类。

第16章“文件输入/输出”，介绍缓冲的和非缓冲的输入/输出（I/O）。我们还将讨论ASCII码文件和二进制文件，另外给出建立一个简单文件的方法。这一章中还将介绍原来C风格的I/O操作。

第17章“调试和优化”，介绍如何调试一个程序及使用交互式调试器的方法。在这一章中，不仅介绍调试程序的方法，还介绍书写程序的方法，这样，调试起来更容易一些。本章还将介绍许多优化技术，借助于这些技术，可以让程序运行得更快也更高效。

第18章“运算符重载”，C++允许你把它的运算符定义为另外的意思，从而扩充了语言。在这一章，读者将定义一个复数类型，并定义对复数的运算。

第19章“浮点数”，使用简单的十进制浮点数格式，介绍浮点数的一些内在问题，如舍入误差、精度丢失、上溢和下溢。

第20章“高级指针”，介绍指针的高级使用方法。使用这些方法，可以构造动态数据类型，如链表和树。

第21章“高级类”，介绍如何从简单的基类派生出复杂类的方法。

最后，在第五部分语言的其它特性中，我们将介绍其余的所有特性。

第22章“例外”，介绍如何处理程序中意想不到的问题。

第23章“模块化程序设计”，介绍如何把一个程序划分为几个文件，以及使用模块化技术设计程序时遇到的问题。还将详细讨论make实用工具。

第24章“模板”，借助于模板，读者可以定义一个通用函数或是生成一系列函数的类。

第25章“移植问题”，介绍程序移植（把一个程序从一种机器移植到另一种机器）时可能遇到的问题。

第26章“组合在一起”，详细讨论完成一个复杂程序的各个步骤，从概念到完成。还将介绍信息隐藏和模块化技术，以及面向对象的程序设计方法。

第27章“从C到C++”，介绍如何把C代码转化为C++代码，同时论述C代码中暗藏的困扰C++程序员的一些问题。

第28章“C++内的‘角落’”，介绍do/while语句、逗号运算符和?:运算符。

第29章“程序设计格言”，列出设计程序时要注意的问题，这些有助于读者编写好的C++程序。

附录A“ASCII码表”，列出了字符代码及对应的值。

附录B“范围”，列出一些C++变量类型的数值表示范围。

附录C“运算符优先级规则”，列出规定运算符运算顺序的规则。

附录D“用幂级数计算正弦函数”，介绍了计算正弦函数的一个计算机程序。

已了解C语言的读者如何读本书

C++是建筑在C语言之上的。如果你了解C语言，那么会发现第2章到第12章的内容很熟悉。

C++确实也包含了一些新的特色，如：

- 一个全新的I/O系统。（在第4章“基本说明和表达式”中将介绍基本内容。第16章“文件输入/输出”中还将详细讨论新的文件系统。）
- 常量和引用变量。（在第5章“数组、修饰符和读数”中叙述。）
- 函数重载、内联函数、引用参数和缺省参数。（详见第9章“变量作用域和函数”。）

从第13章“类的初步讨论”开始，我们开始学习全新的概念。类在C++中的地位非同一般，它是最强的特色之一。

编辑约定

本书中用到下列排版体例：

楷体

用来表示名录，强调首次介绍的新术语和概念。

粗体

用来表示C关键字。另外，在示例中表示用户需原样键入的命令或是其它的文本。

引号

用来在解释正文中标明系统信息或是代码段。

`\%`

是UNIX C外壳提示符。

`$`

是UNIX Bourne外壳或Korn外壳提示符。

`#`

是UNIX超级用户的提示符（Bourne或C外壳）。我们常把它用于只应该在根下运行的例子中。

`[]`

描述程序语法时，括在选项值外。（括号本身不要键入。）

`...`

表示为了简洁或节省空间而忽略的正文（常常是计算机的输出）。

记号**CTRL-X**或是[^]**X**表示使用Ctrl键。它的意思是，在按下Ctrl键的同时，键入“X”字符。其它键的表示与此类似（例如，**RETURN**表示回车键）。

所有命令行的例子后都应加上回车键，除非是另外的表示方式。

获取源代码

你可以通过O'Reilly & Associates公司提供的Internet服务，获取本书中所有程序的源代码。

本书中的示例程序电子文件可通过下列几种方法得到：FTP、Ftpmail、BITFTP和UUCP。最便宜、最快速和最简单的方法是第一种。如果你从前向后读，你能使用的第一种方法可能是最好的。如果你可以直接上Internet，那么可以使用FTP。如果你不能直接使用

Internet, 但可以通过Internet网点发送并接收电子邮件（这包括CompuServe用户），那么可以使用Ftpmail。如果你通过BITNET发送电子邮件的话，可以使用BITFTP。如果上面所说的都不适合于你，那么使用UUCP。

FTP

要使用FTP，你的机器必须能直接访问Internet。下面显示一次会话示例，你应该键入的内容以粗体字印刷。

```
% ftp ftp.uu.net
Connected to ftp.uu.net.
220 FTP server (Version 6.21 Tue Mar 10 22:09:55 EST 1992) ready.
Name (ftp.uu.net:joe): anonymous
331 Guest login ok, send domain style e-mail address as password.
Password: joe@ra.com (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/nutshell/practcpp
250 CWD command successful.
ftp> binary (Very important! You must specify binary transfer for compressed files.)
200 Type set to I.
ftp> get examples.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for examples.tar.gz.
226 Transfer complete.
ftp> quit
221 Goodbye.
%
```

文件是一个压缩的tar文档，键入下列命令可以得到文件：

```
% zcat examples.tar.gz | tar xvf -
```

System V系统需要键入下面的tar命令：

```
% zcat examples.tar.gz | tar xof -
```

如果在你的系统上不能使用zcat命令，要使用单独的uncompress和tar或shar命令。

```
% uncompress examples.tar.gz
% tar xvf examples.tar.gz
```

Ftpmail

Ftpmail是一个邮件服务，任何想发送电子邮件或想从Internet网点接收邮件的人都可以使用它。它涵盖了允许email连接到Internet的任何公司或服务提供者。下面叙述如何使用它。

你向ftpmail@online.ora.com发送邮件。在消息体中，写上你想执行的FTP命令。服务器将为你执行该FTP命令，返回邮件。要得到完整的帮助文件，发送一条没有主题的消息，在信息体中写上一个字“help”。

下面是邮件会话示例。该命令给你送来所选目录中的文件列表，及所选的示例文件。如果你对其中的文件感兴趣的话，则列表很有用。

```
% mail ftpmail@online.ora.com
Subject:
reply-to janetv@z.com      { Where you want files mailed }
open
cd /published/oreilly/nutshell/practcpp
mode binary
uuencode
get examples.tar.gz
quit
.
```

消息尾的签名只要是出现在“quit”之后，就被接收。

BITFTP

BITFTP是用于BITNET用户的一个邮件服务。你向它发出需求文件的电子邮件，它通过电子邮件给你送来文件。BITFTP目前只服务于能直接使用BITNET、EARN或NetNorth的用户。BITFTP是Princeton University提供的公共服务。下面是它的用法。

要使用BITFTP，把含有ftp命令的邮件发送到BITFTP@PUCC。要得到完整的帮助文件，把HELP作为消息体来发送。

下面是你发送给BITFTP的消息体：

```
FTP ftp.uu.net NETDATA
USER anonymous
PASS myname@dunk.edu Put your Internet email address here ( not your BITNET address)
CD /published/oreilly/nutshell/practcpp
DIR
BINARY
GET examples.tar.gz
QUIT
```

一旦你得到了所需的文件，可以遵照FTP的命令从文档中取出文件。因为你使用的可能不是UNIX系统，所以你的系统可能需要uudecode、uncompress、atob和tar。现在有用于VMS、DOS和Mac的版本。

UUCP

UUCP是所有UNIX系统的一个实际标准，它可从IBM兼容的PC和Apple Macintosh机上得到。通过modem从UUNET可得到示例；UUNET的连接费用可以记帐。

不管你有没有帐户，你都可以从UUNET得到示例。如果你或你的公司有UUNET帐户，你就有让UUCP直接连到UUNET的一个系统。找到该系统，并键入：

```
uucp uunet!!~/published/oreilly/nutshell/practcpp/examples.tar.gz
yourhost!!~/yourname/
```

如果你使用的是Bourne外壳（sh）而不是csh，那么反斜杠可被忽略。过些时候（最多一或两天）文件就会出现在目录/usr/spool/uucppublic yourname中。如果你没有帐户，但想要一个，以便能得到电子邮件，你可以打电话703-204-8000，与该公司联系。

把文件/published/oreilly/lis-IR.Z作为一个短的测试文件是个好主意，其中含有可得到文件的文件名和文件大小。

一旦你得到了所需的文件，可以遵照FTP的命令从文档中取出文件。

评论和问题

涉及本书的评论和问题请与出版商联系：

O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472

1-800-998-9938（在美国或加拿大）
1-707-829-0515（国际或本地）
1-707-829-0104（传真）

致 谢

感谢Peg Kovar女士的校对和编辑工作。特别感谢Dale Dougherty 撕毁了我的第一本书，使我不得不把它重新正确地组织在一起。非常感谢Phil Straite和Gregory Satir的艰辛工作。我还要特别感谢所有参与本书编辑的工作人员。我还要感谢O'Reilly & Associates公司制作组的成员——项目经理和责任编辑Nicole Gipson先生；出版助理John Files先生、Juliette Muellner和Jane Ellin女士；以及本书设计人Mike Sierra先生。最后，还要特别感谢努力工作的所有程序员们，他们的代码教会了我许多东西。

目 录

前 言

第一部分 基础

第1章 什么是C++	3
C++历史回顾	3
C++组织	4
如何学习C++	5
第2章 编程基础	7
程序从概念到运行	9
编写一个真正的程序	10
使用命令行编译器编写一个程序	10
使用集成开发环境编写一个程序	13
获取UNIX帮助	26
获取集成开发环境的帮助	28
编程练习	28
第3章 风格	29
注释	29
C++代码	34
命名风格	35
编程盲从	36
缩进与编码格式	36
清楚	37
简明	38
一致性与组织结构	38
继续读下去	38
小结	39
第4章 基本说明和表达式	41
程序元素	41
程序的基本结构	41

简单表达式	43
cout输出类	44
变量和存储	45
变量说明	46
整数	46
赋值语句	47
浮点数	48
浮点数除法与整数除法	48
字符	50
编程练习	51
答疑	51
第5章 数组、修饰符和读数	53
数组	53
串	54
读数据	57
变量初始化	59
多维数组	60
整型	62
浮点型	64
常量与引用说明	64
修饰符	65
十六进制和八进制常量	67
简捷运算符	68
副作用	69
编程练习	71
答疑	71
第6章 判定和控制语句	73
if语句	73
else语句	74
为何不使用strcmp函数	76
循环语句	76
while语句	76
break语句	78
continue语句	79
随处赋值的副作用	80
编程练习	81
答疑	82

第7章 程序设计过程	85
设置	87
规格说明	87
代码设计	89
原型	89
Makefile	90
测试	93
调试	93
维护	96
修改	96
电子考古	96
编排程序	97
使用调试器	97
使用文本编辑器作为浏览器	97
加注释	97
编程练习	100
 第二部分 简单的程序设计	
 第8章 再次讨论控制语句	103
for语句	103
switch语句	106
switch、break和continue	111
编程练习	113
答疑	114
 第9章 变量作用域和函数	115
作用域和存储类	115
函数	118
参数类型--览	131
结构化程序设计基础	131
递归	132
编程练习	134
答疑	134
 第10章 C++预处理器	135
#define语句	135
条件编译	140

#include文件	142
参数宏	143
高级特征	145
小结	145
编程练习	146
答疑	146
第11章 位运算	149
位运算符	150
与运算符 (&)	150
按位或 ()	152
按位异或 (^)	153
非运算符 (~)	154
左移和右移运算符 (<<, >>)	154
设置、清除和检测位	155
位图图形	159
编程练习	164
答疑	164
第三部分 高级类型和类	
第12章 高级类型	167
结构	167
联合	169
typedef	172
enum类型	173
位域或紧缩结构	175
结构数组	177
编程练习	178
第13章 类的初步讨论	179
栈	179
改进的栈	183
类的使用	185
介绍构造函数和析构函数	188
自动生成的成员函数	192
快捷方法	193
风格	194
编程练习	196

第14章 类的进一步讨论	199
友元	199
常量函数	201
常量成员	202
静态成员变量	203
静态成员函数	205
static的含义	206
编程练习	206
第15章 简单指针	209
常量指针	213
指针和输出	214
指针和数组	215
分隔串	219
指针和结构	222
命令行自变量	223
编程练习	227
答疑	227
第四部分 高级编程概念	
第16章 文件输入/输出	233
C++文件I/O	233
转换程序	237
二进制和ASCII码文件	241
行尾难题	242
二进制I/O	244
缓冲问题	244
非缓冲I/O	245
设计文件格式	249
C风格的I/O程序	251
C风格的转换程序	253
C风格的二进制I/O	256
编程练习	258
答疑	258
第17章 调试和优化	261
调试	261

连续调试	270
分而治之	270
只用于调试的代码	270
调试命令行开关	270
通读输出	272
交互调试器	272
调试一个二分查找程序	276
运行时错误	288
公开声明调试方法	290
优化	290
2的幂的效能	292
如何优化	295
实例研究：内联函数对正常函数	297
实例研究：优化一个涂色算法	297
编程练习	297
答疑	298
第18章 运算符重载	299
运算符函数	301
运算符成员函数	310
复数类的完整定义	312
编程练习	322
答疑	323
第19章 浮点数	325
浮点数格式	325
浮点数加法/减法	326
乘法	327
除法	327
上溢和下溢	328
舍入误差	328
精度	329
让舍入误差最小	329
判定精度	330
精度和速度	331
幂级数	332
编程练习	333

第20章 高级指针	335
指针、结构和类	336
delete运算符	337
链表	338
顺序链表	342
双向链表	344
树	349
树的打印	352
程序的其余遗留部分	353
象棋程序的数据结构	356
编程练习	358
答疑	358
第21章 高级类	361
派生类	361
虚函数	367
虚类	373
隐藏在派生类中的函数	375
派生类中的构造函数和析构函数	376
小结	379
编程练习	379
答疑	379
第五部分 语言的其它特性	
第22章 例外	383
栈例外	384
运行时库例外	390
编程练习	390
第23章 模块化程序设计	391
模块	391
公用和私用	391
extern修饰符	392
头文件	394
模块体	395
使用无限数组的程序	395
用于多文件的Makefile	397

无限数组的使用	401
把任务划分为模块	407
模块划分示例：文本编辑器	407
编译器构造	408
电子表格	410
模块设计准则	411
编程练习	411
第24章 模板	413
什么是模板	413
模板：严格的方法	413
函数特例	416
类模板	418
类特例	420
实现困难	420
小结	423
编程练习	423
第25章 移植问题	425
模块化	425
字大小	425
字节顺序问题	426
对齐问题	427
NULL指针问题	428
文件名问题	429
文件类型	429
小结	430
答疑	430
第26章 组合在一起	433
需求	433
代码设计	435
编码	436
功能描述	437
测试	441
修改	441
最后的警告	442
程序文件	442
编程练习	464

第27章 从C到C++	465
总的看法	465
K&R风格的函数	465
struct	466
malloc和free	466
把结构转换为类	468
ssetjmp和longjmp	469
小结	470
编程练习	471
第28章 C++内的“角落”	473
do/while	473
goto	473
?:命令	475
逗号运算符	475
()运算符的重载	476
指向成员的指针	476
“吸血鬼”特性	477
答疑	478
第29章 程序设计格言	479
概述	479
设计	480
说明	480
switch语句	480
预处理器	480
风格	481
编译	481
C++程序员的10条戒律	481
最后的注解	482
答疑	482
第六部分 附录	
附录 A ASCII码表	485
附录 B 范围	489
附录 C 运算符优先级规则	491
附录 D 用幂级数计算正弦函数	493
词汇表	501

第一部分

基 础

本章内容:

- C++历史回顾
- C++组织
- 如何学习C++

第1章

什么是C++

在现代社会中，组织及处理信息的能力是成功的关键。设计计算机的目的就是为了快速而有效地处理大量的信息。但是，除非有人告诉计算机干什么，否则，它不会做任何事情。这正是C++诞生之缘由。C++是一种高级编程语言，能够让软件工程师与计算机进行有效的对话。

C++是一种非常灵活且适应性强的语言。自1980年诞生之日起，它一直被用来设计各种各样的程序，包括用于微控制器的微程序、操作系统、应用程序和图形程序。C++很快成为编程时的候选对象。操纵计算机的人非常需要有合适的语言，而C++可以让你做得更快更好。

C++历史回顾

1970年，两位程序员Brian Kernighan和Dennis Ritchie首创了一种新的语言，取名为C语言。（之所以取这个名字，是因为C取代了他们原先使用的B语言。）设计C语言有一个目的：编写操作系统。它极其简单、灵活，很快就被用来编写各种不同类型的程序，成为世界上最流行的语言之一。

但C有一个很重要的问题，它是面向过程的语言。这就是说，如果要设计一个普通的C程序，程序员必须从数据的描述开始，然后编制操纵数据的过程。

程序员最终发现，如果他能把数据组织起来，并与作用于这些数据的操作结合在一起，程序会变得清楚，而且易于理解。这样的一个组叫做一个对象（object）或是类（class）。通过设计类来设计程序的方法称为面向对象的程序设计（OOD）。

1980年，Bjarne Stroustrup开始用一种新的语言工作，这种语言被称做“带类的C语言”。它增加了一些新特性，全面改进了C语言，其中最重要的特性就是类。这种语言经过改进、扩充，最终成为C++。

C++的成功是因为它允许程序员使用比其它大多数语言更高效的方法组织并处理信息。而且，它植根于C语言。事实上，大多数C程序都可以毫不费力地转变为C++程序。这些程序通常没有利用C++的所有新特性，但它们也能正常运行。通过这种方式，C++允许程序员

在已有C代码的基础上做进一步的开发。

C++组织

C++是作为程序员与裸机之间的桥梁而设计的。最初的想法是想让程序员用一种容易理解的语言设计程序，然后，编译程序再把这种语言翻译为机器可以使用的语言。

计算机程序由两个主要部分组成：数据和指令。计算机并没有区分这两部分，但计算机总是被设计得尽可能通用。程序员可以按他自己的方式组织这两部分。

计算机中的数据被存储为一系列的字节。C++把这些字节组织为有用的数据。程序员使用数据说明来描述他们要操作的信息。例如：

```
int total;      // Total number accounts
```

告诉C++你想用计算机的一段内存来存放一个整数，这个整数名为total。你可以让计算机决定具体使用内存中的哪些字节，而对这些细节你不必担心。

变量total是一个简单变量（*simple variable*）。它仅能存放一个整数，并且只能描述一个量。一系列的整数可以组织为数组。数组在内存中存放的细节问题也由C++来处理。

```
int balance[100]; // Balance (in cents) for all 100 accounts
```

最后，还有更复杂的数据类型。例如，一个矩形可能有一个宽度、一个高度、一种颜色和一种填充方式。C++允许你把这四个属性组织为一个结构（*structure*）。

```
struct rectangle {  
    int width;           // Width of rectangle in pixels  
    int height;          // Height of rectangle in pixels  
    color_type color;    // Color of the rectangle  
    fill_type fill;      // Fill pattern  
};
```

不过，数据仅仅是程序的一个部分。你还需要指令。计算机丝毫也不了解指令是如何组织的，它只知道执行当前指令，并知道到哪里取出下一条指令。

C++是一种高级语言，它允许程序员书写下面这样的高级语句：

```
area = (base * height) / 2.0; // Compute area of triangle
```

计算机把这个语句转换为一系列机器指令。这类语句称为赋值语句（*assignment statement*）。它计算并存储算术表达式的值。

你还可以使用控制语句（*control statement*）来控制执行的顺序。像 if 和 switch 这样的语句能让计算机进行简单的判定。使用 while 和 for 这样的循环语句可以重复地执行一些语句。

语句组可以组成函数 (*function*)。所以如果你想画一个矩形, 只需写一个通用函数, 而且只写一次即可。当想画新矩形时, 可以再次使用这个函数。C++提供一组丰富的标准函数 (*standard function*), 这些函数可以完成常见的功能, 如查找、排序、输入和输出。

一组相关函数可以组织为一个模块 (*module*), 模块相连就构成了程序 (*program*)。

C++语言的一个重要目的是把指令组织为可再使用的组件。这样, 如果你的大部分代码可以“借用”别人已有的, 那么写程序就快得多了。多组可重用模块构成一个库 (*library*)。例如, 如果你需要一个排序程序, 可以使用库中的标准函数 *qsort*, 把它与你的程序连接在一起。

计算机把所有的内容划分为数据和指令。长期以来, 像C这样的高级语言一直延续这样的规则。在C程序中, 你可以定义数据或者书写指令, 但不能把两者结合在一起。

C++独创的主要特色之一是它把数据和指令联合为一个称做类或是对象的结构。面向对象的程序设计允许你把数据和作用于它的操作组织在一起。在C++中, 允许你从一个已有的类中派生出新类, 从而把面向对象的概念又延伸了一步。

派生这个功能非常强大。它允许你在较小的类上, 再建立复杂的类。它还允许你定义一个基本的抽象类, 然后从它派生出具体的类。例如, 抽象类 *shape* 可用来定义 *rectangle*, *triangle* 和 *circle*。

组织是编制好程序的关键。比如本书, 你知道目录在前面, 而索引在后面, 因为这是书的组织方法。组织使得本书的使用更加方便了。

C++语言允许你用简单强大的语法 (*syntax*) 来组织程序。本书不仅讲解C++的语法, 还介绍编制可读及可靠程序的风格规则。强大的语法加上好的编程风格, 就可以创建能完成复杂奇妙操作的强大程序。

如何学习C++

学习编程的唯一方法就是写程序。通过写程序并调试 (*debugging*) 程序, 可以从中学到许多东西, 比只读这本书要多得多。本书包含许多的编程练习, 读者应该尽量都做一遍。做练习时, 应该保持良好的编程风格。即使你只是自己练习, 也要写注释。注释可以帮助你理顺思路; 当编写真正的程序时, 给程序加注释是一种好习惯。

不要总想着: “我写这些程序是为自己使用的, 所以不需要为它们写注释。”首先, 编写时看上去非常简单的代码, 过一阵再读时往往弄不明白。给程序写注释有助于你组织思路。(如果你能用英语书写想法, 那么在C++写注释就会事半功倍的。)

最后, 程序可能会用在意想不到的地方。我曾经在Caltech的一台计算机上写过一个程序。这个程序与系统的关系密切。因为我是唯一使用它的人, 所以我把程序设计为: 当我在命令行输入了错误的命令后, 程序会打印出下列信息:

```
?LSTUIT User is a twit (使用者是个大笨蛋)
```

几年后，我在Syracuse大学读书。计算机科学学院的一位秘书需要一个程序，它与我在Caltech编写的程序很相似，所以我把那个程序修改了一下，交给了那位秘书去用。不幸的是，我忘记了输入错误命令时出现的那个玩笑信息。

可以想像，当计算机科学学院秘书长把我叫到办公室谈话时，我是多么的震惊。这位女士很有权势，连系主任都怕她三分。她看着我，说道：“使用者是个大笨蛋，嗯？”好在她很幽默，否则，我也不会有今天。

本书中会有一些“失败”的程序，请花些时间找出错误之所在。有些问题非常微妙，比如，分号放错了位置，或者该写==的地方误写为=等。本书教给你在小程序中找出这些错误的方法。当在大程序中出现类似错误时，训练有素的你将会迅速指出错误。

本章内容:

- 程序从概念到运行
- 编写一个真正的程序
- 使用命令行编译器编写一个程序
- 使用集成开发环境编写一个程序
- 获取帮助
- 编程练习

第2章

编程基础

计算机是很强大的工具，它可以存储、组织并处理大量的信息。但是，除非有人告诉它们具体的指令，否则它们不会做任何事情。

与计算机进行通讯并不容易，它们需要精确详细的指令。如果我们用英语编写程序会不会简单一些？这样的话，我们就可以告诉计算机：“把我的帐单和存款统计一下，然后把总额告诉我”，计算机应该自动把我的支票簿结算清楚。

但是当必须书写精确指令的时候，你会发现英语是一种很糟糕的语言。英语充满模棱两可的意思，而且不精确。计算机之母Grace Hopper女士曾经评论过她在洗发水瓶子上发现的指令：

洗发

冲洗

重复

她按照指令去做，结果她把洗发水用光了。（洗－冲－重复，洗－冲－重复，洗－冲－重复...）

当然，我们可以试着用精确的英语来写程序。我们必须非常仔细，保证把所有的词汇拼写正确，并保证包含了处理所有情况的指令。如果我们工作得足够努力，就可以写出精确的英语指令，对吗？

其实，有许多人在从事书写精确英语的工作。他们被称做公务员，他们书写的文档被称为政府法规。不幸的是，公务员越努力想使法规精确，文档的可读性越差。如果你曾经读过有关税收条款的规定，就知道精确英语是怎么一回事了。

但是，即使法律条文加入了所有必须的字眼，也还是出了问题。几年前，美国加州通过了一项法律，要求摩托车驾驶员都要戴头盔。这条法律刚颁布不久，一位警察拦住了一位没戴头盔驾驶摩托车的小伙子。小伙子建议警官仔细看看这条法律。

该条法律有两条要求：1) 摩托车驾驶员有一个好的防撞头盔，2) 把头盔用带子绑牢。警察最终也没能给这位驾驶员开罚单，因为他确实绑了一个头盔——在膝盖上。

所以说，因受自身的限制，英语不适于做计算机语言。那么，我们怎样才能与计算机通话呢？

第一台计算机价值数百万美元，那时，一个好程序员的年薪达到15,000美元。程序员使用一种能把所有指令转化为一串数的语言来编写程序，这种语言叫做**机器语言**(*machine language*)。它可以直接输入到计算机中。典型的机器语言程序如下：

```
1010 1111
0011 0111
0111 0110
.. and so on for several hundred instructions
```

机器以数来“思考”，而人却不行。为了给这些老式的机器编程，软件工程师使用了一种简单的语言来书写程序。在这种语言中，每个字代表一个单独的指令。因为程序员必须自行把每行程序翻译或汇编为机器语言，所以这种语言被称为**汇编语言**(*assembly language*)。

一个典型的程序如下所示：

```
Program Translation
MOV A,47 1010 1111
ADD A,B 0011 0111
HALT    0111 0110
.. and so on for several hundred instructions
```

这个过程如图2.1所示。

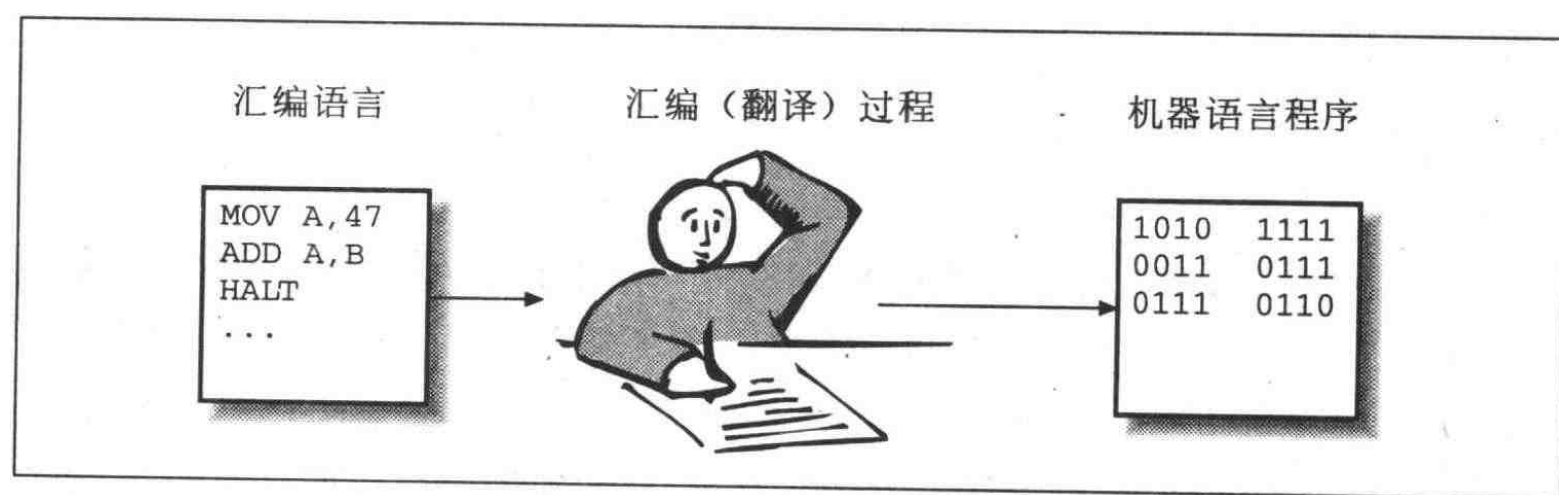


图 2.1 汇编一个程序

翻译是一项困难、乏味、精确的任务。有一位软件工程师认为这项工作完全适合于计算机，所以他写了一个程序，叫做**汇编程序**(*assembler*)。这样这项工作就可以自动完成了。

他把这个想法告诉了他的老板，但立即得到了责骂：“你竟敢让这样一台昂贵的机器仅仅去干‘文书’的工作？”如果拿计算机运转一小时的花费与程序员工作一小时的费用相比，这个态度并不是没有道理的。

幸运的是，随着时代的前进，程序员的费用上升了，而计算机的费用下降了。所以让程序员用汇编语言书写程序，然后再使用汇编程序把这个程序翻译为机器语言程序，这样的过程是最节省费用的。

汇编语言使用程序员很容易理解的方法来组织程序。但是，机器理解这些程序更加困难了。这样的程序在机器运行之前必须先翻译。这是一种潮流的开始。编程语言变得越来越方便于程序员使用，而计算机把它们翻译为机器可用语言的时间却越来越长了。

多年来，研究人员已设计了一系列的高级语言 (*high-level language*)。这些语言都力图让程序员用易于理解的方式书写程序，同时又必须足够精确而且简单，以便计算机能够明白。

早期的高级语言只能处理具体的应用。FORTRAN用于数值计算；COBOL用于编写商业报告；而PASCAL用于教学。（这些语言中，有许多远远超出了它们最初的使用范围。据说，Nicklaus Wirth说过：“如果我知道PASCAL会变得这样成功，在当初设计时，我会更加仔细。”）后来，Brian Kernighan和Dennis Ritchie开发了C语言，Bjarne Stroustrup把它发展为C++。

程序从概念到运行

C++程序是用一种高级语言编写的，其中使用了字符、数字以及能在计算机键盘上找得到的其它符号。实际上，计算机运行一种非常低级的语言 (*low-level language*)，这种语言称为机器代码 (*machine code*)（一串数字）。所以，程序使用之前，必须先做转化。

程序最初是由程序员酝酿的。他使用文本编辑器 (*text editor*) 把他的思路写到一个文件中，这个文件叫源文件 (*source file*) 或是源代码 (*source code*)。该文件由编译器转换为目标文件 (*object file*)。然后，连接程序 (*linker*) 把目标文件与标准库 (*standard library*) 中的预定义程序结合在一起，产生一个可执行程序 (*executable program*)（一组机器语言指令）。在随后的介绍中，你会了解程序的这几种不同形式是如何相互配合工作并生成最终程序的。

图2.2列出了用高级语言编写的一个程序转为可执行程序时所需的步骤。

包 装

幸运的是，你不必单独运行编译器、汇编程序和连接程序。大多数C++编译器使用“包装 (*wrapper*)”程序，由它来决定需要运行哪些程序，并使之运行。

有些编程系统更进了一步，它们为开发人员提供一个集成的开发环境 (IDE)。IDE包含了一个编辑器、编译器、连接程序、项目管理程序、调试器和其它一些工具，这些都放在一个方便的包内。Borland和微软两公司都为它们的编译程序提供了IDE。

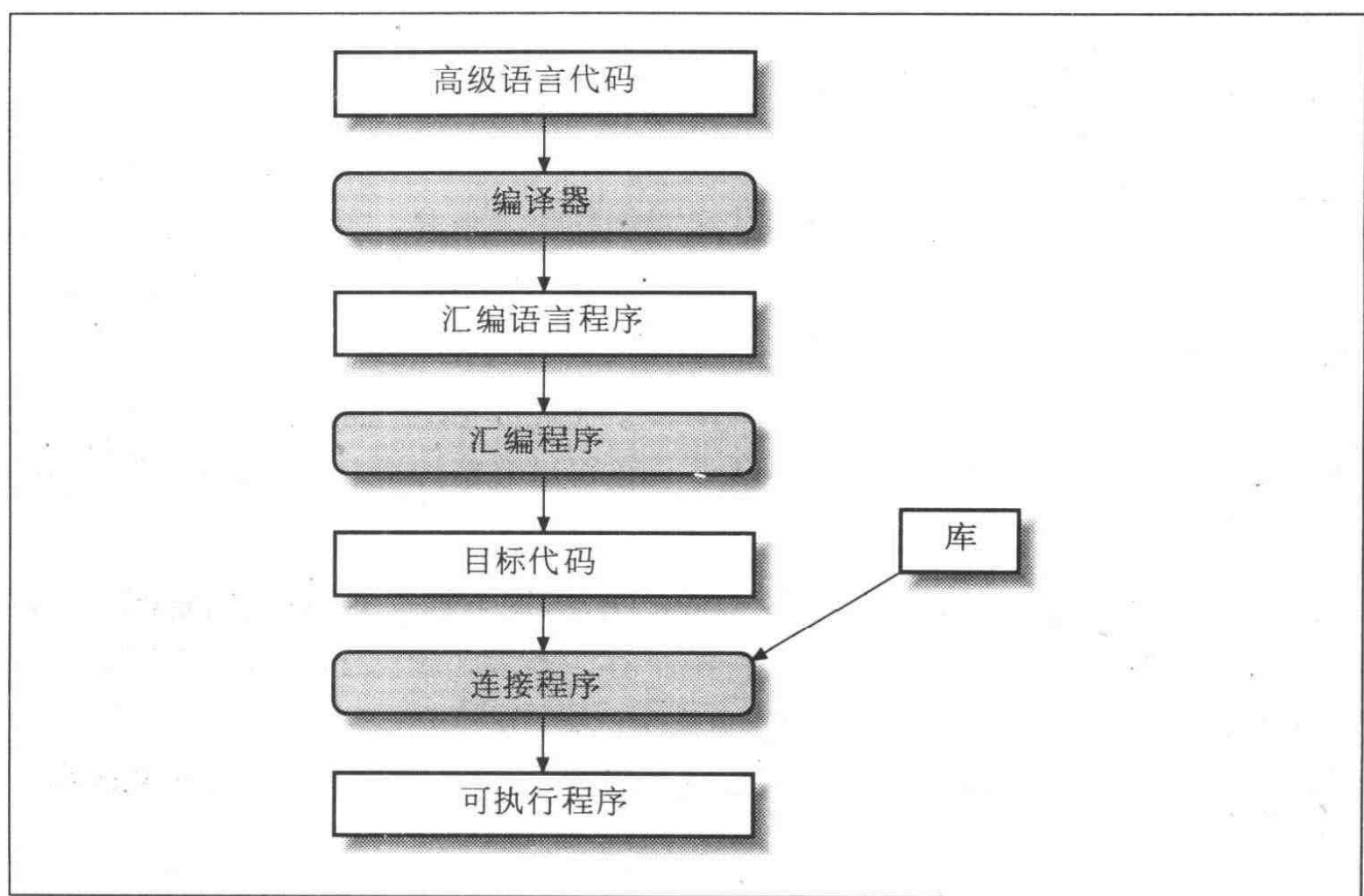


图 2.2 把高级语言程序转为可执行程序

编写一个真正的程序

在你能真正动手编制自己的程序之前，需要了解如何使用基本的编程工具。这一部分将向读者详细介绍录入、编译并运行一个简单程序的各个步骤。

这一部分将介绍两类不同编译器的使用方法。第一种类型为独立的或是命令行编译器。这类编译器以批处理的方式从命令行运行。换句话说，你要键入一个命令，然后编译器把源代码转化为可执行程序。另一类编译器包含在IDE中。

大部分UNIX系统使用命令行编译器。只有很少的几个IDE类型编译器运行在UNIX下。而另一方面，用于MS-DOS和Windows的编译器差不多都包含集成开发环境。因为命令行方式根深蒂固，所以这些编译器也包含命令行编译器。

使用命令行编译器编写一个程序

这一部分将学习使用命令行编译器编写程序的详细步骤。我们将列出使用普通UNIX编译器、Free Software Foundation的g++编译器、Turbo-C++、Borland C++和Microsoft Visual C++的命令。

但是，如果你使用Borland或是微软编译器，应该转到介绍IDE使用方法的部分。

第1步：为你的程序找个地方

如果为每个程序建立一个单独的目录，那么管理工作将非常简单。本例中，将建立一个名为`hello`的目录，该目录下存放`hello`程序。

在UNIX系统下，键入如下命令：

```
% mkdir hello
% cd hello
```

在MS-DOS系统下，键入如下命令：

```
C: MKDIR HELLO
C: CD HELLO
```

第2步：建立一个程序

程序最初是一个文本文件。示例2.1列出了`hello`程序的源代码。

示例2.1 Source for the `hello.cc` program

```
#include <iostream.h>
int main()
{
    cout << "Hello World\n";
    return (0);
}
```

使用你喜欢的文本编辑器键入这个程序。在UNIX系统下，相应的文件名为`hello.cc`，在MS-DOS/Windows下，文件名为`HELLO.CPP`。

警告

MS-DOS/Windows用户不要用字处理软件来写程序，比如不要使用Microsoft Word或是WordPerfect来写。字处理软件在文件中加入了排版格式符号，而编译器不能正确区分这些符号。你必须使用能编辑ASCII文件的文本编辑器，例如MS-DOS EDIT。

第3步：运行编译器

编译器把你刚录入的源文件转变为可执行程序。每个编译器都有不同的命令行。下面是一些最流行的编译器。

UNIX系统下的CC编译器（通用UNIX）

大多数基于UNIX的编译器都遵从同一个标准。C++编译器名为CC。用下列命令就可以

编译我们的hello程序:

```
% CC -g -ohello hello.cc
```

-g选项表示可以进行调试。(编译器在程序中增加了额外的信息,以使调试更容易一些。)开关-ohello告诉编译器最后生成的程序名为hello,最后一项hello.cc是源文件的名称。其它选项的用法请参看你使用的编译器手册。在UNIX系统下,有几个不同的C++编译器,所以你使用的命令行可能略有不同。

Free Software Foundation的g++编译器

Free Software Foundation公司研制了多个高质量的程序。(如想得到它们的软件,请查阅词典中“Free Software Foundation”词条的内容。)在他们提供的程序中,有一个C++编译器,名为g++。

用g++编译器来编译程序时,要键入下列命令:

```
% g++ -g -Wall -ohello hello.cc
```

这个命令中多了一个开关-Wall,它表示显示所有的警告信息。

Borland公司用于MS-DOS的Turbo-C++

Borland公司有一个便宜的MS-DOS方式C++编译器,名为Turbo-C++。该编译器对学习编程的人来说,是一个理想的程序。Turbo-C++的命令行为:

```
C:>tcc -ml -v -N -P -w -ehello hello.cpp
```

-ml选项告诉Turbo-C++使用大内存模式。(该PC机在创建程序时可以使用多种内存模式。本书不讨论这个问题。我们赞同这样的观点:“使用大模式时不必担心,除非有一天你成为一名有经验的程序员。”)

-v开关告诉Turbo-C++编译器在程序中加入调试信息;-w选项将打开警告;-N选项将检查堆栈。实际上,编译器既编译C程序,也编译C++程序。我们使用-P开关强制进行编译C++程序。最后一个选项-ehello告诉Turbo-C++建立一个名为hello的程序,而hello.cpp是源代码的名字。其它选项的用法请参看Turbo-C++参考手册中相关的内容。

MS-DOS和Windows下的Borland C++

除了Turbo-C++编译器,Borland公司还有另外一个用于MS-DOS/Windows的全功能、专业级的编译器,名为Borland C++。它的命令行为:

```
C:>bcc -ml -v -N -P -w -ehello hello.cpp
```

Turbo-C++和Borland C++两编译器使用的命令行选项一样。

Microsoft Visual C++

Microsoft Visual C++是用于MS-DOS/Windows的另一个C++编译器。它的功能不如Borland的产品强大,特性也不如后者多。但在本书中,我们将用它编译大多数的程序。(1.5版不能很好地处理模板和例外情况。)

这个编译器使用下面的命令行：

```
C:>cl /AL /Zi /W1 hello.cpp
```

/AL选项告诉编译器使用大内存模式。使用/Zi选项将打开调试开关，使用/W1选项将显示警告信息。

第4步：运行程序

现在，键入命令就可以运行程序了。例如，在UNIX或是MS-DOS提示符下，键入：

```
hello
```

屏幕上将显示下列信息：

```
Hello World
```

使用集成开发环境编写一个程序

编程时，集成的开发环境提供所有必需的工具。这些工具有编辑器、编译器和调试器，它们集中在一个紧凑的包内供程序员使用。

第1步：为你的程序找个地方

如果为每个程序建立一个单独的目录，那么管理工作将非常简单。本例中，将建立一个名为**HELLO**的目录，该目录下存放**hello**程序。

在MS-DOS系统下，键入如下命令：

```
C: MKDIR HELLO
```

```
C: CD HELLO
```

第2步：录入、编译并运行程序

每个IDE都略有不同，针对每个编译器，我们列出各自的指令。

Turbo-C++

1. 使用下列命令启动Turbo-C++的IDE:

```
C: TC
```

2. 使用Options|Compiler|Code Generation命令打开Code Generation对话框，对话框如图2.3所示。把内存模式改为大模式。
3. 使用Options|Compiler|Entry/Exit命令打开堆栈检查对话框，该对话框如图2.4所示。
4. 使用Options|Compiler|Messages|Display命令打开Compiler Message对话框，该对话框如图2.5所示。选择All，以显示所有的警告信息。

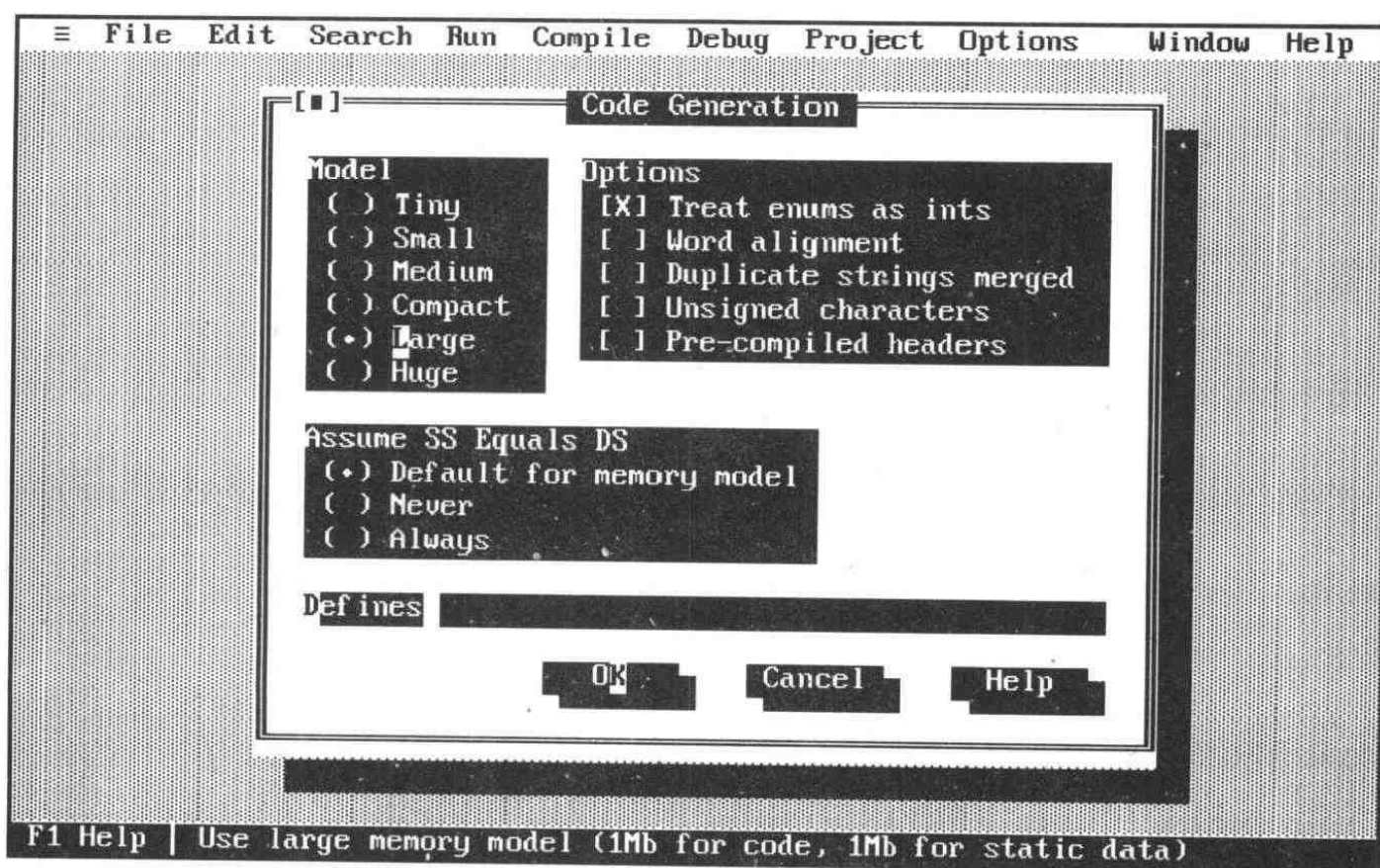


图 2.3 Code Generation对话框

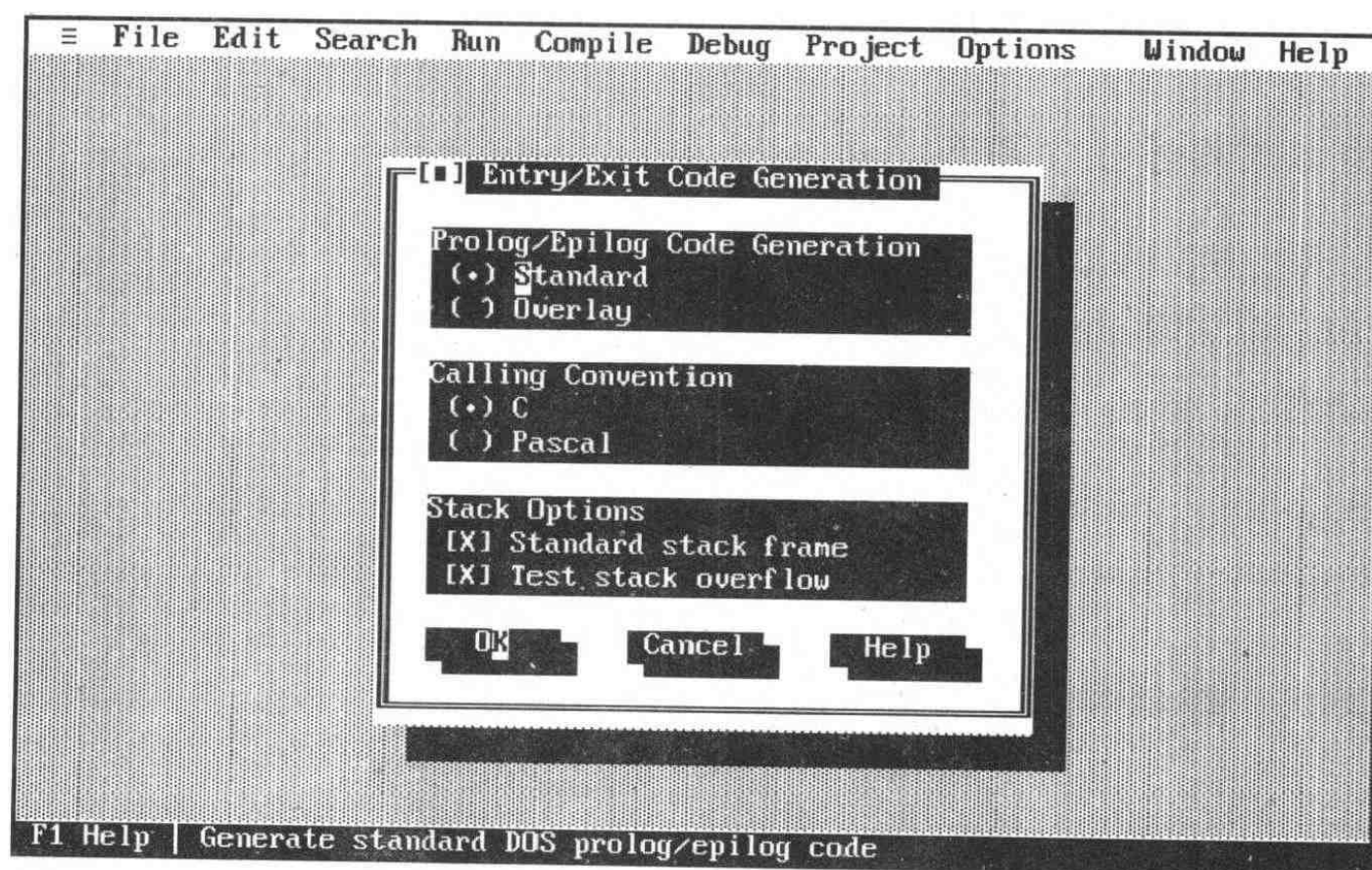


图 2.4 Entry/Exit Code Generation对话框

5. 使用Options\Save命令，把到目前为止所有你所使用过的选项保存起来。
6. 使用Open Project File对话框，选择一个项目文件。本例中，你的项目文件名为HELLO.PRJ。屏幕上出现的窗口如图2.6所示。
7. 按Insert键，把文件加到项目中。你要加入的文件是HELLO.CPP，窗口如图2.7所示。
8. 按ESC键，退出“Add File”循环。

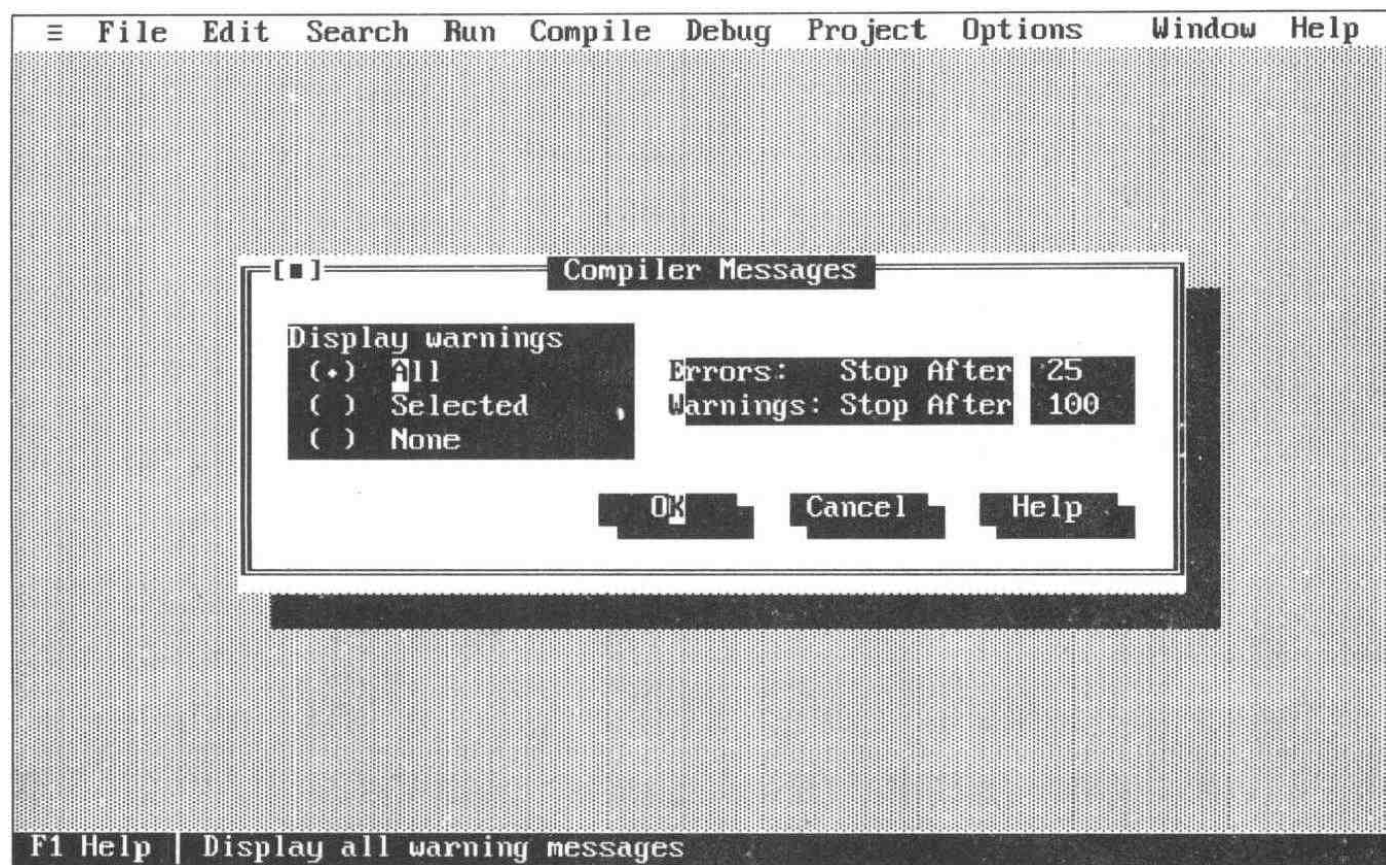


图 2.5 Compiler Messages对话框

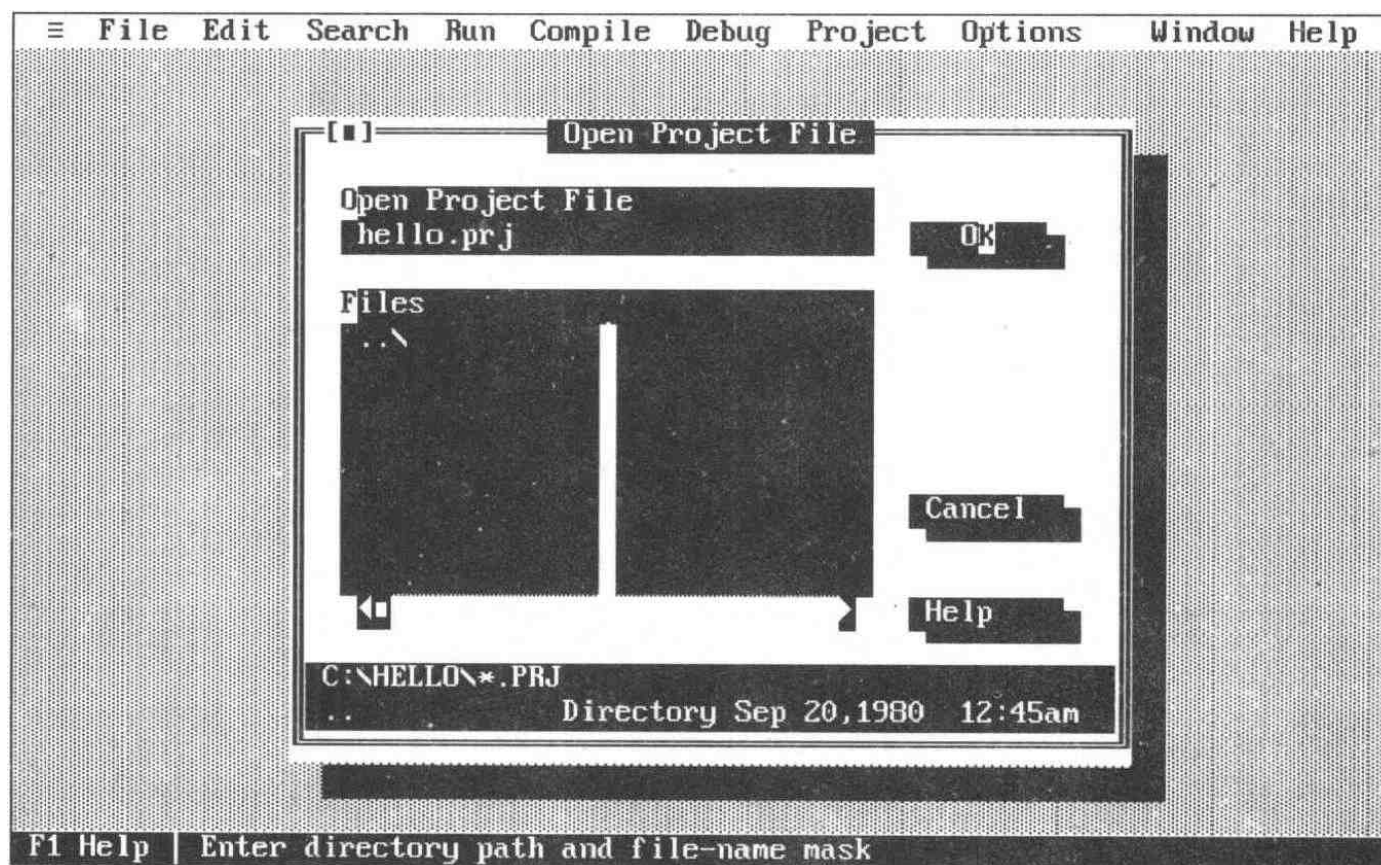


图 2.6 Open Project File对话框

9. 按下箭头，光标上移一行。写有`hello.cpp`的行将呈高亮显示，窗口如图2.8所示。
10. 按Return键，编辑这个文件。
11. 录入下列代码。

```
#include <iostream.h>
int main()
{
    cout << "Hello World\n";
}
```

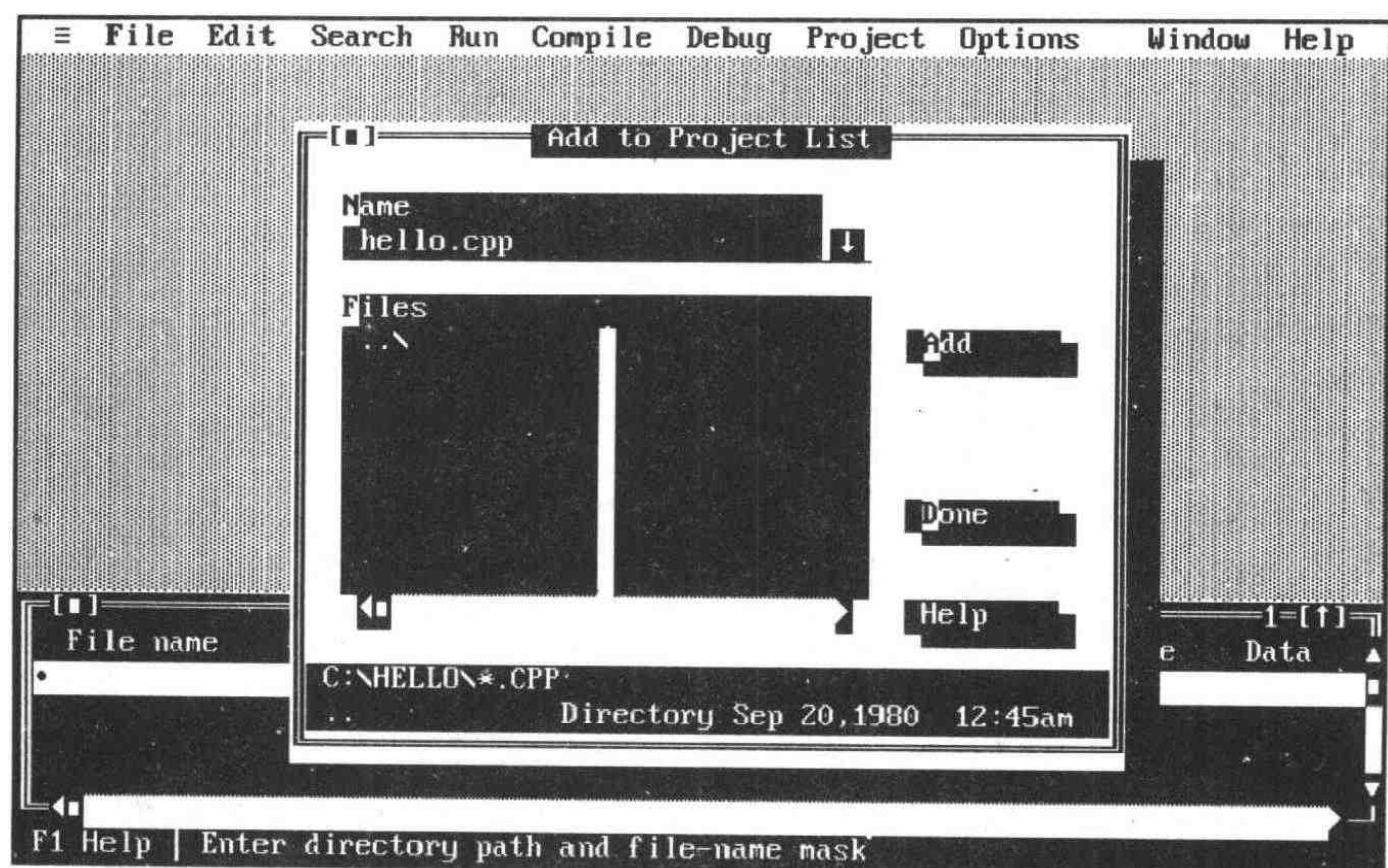


图 2.7 Add to Project List对话框

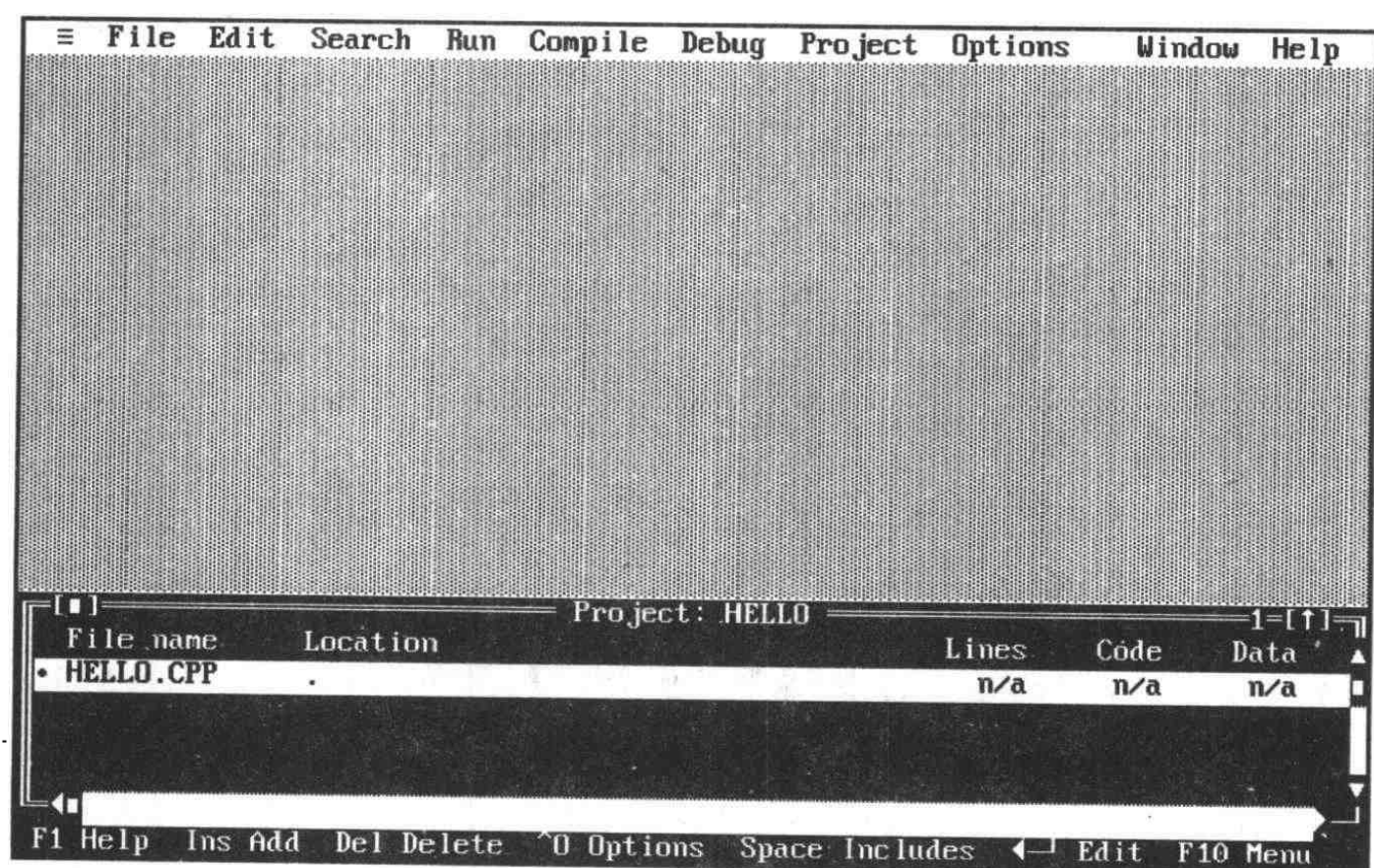


图 2.8 “Hello”项目

```

    return (0);
}

```

此时窗口如图2.9所示。

12. 使用RunRun命令运行这个程序。

13. 程序运行后，控制权交给了IDE。这就是说，你看不到程序的输出。如果想看程序的结果，你必须使用WindowUser命令切换到用户屏幕。

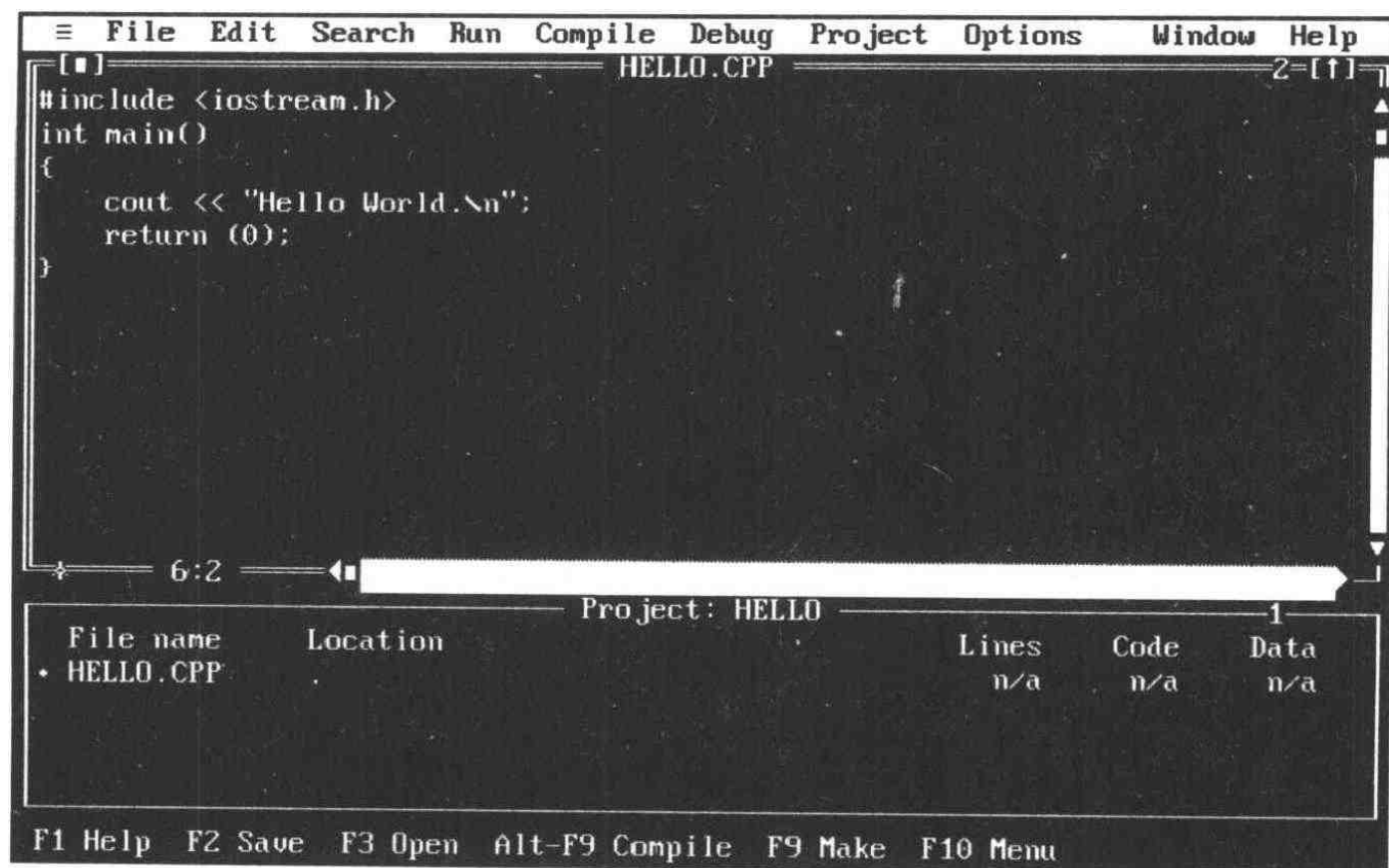


图 2.9 已完成的项目

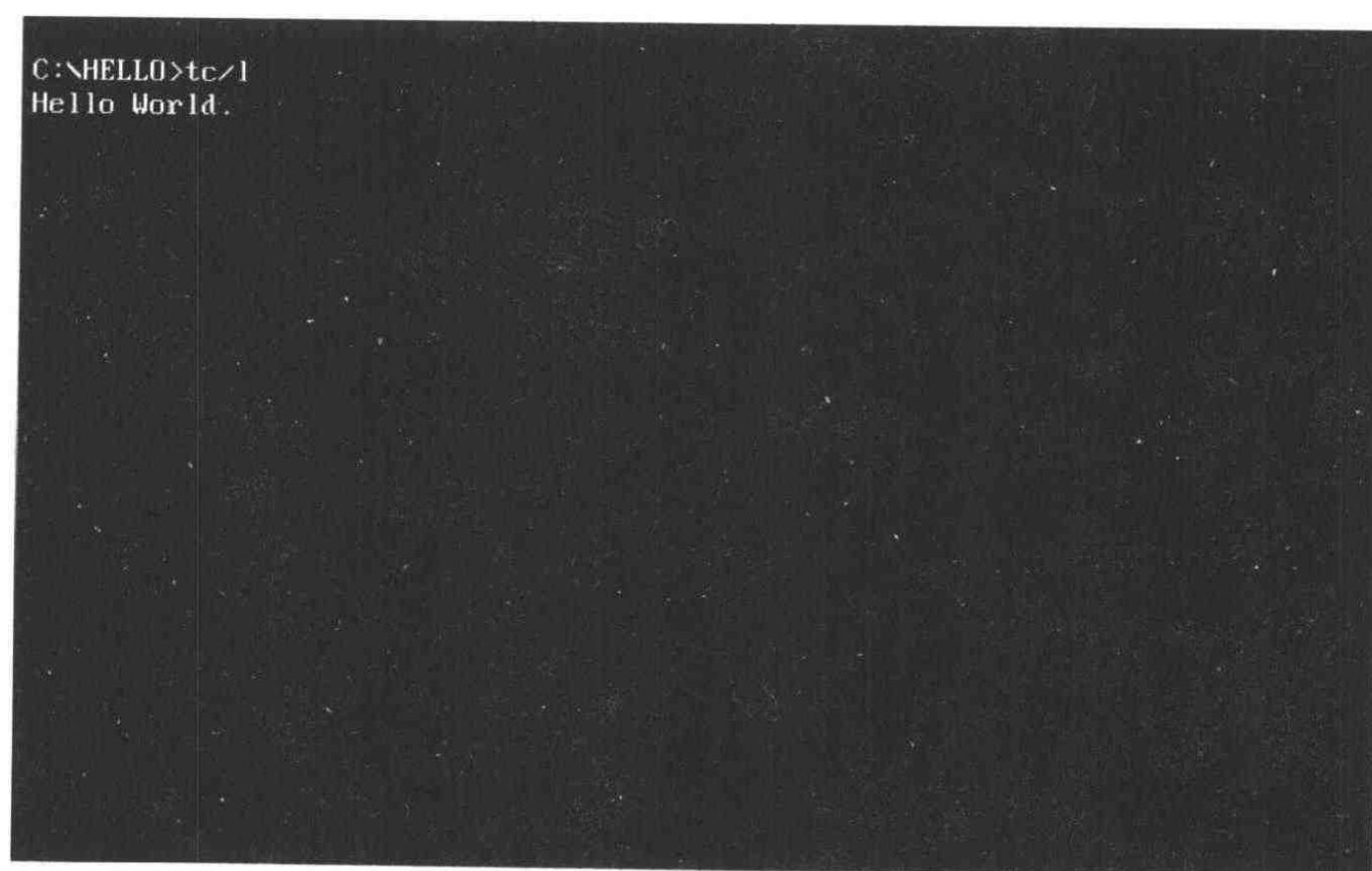


图 2.10 用户屏幕

按任意键后，将返回到IDE中。图2.10显示程序的输出结果。

14. 一切做完以后，可以使用File|Save命令保存这个程序。

15. 使用File|Quit命令退出IDE。

Borland C++

1. 建立一个子目录用来存放hello程序，该目录名为HELLO。读者可以使用Windows的File Manager（文件管理器）建立目录，也可以在MS-DOS提示符下键入下列命令来建立目录：

mkdir \HELLO

2. 在Windows中，双击Borland C++图标，启动IDE。程序开始运行，屏幕上出现空白的工作区，如图2.11所示。

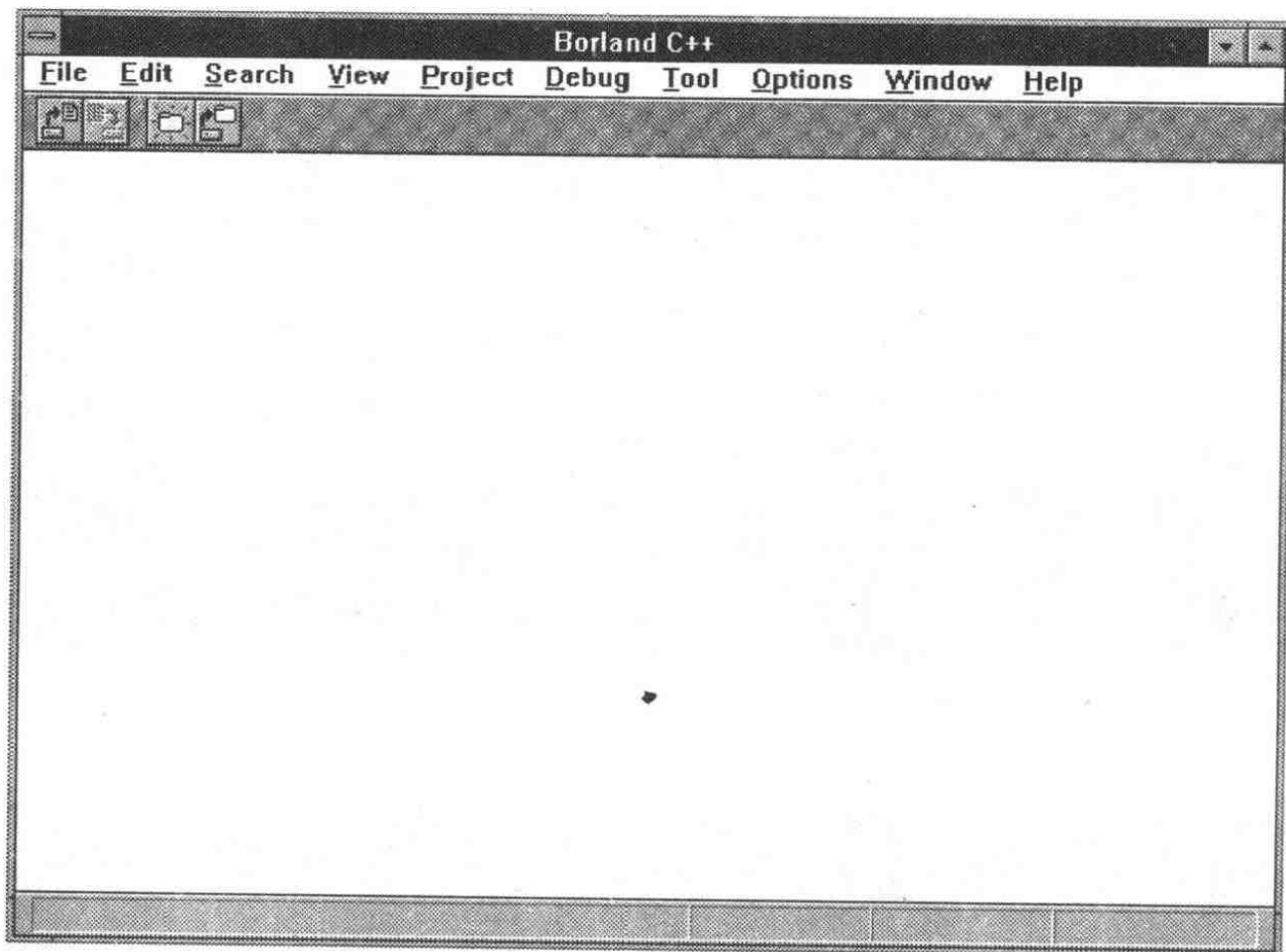


图 2.11 Borland C++的初始屏幕

3. 在菜单中选取 Project\New Project项，为我们的示例程序建立一个项目。在“Project Path and Name:”后的空白处填入：`c:\hello\hello.ide`。在Target Type 选择框中选择EasyWin[.exe]。在Target Model下拉列表中选择Large。相应的窗口如图2.12所示。

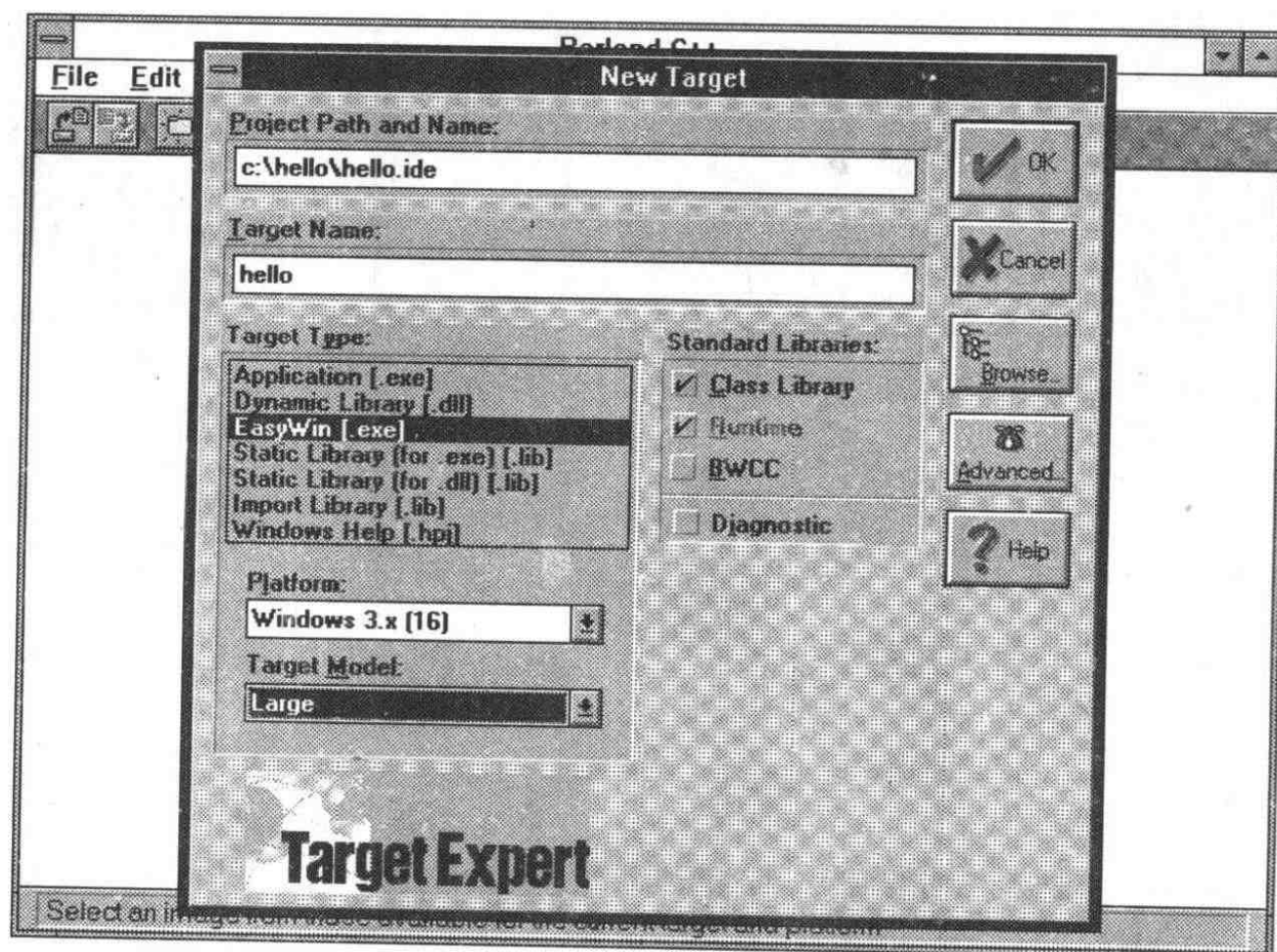


图 2.12 New Target对话框

4. 点击Advanced按钮。打开Advanced Options对话框。清除.rc和.def选项。这个窗口如图2.13所示。

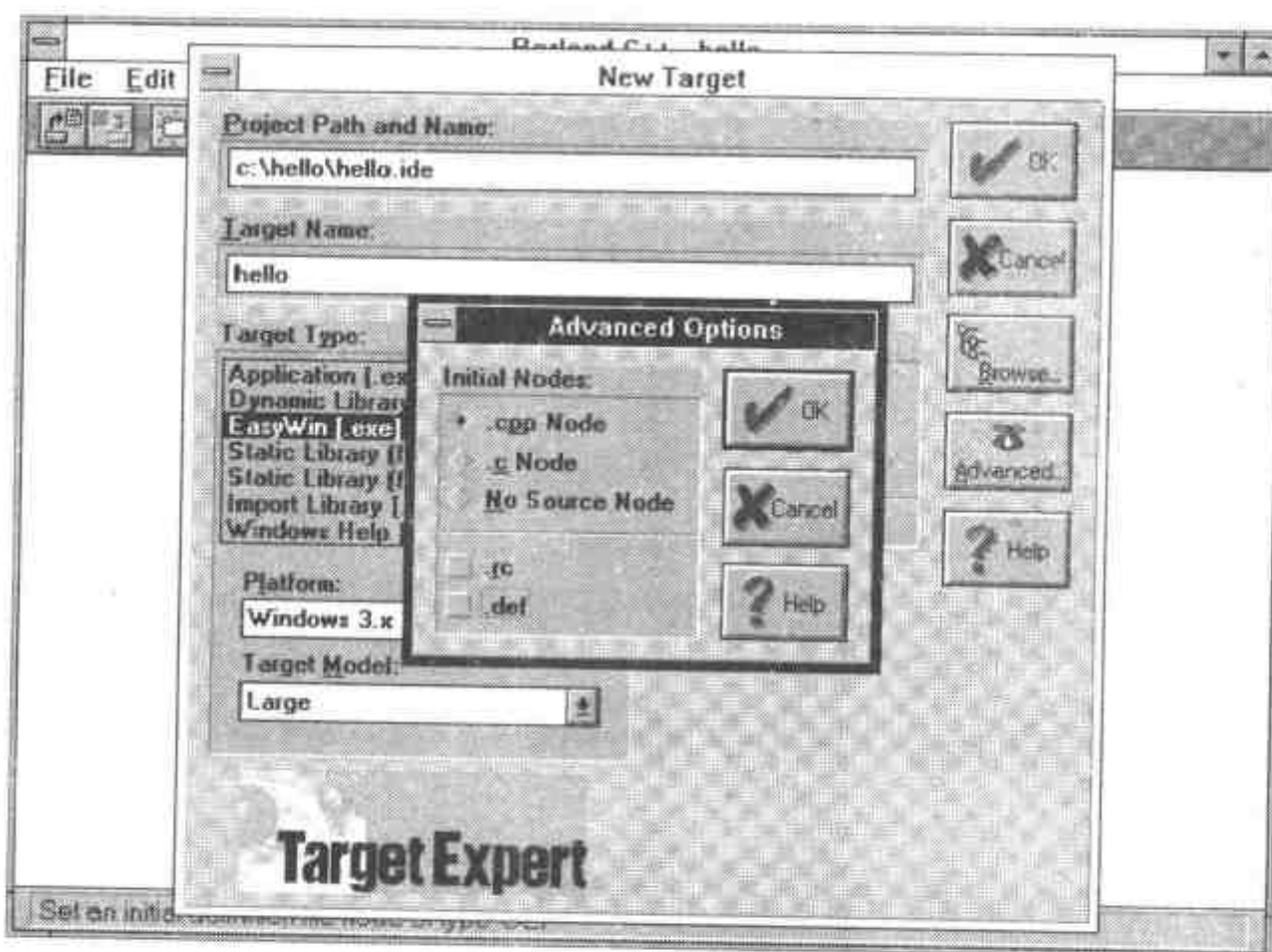


图 2.13 Advanced Options对话框

5. 点击OK按钮，返回New Target对话框。
6. 按下Alt-F10，激活节点子菜单，该子菜单如图2.14所示。

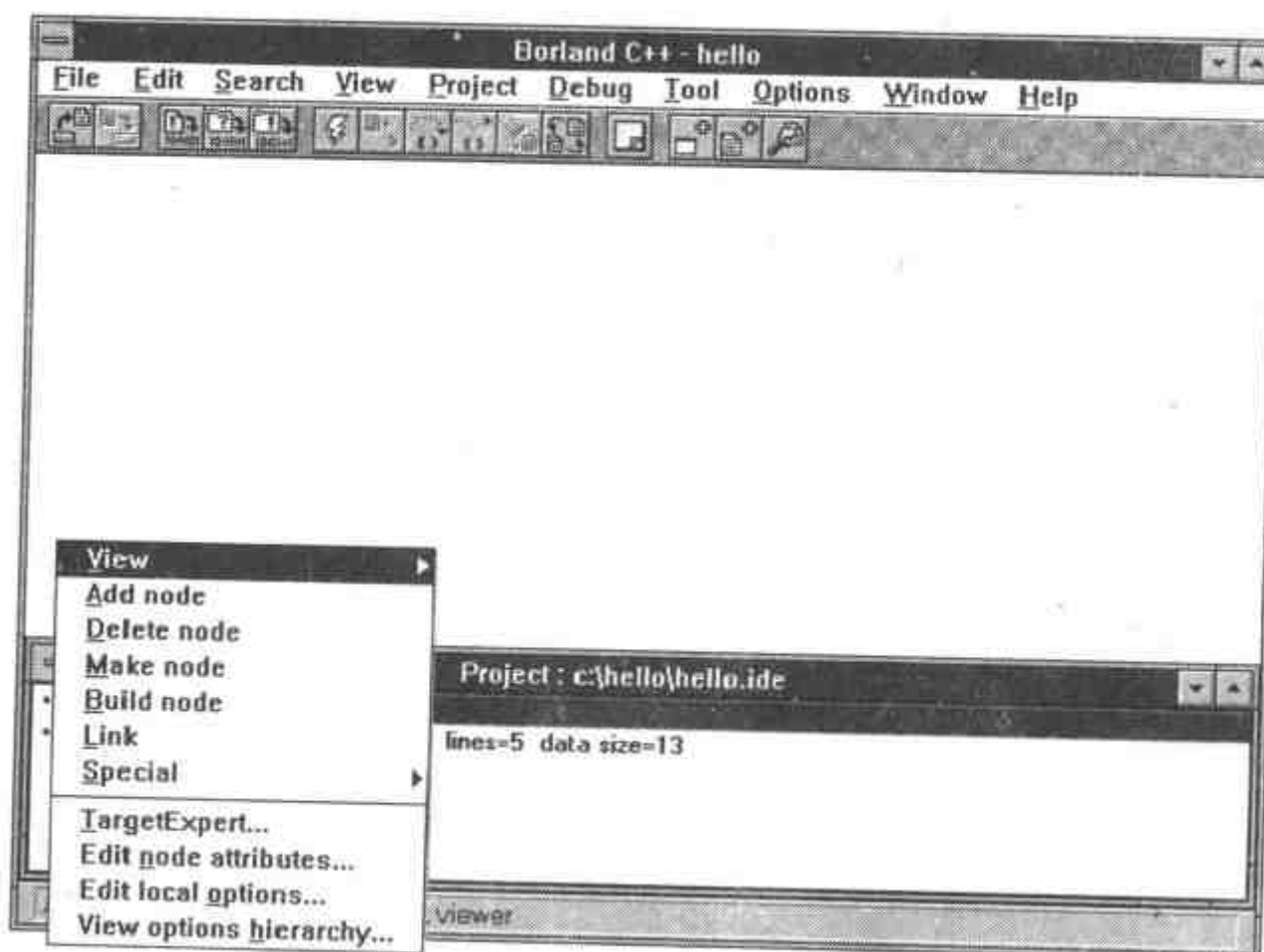


图 2.14 Target Options子菜单

7. 选择Edit Node Attributes, 出现的对话框如图2.15所示。在Style Sheet空格处选择“Debug Info and Diagnostics”项。点击OK后返回到主窗口。

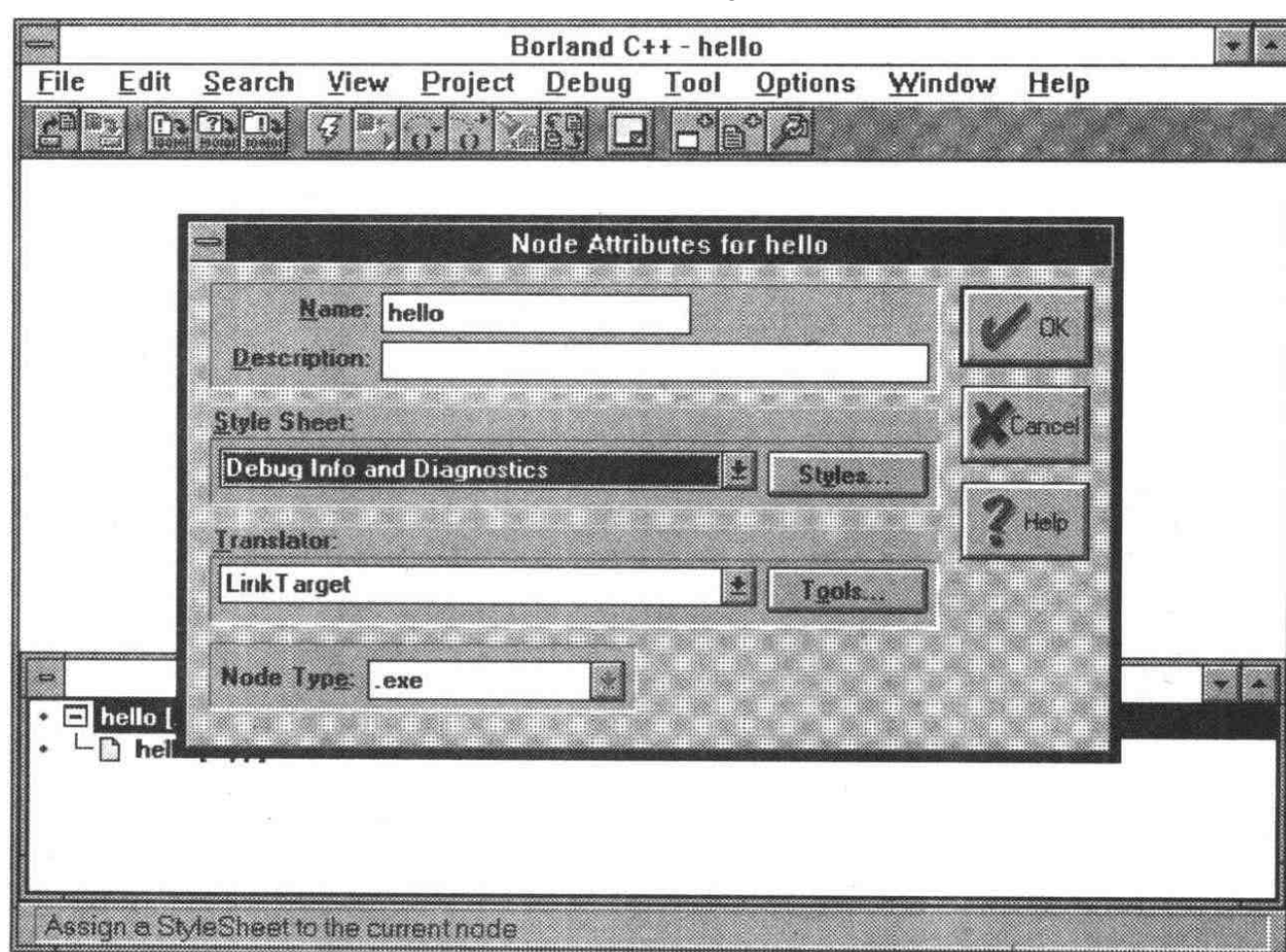


图 2.15 Node Attributes对话框

8. 选择Options|Project Options项, 进入Project Options 对话框。选项条向下移动到Compiler项, 点击“+”, 以包含这个选项。

打开Test stack overflow选项, 见图2.16。点击OK, 保存这些选项。

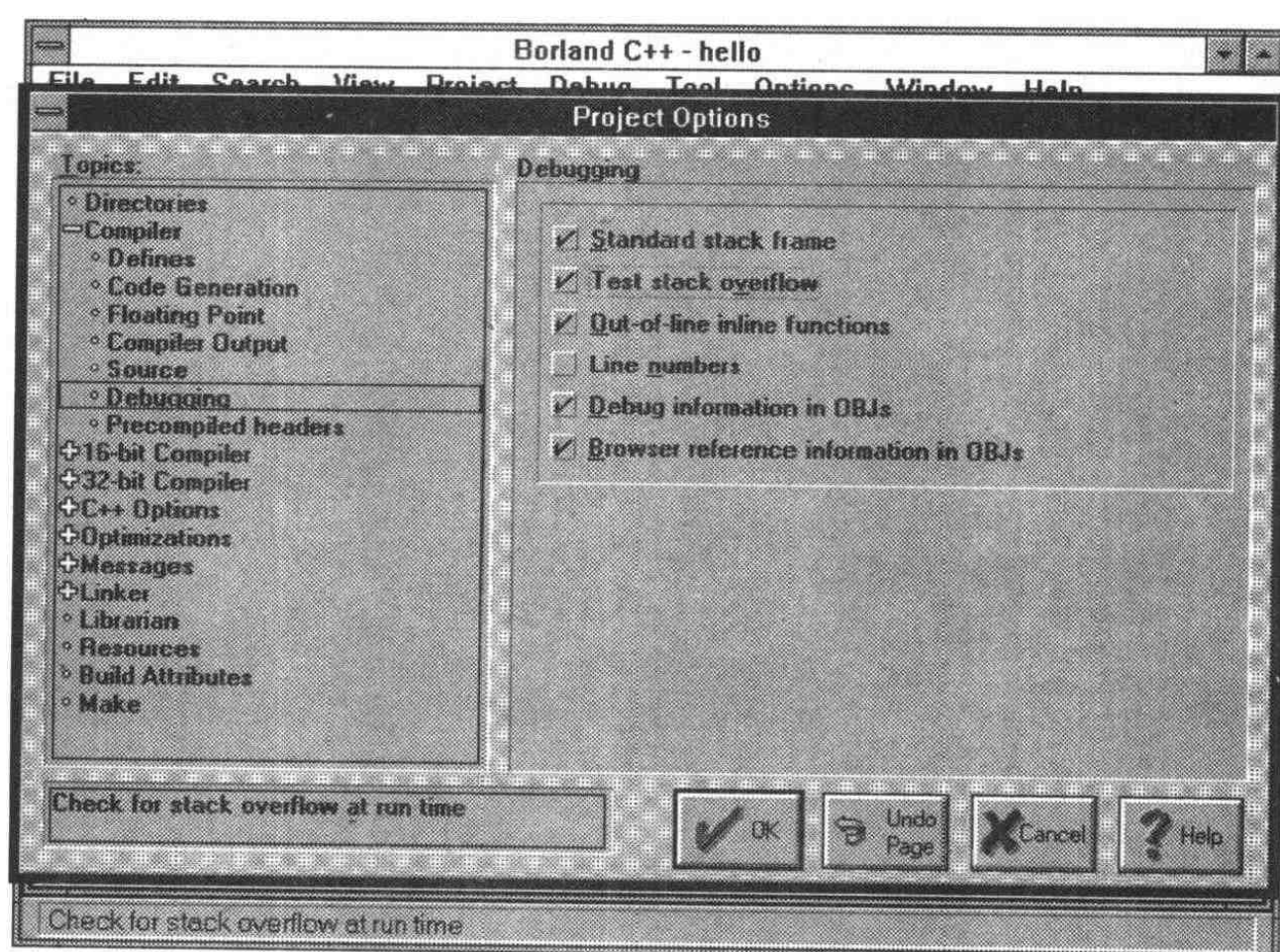


图 2.16 Project Options对话框

9. 点击OK返回主窗口。按下箭头，在project窗口中选择hello[.cpp]项（见图2.17）。

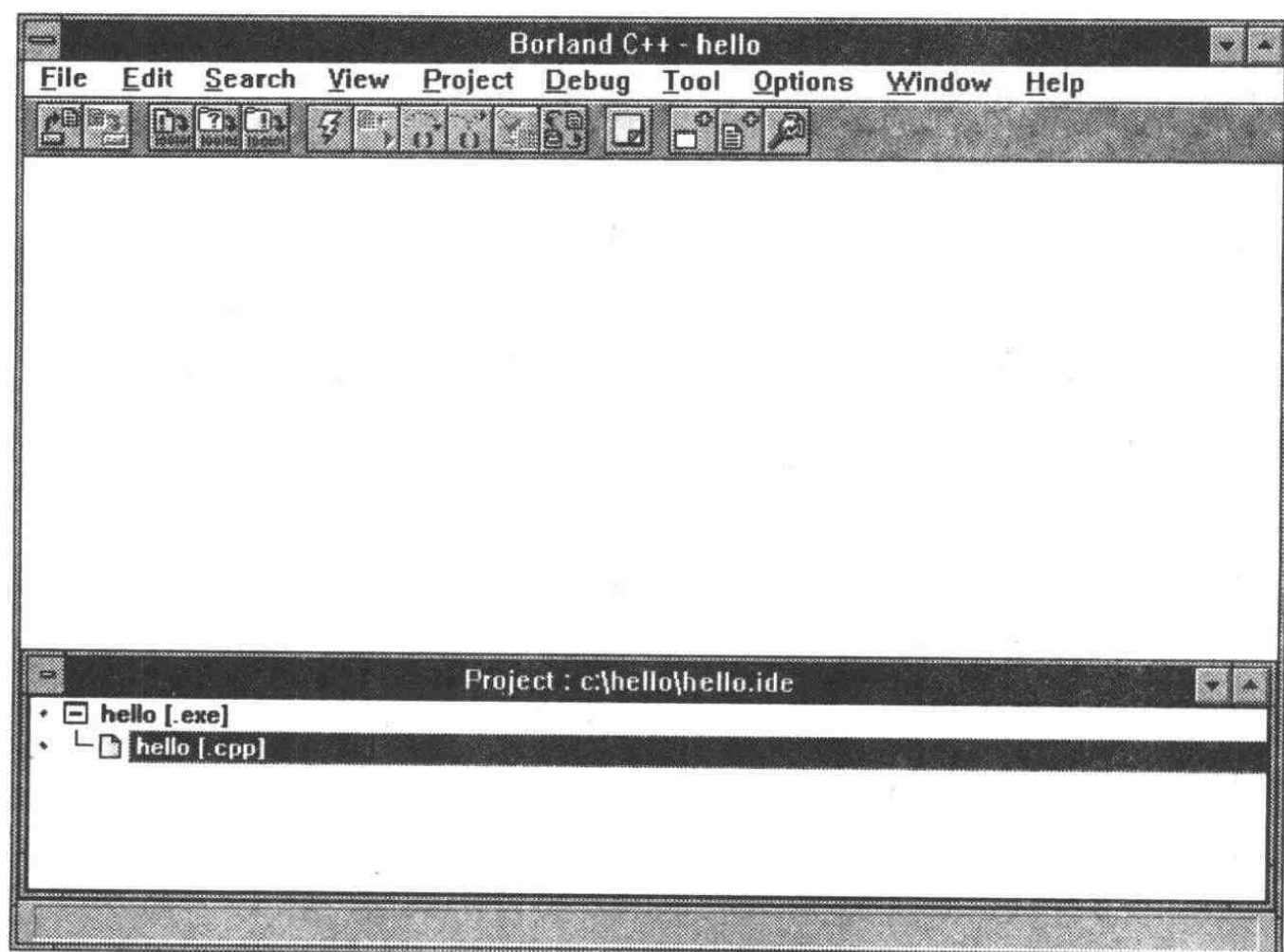


图 2.17 Hello Project

10. 按Return，开始编辑文件hello.cpp。键入下列代码：

```
#include <iostream.h>
int main()
{
    cout << "Hello World\n";
    return (0);
}
```

完成后，屏幕如图2.18所示。

11. 选取Debug|Run菜单项，编译并运行这个程序。程序运行后，将在窗口中显示“Hello World”信息，窗口如图2.19所示。

Microsoft Visual C++

1. 建立一个子目录用来存放hello程序，该目录名为HELLO。读者可以使用Windows的File Manager（文件管理器）建立目录，也可以在MS-DOS提示符下键入下列命令来建立目录：

```
mkdir \HELLO
```

2. 在Windows中，双击Visual C++图标，启动IDE。程序开始运行，屏幕上出现空白的工作区，如图2.20所示。

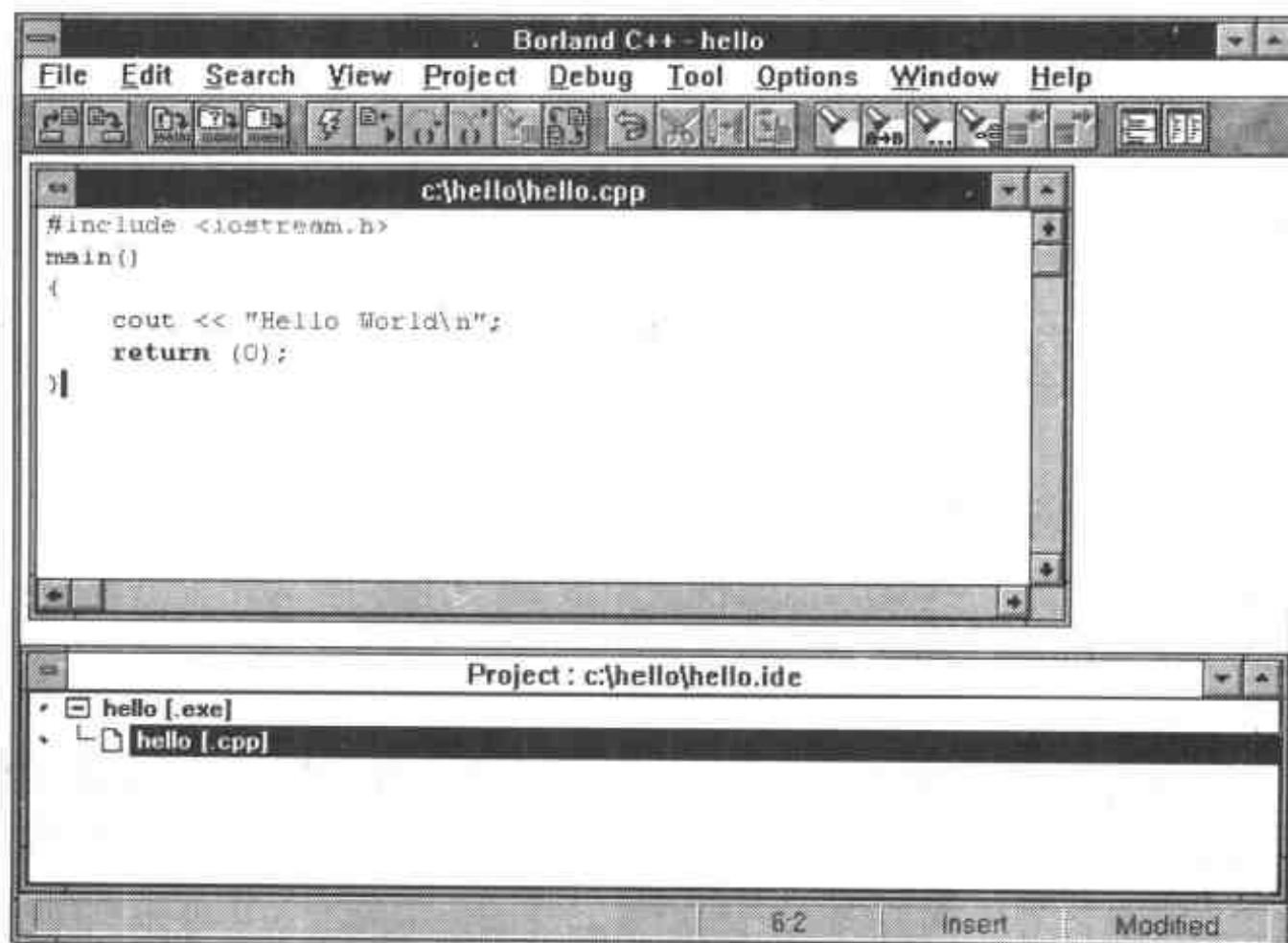


图 2.18 “Hello World” 程序

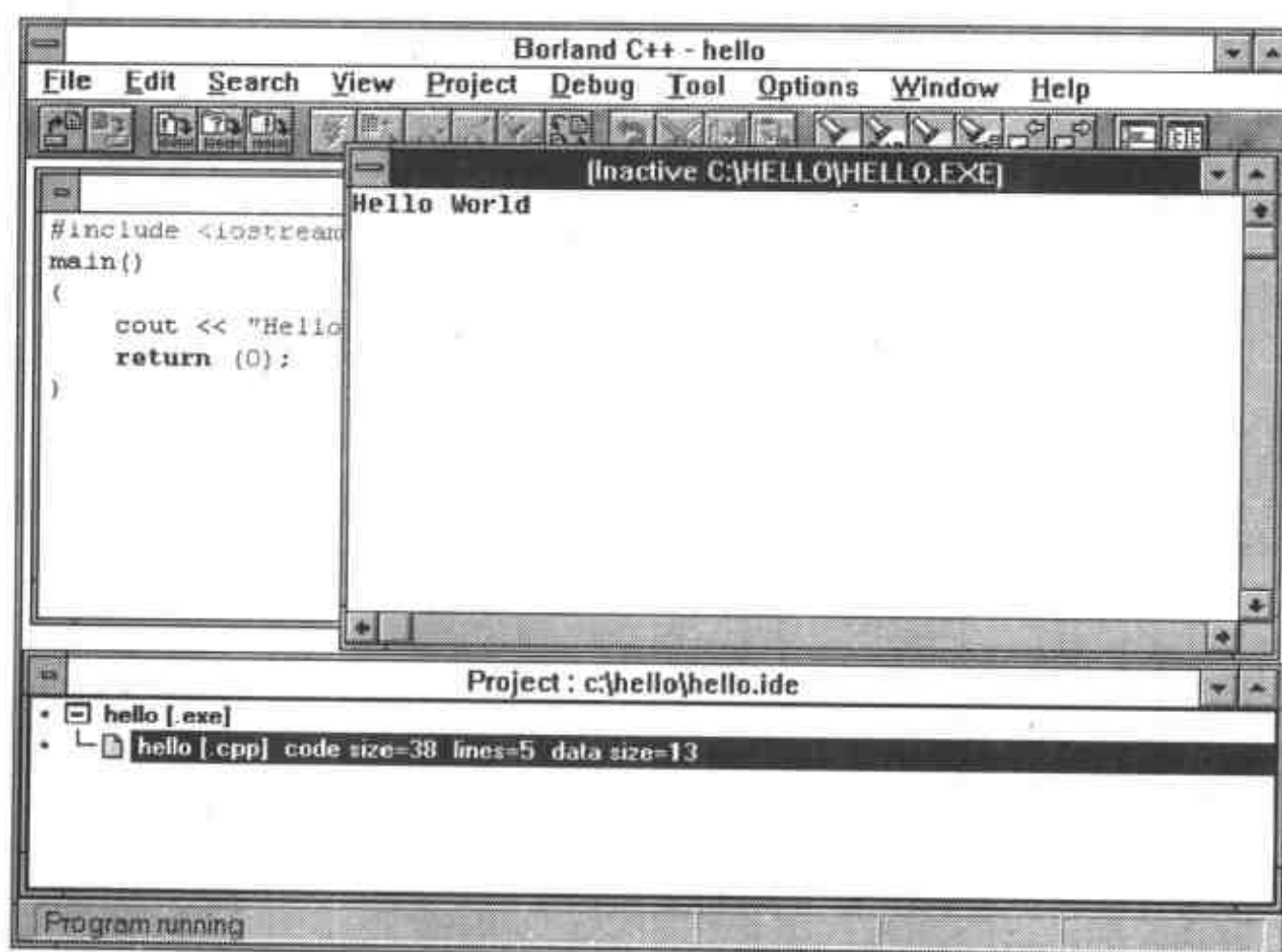


图 2.19 程序运行后显示 “Hello World”

3. 在菜单中选取Project\New项，出现的New Project对话框如图2.21所示。在Project Name后的空白处填入：\hello\hello.mak。把Project Type改为QuickWin application[.EXE]。

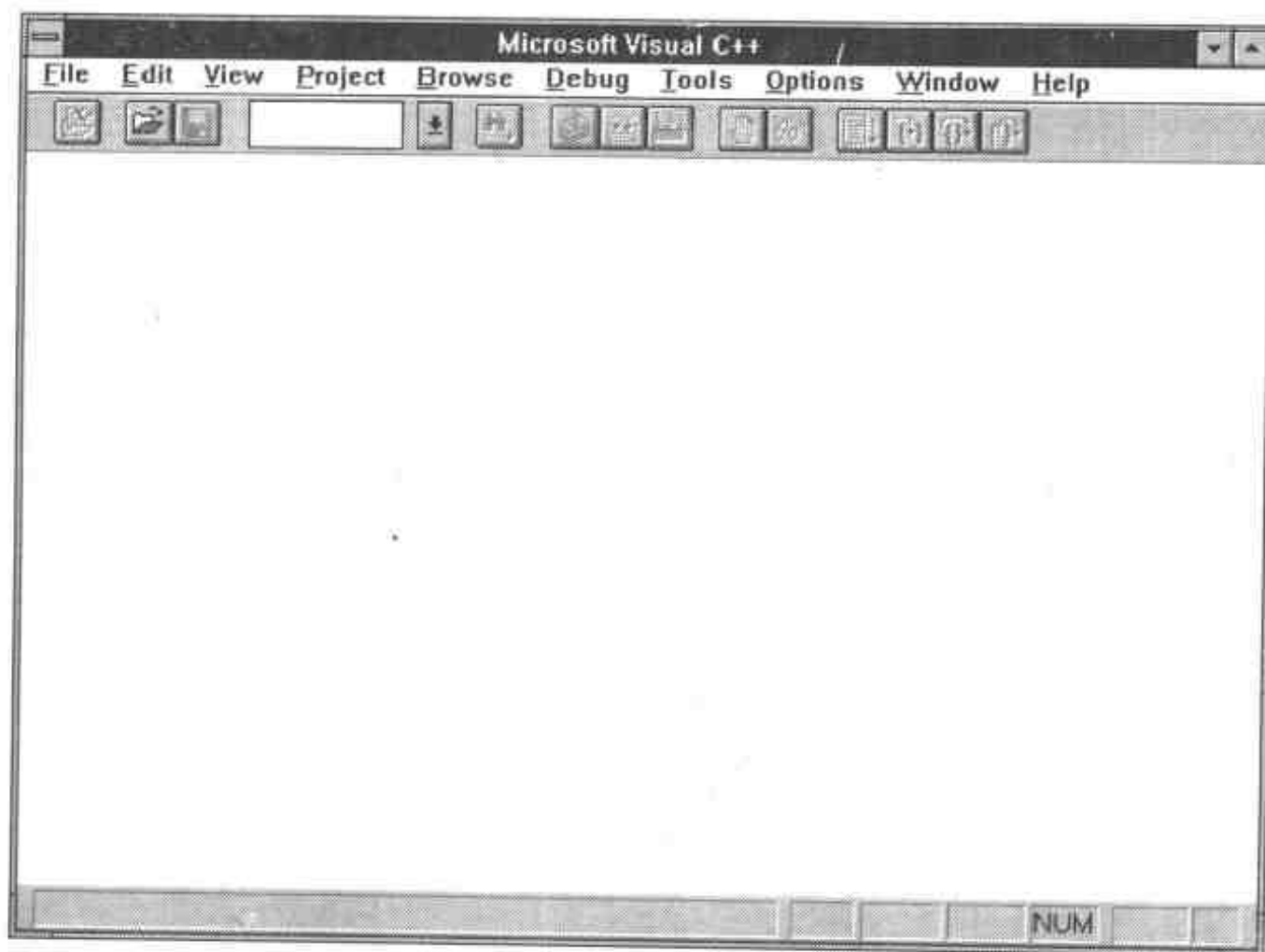


图 2.20 Microsoft Visual C++的初始屏幕

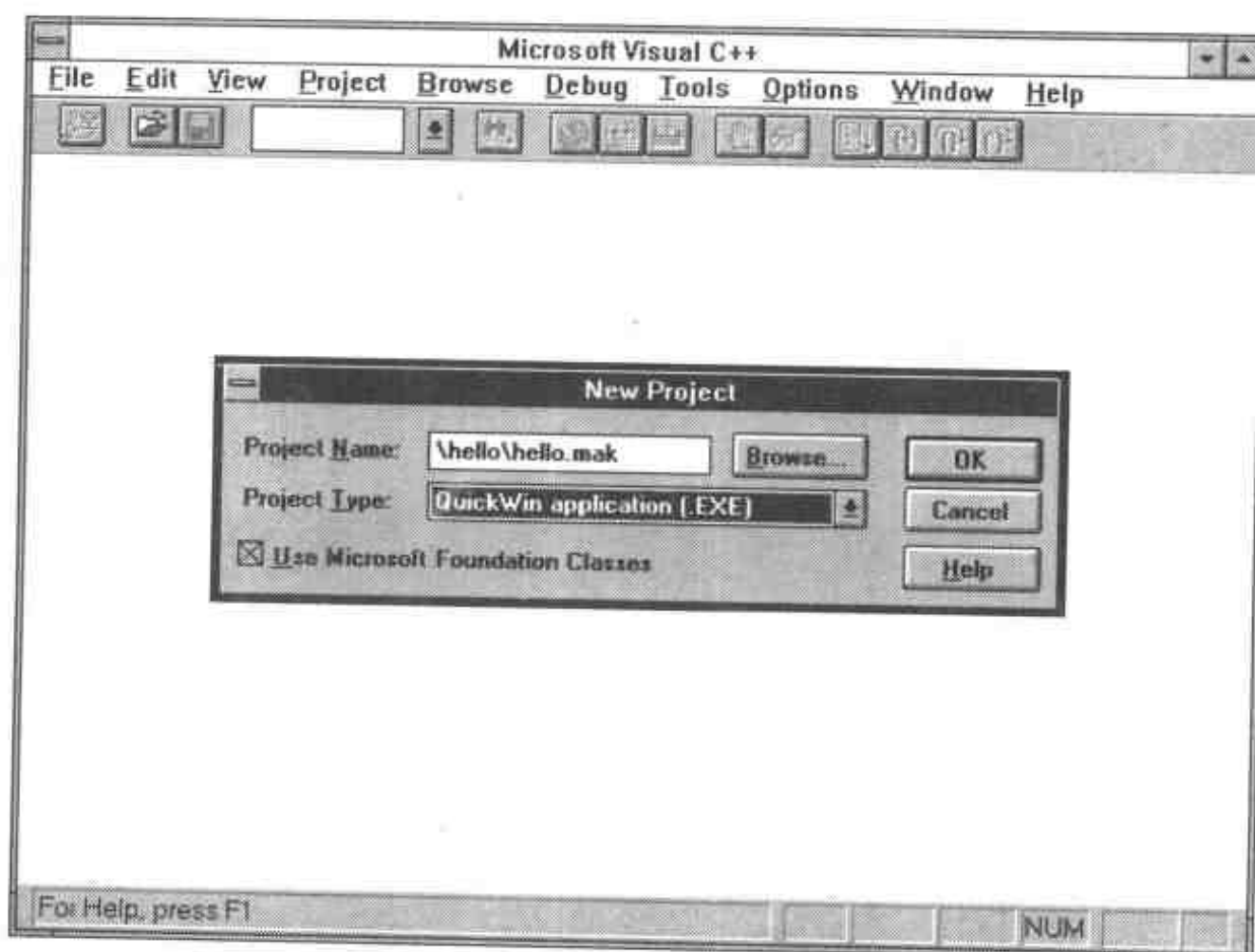


图 2.21 创建Project画面

4. 使用Visual C++的Edit对话框给这个项目中的源文件取名字（见图2.22）。本例中，我们只有一个文件`hello.cpp`。点击Add，把这个名字加入项目中，然后点击Close，告诉Visual C++再没有别的文件了。

5. 选择Options\Project Options, 出现的Project Options对话框如图2.23所示。
点击Compiler按钮, 修改编译选项。

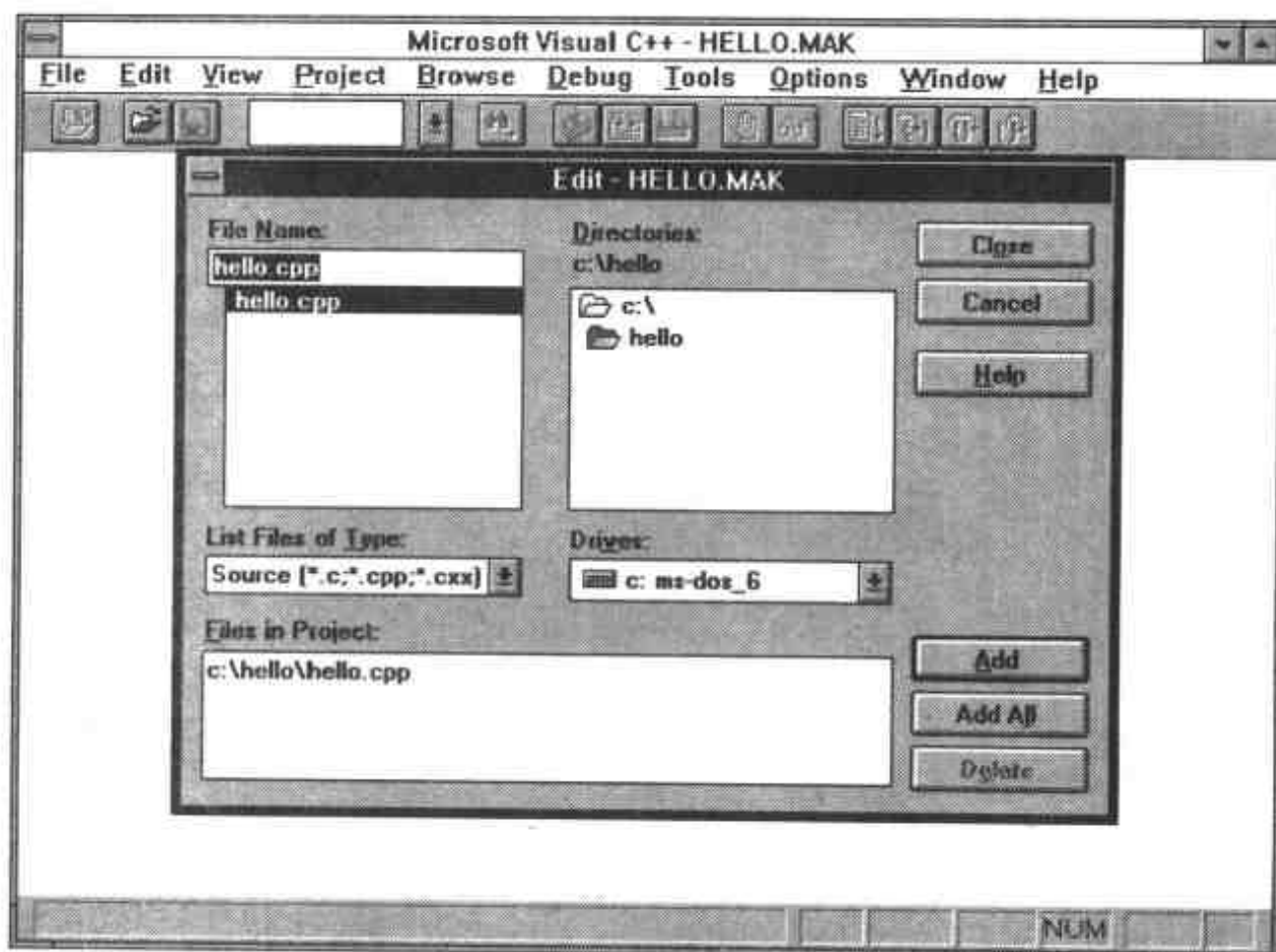


图 2.22 项目编辑对话框

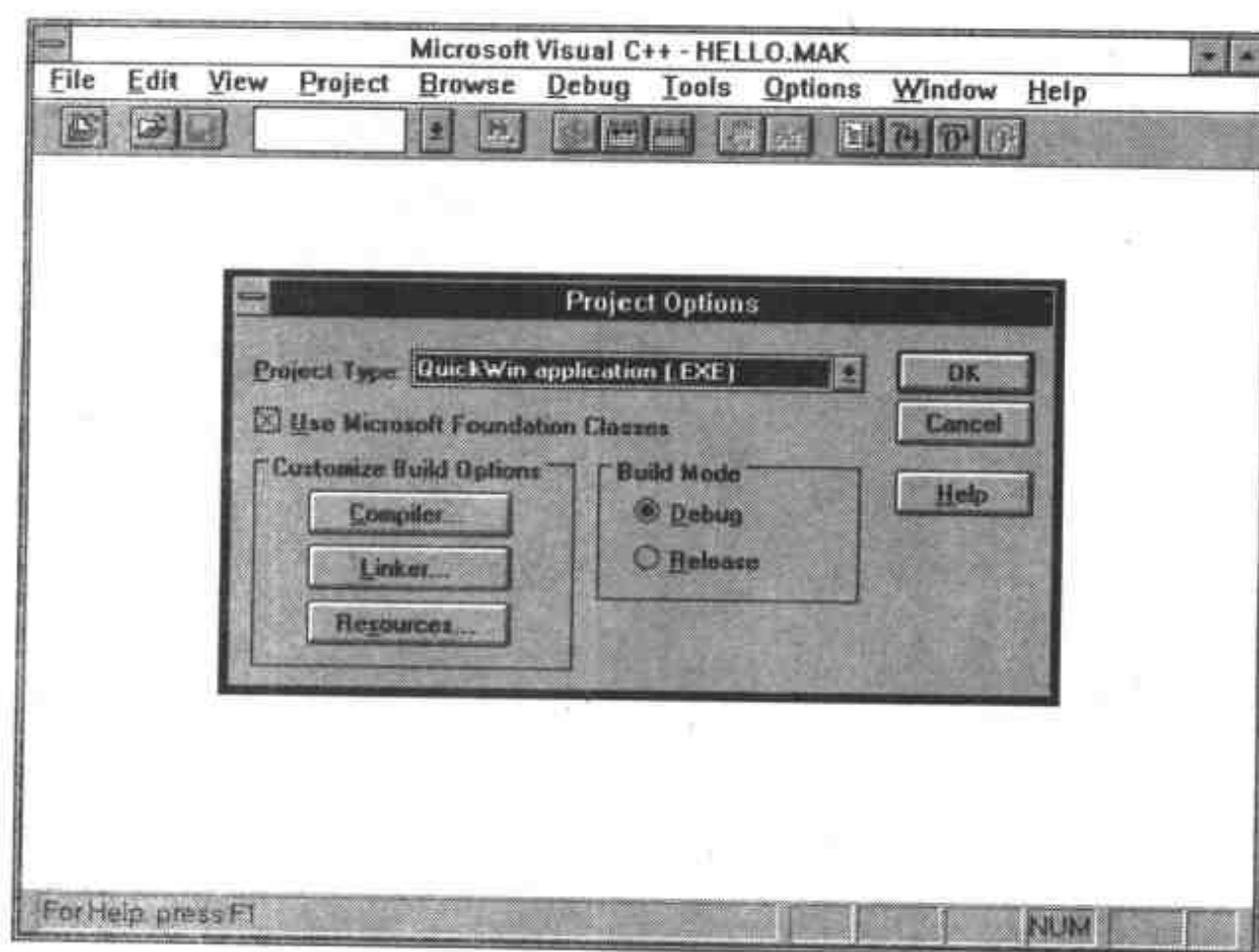


图 2.23 Project Options对话框

6. 在Category选项中，向下移动高亮条到Custom Options项，把警告级别改为4，如图2.24所示。
7. 然后把Memory Model改为大模式（见图2.25）。
8. 点击OK按钮关闭对话框。此时又回到Project Options对话框中。再点击OK退出该

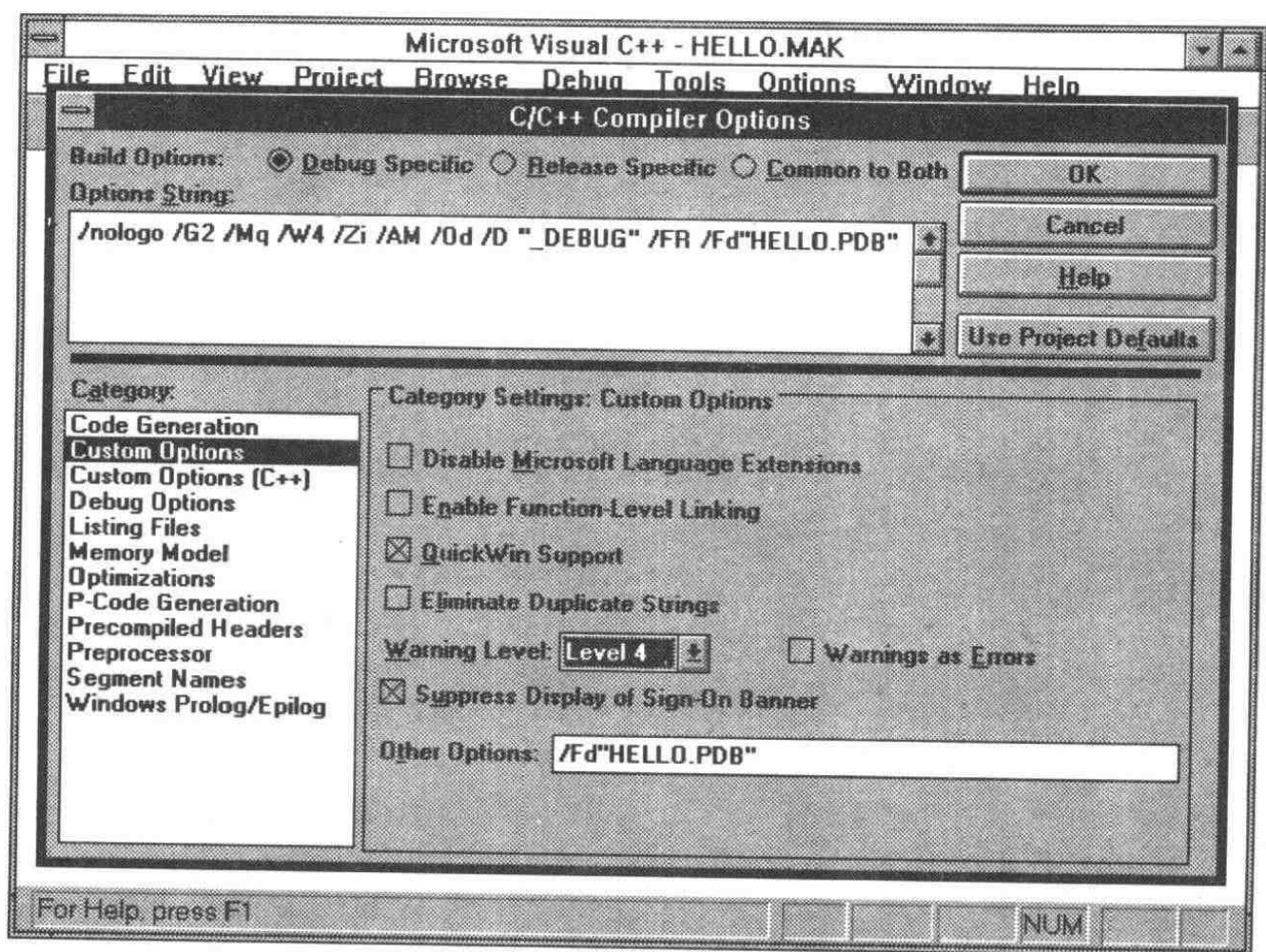


图 2.24 Compiler Options对话框

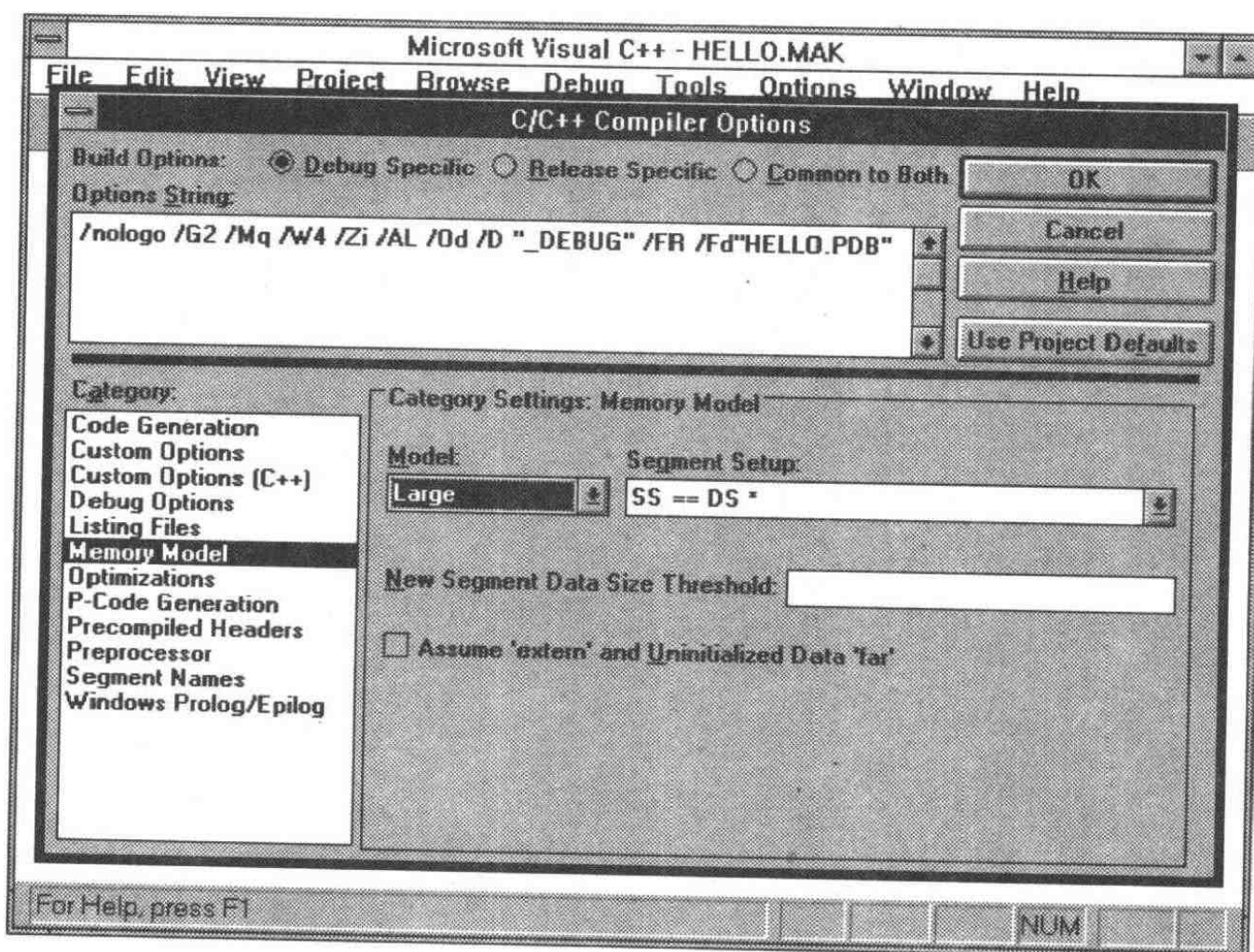


图 2.25 Memory Model选项

对话框。

9. 选取FileNew, 启动一个新程序文件。键入下列代码:

```
#include <iostream.h>
int main()
{
    cout << "Hello World\n";
    return (0);
}
```

结果如图2.26所示。

10. 使用FileSave As菜单项, 给文件取名为hello.cpp。

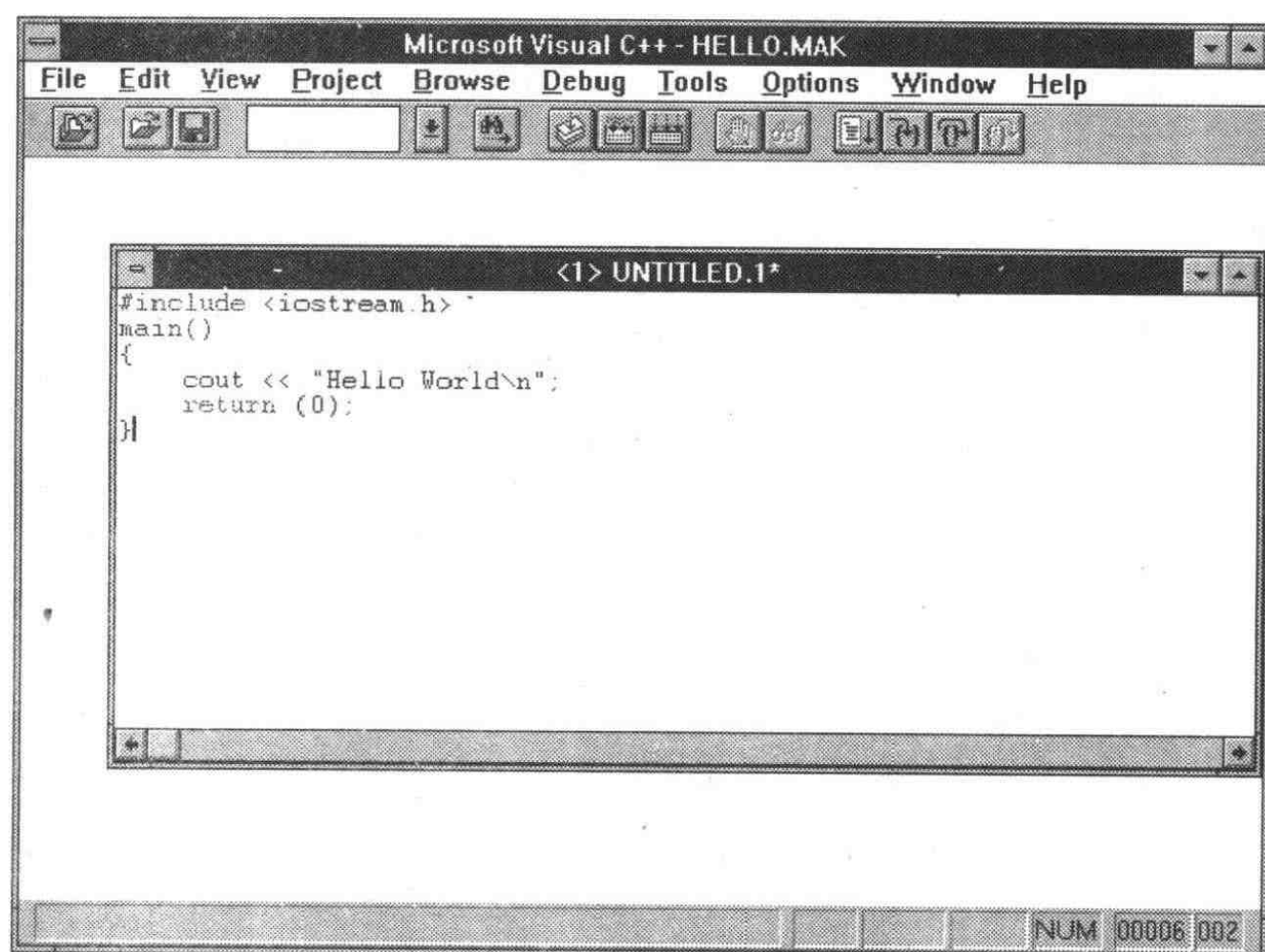


图 2.26 键入了“Hello World”程序的Visual C++

11. 使用ProjectBuild命令, 编译该程序。程序建立时, 编译器将输出相应的信息。此时的屏幕如图2.27所示。

12. 现在可以使用DebugGo命令来运行程序了。程序执行的结果见图2.28。

获取UNIX帮助

大多数UNIX系统都有名为“man pages”的联机文档系统。使用man命令就可以访问该文档。(UNIX用man作为手册“manual”的缩写。)如要得到有关一个具体主题的信息, 可以使用下列命令:

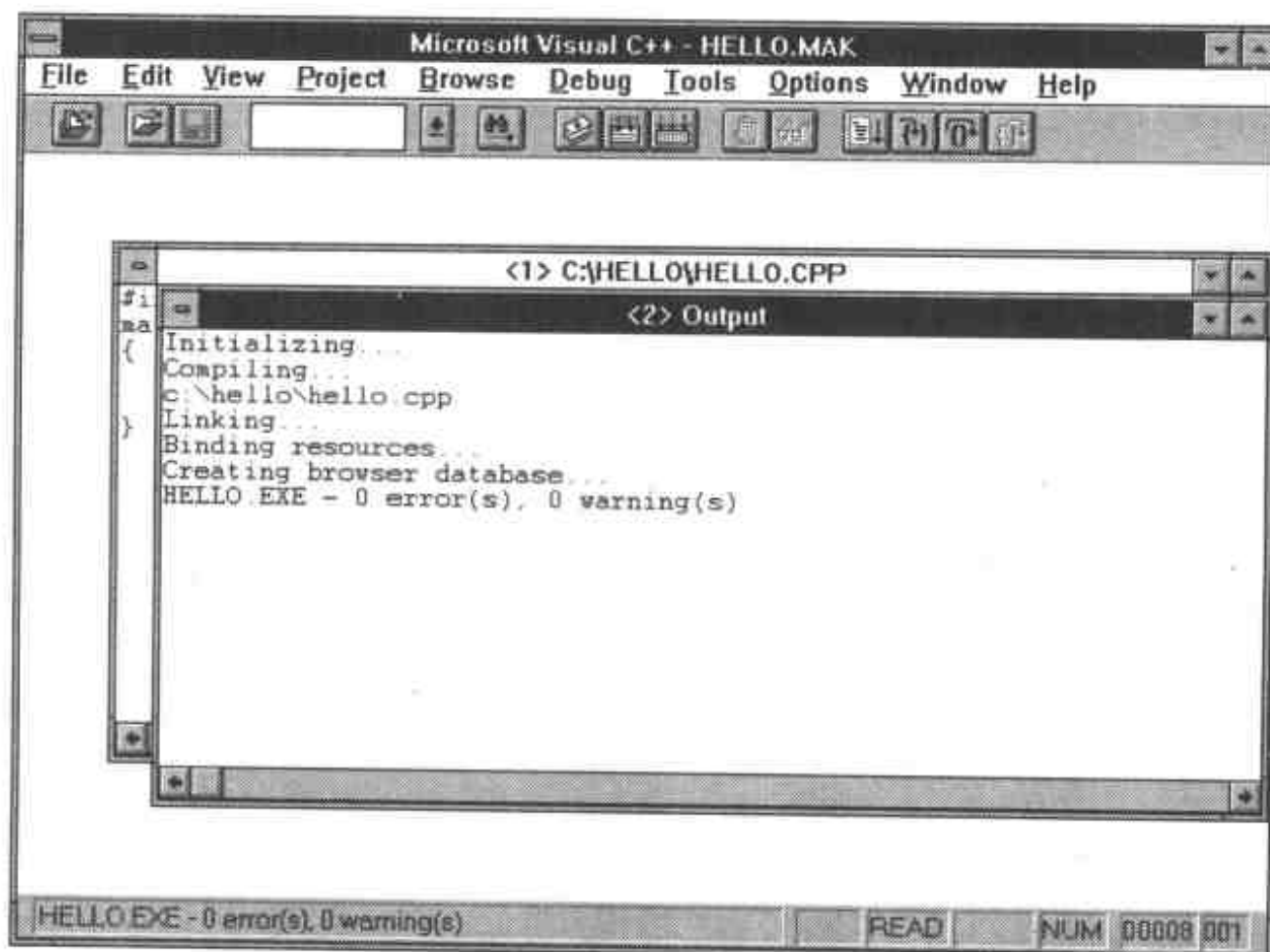


图 2.27 Visual C++建立画面

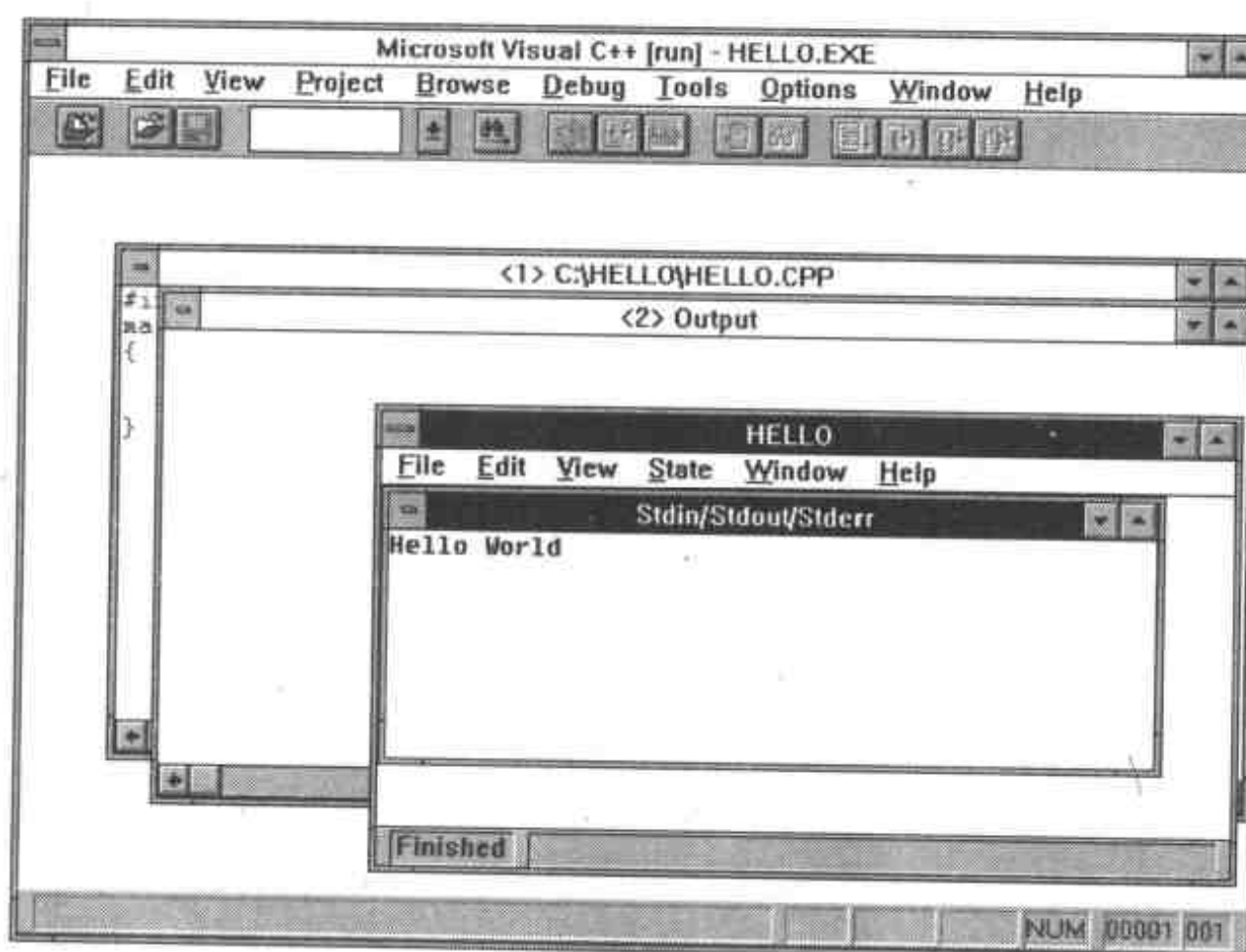


图 2.28 “Hello World” 结果

man 主题

例如，如想得到iostream包内定义的类的信息，可以键入：

man iostream

该命令还有按关键字查找方式：

man -K 关键字

如想得到本标题下含有“output”字的手册页的名字，可以使用如下命令：

man -k output

获取集成开发环境的帮助

像Turbo-C++、Borland C++和Microsoft C++这样的集成开发环境，都有一个Help菜单项。使用这个菜单项可以激活基于超文本的Help系统。

编程练习

练习2.1： 在你的计算机上，键入hello程序，并运行它。

练习2.2： 找几个程序例子，录入并运行之。

本章内容:

- 注释
- C++代码
- 命名风格
- 编码盲从
- 缩进与编码格式
- 清楚
- 简明
- 一致性与组织结构
- 继续读下去
- 小结

第3章

风格

本章讨论如何用良好的编程风格建立一个简单易读的程序。在你还不知道如何编程的时候去讨论编程风格问题，好像有点本末倒置，但实际上编程风格是编程中最重要的部分。风格是辨别真假编程艺术家的试金石。在键入你的第一行代码之前，应该先学习良好的编程风格，这样，你写出的程序的质量才最高。

与通常的看法不同，程序员并没有花大部分时间写程序。花在维护、升级和调试已有代码的时间，比用在编写新代码的时间要多得多。程序维护的时间在飞速增长。从1980年到1990年，典型应用程序中代码的平均行数从2.3万行增长到120万行，完善系统所需的平均时间从4.75年延长为9.4年。

更糟糕的事情是，据软件维护协会1990年年度报告报道，在被调查的管理人员中，有74%的人认为：在他们的部门正使用的系统必须由特定的人来维护，因为其他的人根本不懂这些系统。

大多数软件都建立在已有软件的基础上。最近我完成了12个新程序。其中只有一个是从零做起，另外的11个程序都是修改已有程序后完成的。

程序员认为，编程的目的只有一个，即给计算机提供一组紧凑的指令。其实这不对。仅为机器而写的程序有两个问题：

- 很难判定程序的正确性。因为即使是作者本人，有时也会弄不懂程序的含义。
- 程序的修改和升级很困难。因为维护人员必须花大量的时间去读源代码，了解程序到底执行了哪些操作。

注 释

一个程序最终服务于两个目的：首先，它向计算机提供一组指令；其次，它以明了、易读的方式向程序员描述程序本身做什么。

示例2.1包含了一个明显的错误。这个错误是许多程序员的通病，它带来的麻烦比其它

任何问题都大。

错误就是：程序中没有注释。

一个正常工作的但没有注释的程序如同一个等待爆炸的定时炸弹。早晚会有人修改或升级这个程序，而注释的缺乏会使修改工作难上十倍。但是，一个具有良好注释的简明程序是一件艺术品。学习如何写注释，与学习如何正确地编写代码同样重要。

C++的注释有两种形式。第一种是以/*开头，以*/结尾。这种形式的注释可以跨多行，如下面所示：

```
/* This is a single-line comment. */
/*
 * This is a multiline comment.
 */
```

另一种形式的注释是以//开头，直到该行的结束：

```
// This is another form of comment.
// The // must begin each line that is to be a comment.
```

使用/* */形式的注释的好处是，你可以跨行书写注释，而//形式的注释则必须在每行都写上//。使用/* */形式的注释时如果忘记写*/，会破坏你的代码。

你应该使用哪类注释？当然是能使你的程序尽可能清晰易读的那种。大多数情况下，这要因因人而异。本书中大段多行的注释，使用/* */类型，而//形式用于只占一行的注释。

不管使用哪种类型的注释，重要的是必须要用。“hello world”程序加上注释后如示例3.1所示。

示例3.1 *hello2/hello2.cc*

```
/******
 * hello -- program to print out "Hello World".      *
 *      Not an especially earth-shattering program.  *
 *                                                    *
 * Author: Steve Oualline                             *
 *                                                    *
 * Purpose: Demonstration of a simple program        *
 *                                                    *
 * Usage:                                              *
 *      Run the program and the message appears      *
 *****/
#include <iostream.h>
main()
{
    // Tell the world hello
    cout << "Hello World\n";
```



```
    return (0);  
  
}
```

本例中，开始部分的注释放在一个称为注释框(comment box)的由星号(*)组成的框内。这样做是为了突出较重要的注释，这很像本书中标题所用的粗体字。不太重要的注释不加框。例如：

```
// Tell the world hello  
cout << "Hello World\n";
```

写程序前，必须对要干什么有个明确的思路。最好的办法是用一种清楚易懂的语言把它们写下来。一旦操作过程陈述清楚，就可以把它们转化为计算机程序了。

明白你要做什么，是编程中最重要的部分。我曾经写过满满两页纸注释，来描述一个复杂的图形算法。在编写程序代码之前，我两次修改了这段注释。实际上最后的程序只占半页。因为我把思路整理得很清楚（也很幸运），所以这个程序一次通过。

你的程序读起来应该像一篇文章，它应该尽可能地清楚易懂。良好的程序设计风格来自于经验和实践。后面将提到的风格就是作者多年编程经验之谈。你可以借鉴并形成自己的风格。这些不是规则，仅仅是建议。规则只有一条：尽可能地让你的程序清楚、简明、简单。

程序的开头是一个注释块，其中包含有关该程序的信息。给这段注释加框，以使它更醒目。下面我们列出在程序的开始部分应该说明的内容。并不是所有的程序都要包含所有的部分，要根据具体情况选择使用。

开头

第一段注释中应该包含程序名，还应该有该程序的简短描述。你可能已拥有最令人惊异的程序，它能解决世界上的所有问题，但是如果没有人知道它用于何处，那它就一无用处。

作者

为了设计这个程序你解决了很多的麻烦。为此而自豪吧。而且，如果以后其他人要修改这个程序，他或她也可以向你索取相关的信息，并得到你的帮助。

目的

你为什么编制这个程序？它干什么？

用法

这一段将简单叙述程序如何运行？理想情况下，每个程序都应该有叙述使用方法的若干文档，但事实并非如此。Oualine关于文档的法则说：有90%的程序没有文档。剩余的10%中，又有9%的程序，其文档的修改速度赶不上程序的修改速度，所以这些文档是完全无用的。最后，实际上只有1%的程序有文档，而且文档的修改与程序的修改是同步的，这些文档是用日语书写的。

排版符号的不足

排版时你可以使用不同的字型和字号，用粗体和斜体来代表文本中不同的部分。但设计程序时，只能使用占一个位置的字符。程序员可以用各种巧妙的方法克服这方面的不足。

下面是几种不同的注释方法:

[illegible]

为了避免成为Oualline文档法则的牺牲品，请把文档与程序放到一起。

出处

拷贝程序是合法的程序设计方法（如果你没有违反版权法的话）。实际上，只要你拿到了程序，采用的是什么方式倒无关紧要，但是，贪功是不可取的。在这一部分中，你应该提及你所拷贝的程序的原作者。

文件格式

列出程序中读或写的文件，并简要描述它们的格式。

限制

列出本程序的所有局限或限制。例如，数据文件的格式必须要正确；程序不检查输入错误等。

修改史

这一部分含有一张表，指出是谁、在何时修改了这个程序的哪些地方。许多计算机有源控制系统（UNIX是RCS和SCCS；MS-DOS/Windows是MKS-RCS、PCVS），这些系统可以为你记录这些信息。

错误处理

如果程序发现错误，它该怎么办？

注释

包括一些特殊的注释，或是注释中没有的一些信息。

程序开始部分的注释格式依赖于编程环境的需要。例如，如果你是学生，导师会要求你在程序开头写清作业号、姓名、学号和其它信息。在工业界，应该包含项目号或是部分号。

程序员需要了解的有关程序的内容也应该在注释中解释，其它就没有了。一个程序的注释有可能过多（这很少发生，但确实出现过）。当你确定开头注释的格式时，一定要保证所包含的每个字都有必要。

插入注释——简单的方法

如果你使用的是UNIX编辑器vi，那么把下面的内容放到你的.exrc文件中，这样画框时就容易多了。

```
:abbr #b /*****  
:abbr #e *****/
```

这两行定义了vi缩写词**#b**和**#e**，所以在块首键入**#b<return>**时，将出现下列字符串：

```
/* **** */
```

（这个串是注释框的首行）。键入**#e<return>**将显示注释框的结束行。星号的个数要仔细计算，以保证注释框的结尾与**tab**键所停的位置在同一列上。

C++代码

真正的程序代码包含两部分：变量和可执行的指令。变量用来存放程序中使用的数据。可执行的指令告诉计算机该对数据进行什么操作。C++类是数据与操纵数据的指令的结合。它们提供了封装指令和数据的方便方法。

变量是计算机内存中存放一个数值的地方。C++用变量名来标识这个地方。名字的长度可以任意，所以最好挑选一个意义明确的名字。（实际上，名字长度还是有限制的。但这个数很大，你可能永远也不会遇到这个麻烦）。C++中的每个变量都必须被说明。（变量说明将在第9章“变量作用域和函数”中讨论。）下面的说明告诉C++你准备使用三个整型（int）变量，名字分别为p、q和r：

```
int p,q,r;
```

但读者并不知道这三个变量干什么用。它们可以表示插头的角度，或表示Space Invaders游戏中plasma bolt的位置和加速度。应避免缩写词。过分的缩写很难读，也不易理解。（例如，Exs. abb. are diff. to rd. and hd. to ustnd.）

现在考虑另一个说明：

```
int account_number;
int balance_owed;
```

我们知道，我们正处理一个记帐程序，但我们还需要使用更多的信息。例如，balance_owed是以元还是以分为单位？如果在每个说明之后加上注释，解释我们的目的就好多了。

```
int account_number;    // Index for account table
int balance_owed;      // Total owed us (in pennies)
```

在每个说明的后面加一个注释，实际上就建立了一个小字典。在字典中，我们定义了每个变量名的意义。由于每个变量的说明都在一个已知的位置，所以很容易查到一个名字的具体含义。（程序设计工具，如编辑器、交叉引用器和grep，也可以帮助你快速地找到一个变量的定义。）

单位很重要。有一次我被要求修改一个程序，这个程序把土地数据文件从一种格式转换为另一种格式。在程序中到处使用了许多不同的长度单位，而且变量说明中没有注释。我努力想找出程序中使用的是什么单位，但这是不可能的。最后，我放弃了，并在程序中加上了下面这段注释：

```
/* *****
 * Note: I have no idea what the input units are, nor
 *       do I have any idea what the output units are,
 *       but I have discovered that if I divide by 3
 * ***** */
```

```

*      the plots look about the right size.      *
*****/

```

很多刚入门的程序员会犯这样一个错误，他们只描述代码，但不描述变量。例如：

```
int top_limit;      // Top limit is an interger [bad comment]
```

从代码中很容易看出`top_limit`是一个整型量。我想知道的是`top_limit`是什么。请告诉我：

```
int top_limit;      // Number of items we can load before losing data
```

你应该抓住一切机会，确保你的程序清楚易懂。不要耍小聪明。耍小聪明只会产生不可读且不可维护的程序。从程序本身来看，它非常复杂。你所做的任何降低这种复杂性的事情，都会使程序变得更好。考虑一下下面这行由一位很聪明的程序员书写的代码。

```
while ('\\n' != *p++ = *q++);
```

读者看一遍后，几乎不可能说出这么多乱七八糟的符号是什么意思。像下面这样写就很合适：

```
while (1) {
    *destination_ptr = *source_ptr;
    ++destination_ptr;
    ++source_ptr;
    if (*(destination_ptr-1) == '\\n')
        break; // exit the loop if done
}
```

虽然后一种写法较长，但它更清楚更易懂。即使不太熟知C++的程序设计新手，也能说出这段程序是把数据从源地址移到目标地址中。

计算机并不在意你使用哪一种写法。好的编译器会为两种写法产生相同的机器代码。程序员才是其中的受益者。

命名风格

名字中可以包含大小写字母。本书中，表示变量名时全部使用小写字母（如`source_ptr`、`current_index`）。全部大写字母表示常量（如`MAX_ITEMS`、`SCREEN_WIDTH`）。这是大多数C和C++程序所遵从的惯例。

许多刚入门的程序员使用大小写字母混合的名字（如`RecordsInFile`）。有时他们用首字母大写来表示这个变量的有关信息。例如，`recordsInFile`可能表示这是一个局部变量，而`RecordsInFile`将表示这是一个全局变量。（有关局部变量和全局变量的详细内容，请见第9章“变量作用域和函数”）。

使用哪种命名约定由你决定。这是个人习惯问题，而不是风格问题。但是，使用固定

的命名风格很重要。本书中，我们选用第一种风格，小写表示变量名，大写表示常量，这种用法将贯穿全书。

编码盲从

计算机科学家已设计出许多种编程方法。包括结构化程序设计、自顶向下的程序设计和少用goto语句的程序设计。每种方法都有自己的追随者或崇拜者。我之所以使用盲从一词，是因为人们总是被告之要遵从一些规则，但不知道其中的缘由。例如，盲目追求少用goto语句的人从来不使用一个goto语句，即使在该使用的时候也不用。

本书中介绍的规则是多年编程实践之心得。我发现，按照这些规则，我可以写出更好的程序。读者不必盲目地遵从这些规则。如果你能找到更好的一套规则，一定要使用它。（如果它真的能起作用，请给我寄封短信。我也很乐意使用）。

缩进与编码格式

为使程序易读，大多数程序员都在他们的程序中采用缩进的方式。C++程序的一般规则是每一个新块或是条件句都要缩进一层。示例3.5中有三层逻辑，各自都有缩进层。while语句在最外层。while所包含的语句是下一层。if (break)内的语句在最内层。

有两种形式的缩进写法，究竟哪种更好，仍在争论不休。第一种是简短形式：

```
while (! done) {
    cout << "Processing\n";
    next_entry();
}
if (total <= 0) {
    cout << "You owe nothing\n";
    total = 0;
} else {
    cout << "You owe " << total << " dollars\n";
    all_totals = all_total + total;
}
```

在这种写法中，大多数花括号与语句放在同一行中。另一种写法是把花括号单独放在一行：

```
while (! done)
{
    cout << "Processing\n";
    next_entry();
}
if (total <= 0)
```



```

{
    cout << "You owe nothing\n";
    total = 0;
}
else
{
    cout << "You owe " << total << " dollars\n";
    all_totals = all_total + total;
}

```

这两种写法都很普遍。你可以随意使用。本书使用简短形式。

缩进量由程序员自己决定。常见的为2个、4个和8个空格。研究表明，缩进4个空格可使程序最易读。你可以选用你一直采用的缩进量。

清 楚

程序读起来应该像一篇技术论文。它应该分为节和段。过程自然成为一个节。你应该把代码组织为段。在段首写一个主题注释，并用空行把各段区分开来，这不失为一个好主意。例如：

```

// poor programming practice
temp = box_x1;
box_x1 = box_x2;
box_x2 = temp;
temp = box_y1;
box_y1 = box_y2;
box_y2 = temp;

```

下面的写法更好一些：

```

/*
 * Swap the two corners
 */

/* Swap X coordinate */
temp = box_x1;
box_x1 = box_x2;
box_x2 = temp;

/* Swap Y coordinate */
temp = box_y1;
box_y1 = box_y2;
box_y2 = temp;

```

简 明

你的程序应该简单。下面是一些经验之谈：

- 每个函数不要长过一两页。（见第9章“变量作用域和函数”）。如果它变得太长，可以把它分为两个更简单点的函数。之所以要这样，是因为人脑的短期记忆只能记这么多。三页纸的内容是人一口气能记住的最大量。
- 避免使用像多个嵌套的if语句这样的复杂逻辑。你的代码越复杂，需要的缩进层也就越多。当你的代码接近右边界时，就该考虑把代码划分为多个过程了，从而减小代码的复杂程度。
- 你是不是读过这样的句子，作者写了又写，一个句子接着一个句子，中间用“和”连起来，好像不明白几个短句子可能表达得更清楚。这很烦人，对吗？

C++语句永远不应该写个没完。应避免使用长句子。如果一个等式或是公式有可能占据一两行以上，就要把它分成两个更短些的式子。

- 把大的单个的源代码文件分为多个更小些的文件。（有关使用多个文件进行程序设计的问题，请看第23章“模块化程序设计”）。一般地，我喜欢把我的文件保持在1500行以下。否则的话，编辑和打印都很困难。
- 使用类时（见第13章“类的初步讨论”），每个模块中放一个类。
- 最后，最重要的一条规则是：让你的程序尽可能地简单易懂，即使它违背了某些规则也无关紧要。我们的目的是要清楚，本章中列出的这些规则都是为了帮你达到这个目的。如果某条规则有悖于此，那就不去管那条规则。我曾见过一个程序，它只有一条语句，该语句长20多页。但是，因为程序本身的特性，这条语句很简单易懂。

一致性与组织结构

编制高质量程序时，良好的风格是唯一的要素。一致性也是一个因素。本书英文版的结构是目录在前，索引在后。几乎所有的书都有相似的结构。这种一致性使得在索引中找一个词或是在目录中找章节标题很容易。

不幸的是，程序设计人员已开发了多种代码设计风格。每种都有各自的优缺点。编程小组内进行高效编程的诀窍是：选择一种风格，并坚持使用它。这样能避免当使用不同风格设计程序时所带来的问题以及混乱。

良好的风格固然好，但一致性更好。

继续读下去

本章我们仅讨论了一些基本风格，后面几章将继续这个话题。在学习语言元素的同时，

我们将继续介绍新的编程风格。

小 结

程序应该简明易读。它是一组计算机指令，但也描述算法和其中使用的数据。每件事都应该有带注释的文档。注释有两个目的。首先，它们为该程序的维护人员描述程序；第二，注释可以帮你记录你的所做所为。

班级讨论题1：建立班级作业的风格表。讨论程序中该包含哪些注释，为什么？

班级讨论题2：分析一个已有程序的风格。这个程序这样写是不是很清楚很易懂？是否可以改进它的写法？

练习3.1：把本书中的其它练习题看一遍，给这些程序写注释块。这样做有几个目的。第一，可以给你提供写注释的练习机会；第二，这可以减轻老程序员的歉疚：“我没有时间写注释啊。”

本章内容:

- 程序元素
- 程序的基本结构
- 简单表达式
- cout输出类
- 变量和存储
- 变量说明
- 整数
- 赋值语句
- 浮点数
- 编程练习
- 答疑

第4章

基本说明 和表达式

程序元素

如果你要盖楼，需要准备两件事：砖及蓝图，蓝图告诉你如何把砖垒在一起。在计算机上设计程序也需要两件事：数据（变量）和指令（代码）。变量是构筑程序的基本材料。指令告诉计算机该对变量进行什么操作。

注释用来描述变量和指令。它们是作者为程序而作的注解，所以应该清楚易读。计算机忽略掉所有的注释。

在建筑上，开始一项工程之前，我们必须预订所需的原材料：“我们需要500块大砖，80块半大的砖和4块铺路石”。类似地，在C++中，在使用变量之前必须先说明它们。你必须为每块“砖”命名，并告诉C++要使用的砖是什么类型的。

变量定义后就可以使用了。建筑上的基本结构是房间。把许多房间组合在一起就形成了一座建筑物。在C++中，基本的结构是函数。函数可以组合为程序。

初出茅庐的建筑人员不会先去盖帝国大厦，而先从一居室的房子着手。本章中，你应该集中精力学习编写简单的、单函数的程序。

程序的基本结构

程序的基本元素是数据说明、函数和注释。让我们看看这些元素是如何组织为一个简单的C++程序的。

单函数程序的基本结构为：

```
/******  
* 首段注释 *
```

```
*****/
```

数据说明

```
main()
{
```

可执行语句

```
    return(0);
}
```

首段注释告诉程序员有关该程序的所有信息。数据说明 (data declarations) 描述该程序要使用的数据。

唯一的函数是main。这个函数的名字是特殊的，因为它是被调用的第一个函数。其它的任何函数都直接或间接地从main中调用。main函数的开头是：

```
main()
{
```

它的结尾是：

```
    return(0);
}
```

return(0);这一行告诉操作系统 (UNIX或是MS-DOS/Windows) 程序正常退出 (status=0)。非零的status表明有错误——返回值越大，表明错误越严重。一般地，1表示最简单的错误，例如找不到文件或命令行语法错等。

下面我们看看“Hello World”程序 (示例3.3)。

程序的开头是一个由/*和*/围住的注释框。然后是下面这行：

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

这条语句告诉C++你准备使用一组标准类，这组类名叫I/O流类。这是一种类型的数据说明*。随后你可以使用这个包中的cout类。（我们将在第13章“类的初步讨论”中定义一个更完整的类。在我们深入了解cout之前，我们将一直把cout看作是一个“黑匣子”，它把数据传送给控制台。）

main程序包含如下指令：

```
cout << "Hello World\n";
```

* 技术上，该语句可以从include文件中取一组数据说明。第10章“C++预处理器”将讨论include文件。

这是一条可执行语句，它告诉C++在屏幕上显示出"Hello World"信息。C++使用分号结束一条语句，这很像我们用句号表示一个句子的结束。与BASIC这样的面向行的语言不一样，行的结束并不意味着语句的结束。本书中的句子可以写在若干行内——行尾只看作是分隔词的一个空格。C++采用同样的方法。一个单句可以扩展在多行内。同样地，你可以把几个C++语句放在同一行，正如同你可以在同一行写若干个句子一样。但是大多数情况下，如果每个句子都从新行开始，则程序将更易读。

我们使用标准类cout（控制台输出）来输出信息。一个标准类通常是一个有用的C++对象，它已被定义在标准库中。一个库就是类、函数和数据的一个集合，它们被组织在一起，可以重用。标准库包含了用于输入、输出、排序、高级数学计算和文件操作的类及函数。有关库函数和标准类的完整列表请参见C++参考手册。

“Hello World”是最简单的C++程序之一。它没有计算，仅仅是把一条信息显示到屏幕上。这是个开始点，一旦你掌握了这个简单的程序，实际上就已学到了许多东西。因为这个程序并不像看上去那样简单。但只要你能让它运行，你就能编出更复杂的程序来。

简单表达式

计算机不仅能打印字符串，它们还能执行计算。表达式就是用来说明简单计算的。C++有五种简单运算符，如表4.1中所列。

表4.1 简单运算符

运算符	含 义
*	乘
/	除
+	加
-	减
%	模（除后取余）

乘（*）、除（/）和模（%）均比加（+）和减（-）优先。括号用来表示一组项的计算。所以：

$(1 + 2) * 4$

的结果为12，而

$1 + 2 * 4$

的结果为9。

示例4.1中的程序计算表达式 $(1 + 2) * 4$ 的值。

示例4.1 Simple Expression

```
main()
{
    (1 + 2) * 4;
    return(0);
}
```

虽然我们计算了结果，但我们并没有用它。（该程序将产生一条“null effect”的警告信息，指出在程序中含有书写正确、但不起作用的语句）。

设想一下，盖房子时，如果我们对工人说：“推上你的手推车，在卡车与房基之间来回走”，

“你是不是想让我用手推车运砖？”

“不，只是来回地走。”

那位工人肯定如丈二和尚摸不着头脑一般。

所以你必须输出计算的结果。

cout输出类

标准的class变量cout用于向控制台输出数据。随后我们将在第13章“类的初步讨论”中学习类的概念。但现在，我们只知道运算符<<告诉C++要输出的内容。所以语句：

```
cout << "Hello World\n";
```

告诉C++取来字符串“Hello World\n”，并把它输出到控制台。多个<<操作符可以一起使用。例如，下面两行语句输出的内容相同：

```
cout << "Hello World\n";
cout << "Hello " << "World\n";
```

采用这种方法也可以输出表达式的值，如：

```
cout << "Half of " << 64 << " is " << (64 / 2) << "\n";
```

运行时，它将在控制台输出如下信息：

```
Half of 64 is 32
```

* 技术上，<<是左移运算符；但是cout类可以重载这个运算符，而把它用作输出运算符。（有关I/O类的详细内容请参见第6章“文件输入/输出”，有关重载定义的内容请参见第18章“运算符重载”。）

注意，我们在“Half of”中的“of”后面加了一个空格。在“is”的两边各加了一个空格。输出时这些空格都是需要的，它们把数字与字母隔开。假定我们不加这些空格，代码如下：

```
// Problem code
cout << "Half of" << 64 << "is" << (64 / 2) << "\n";
```

一眼看上去这行代码完全正确。在每个数字的两边都有空格。但这些空格不含在任何字符串中，所以不会被输出出来。这行代码的执行结果如下：

```
Half of64is32
```

忽略了所需的空格是编写初稿时常犯的错误。请记住，只有引号内的文本才被输出。

变量和存储

C++允许你把值存放到变量中。每个变量都由一个变量名 (*variable name*)来标识。

另外，每个变量都有一个变量类型 (*variable type*)。类型告诉C++这个变量将来的用法以及它将保存哪种数（实数、整数）。

名字由字母或下划线（_）开头，后跟任意个字母、数字或下划线。大写字母与小写字母不等同，所以名字“sam”“Sam”和“SAM”是指三个不同的变量。为避免混淆，最好使用不同的名字，而不要依靠大小写来区分。

大多数C++程序员使用全小写变量名。有些名字，如int、while、for和float在C++中具有特定的含义，它们被称作保留字 (*reserved word*)。它们不能用作变量名。

下面是一些变量名实例：

```
average    // average of all grades
pi         // pi to 6 decimal places
number_of_students // number of students in this class
```

下面的字符串不是变量名：

```
3rd_entry  // Begins with a number
all$done   // Contains a "$"
the end    // Contains a space
int        // Reserved word
```

应避免取相似的变量名。例如，下面的变量名取得很糟糕：

```
total      // total number of items in current entry
totals     // total of all entries
```

较好的变量名如下：

```
entry_total    // total number of items in current entry
all_total      // total of all entries
```

变量说明

在C++中，变量使用之前一定要先在说明语句 (*declaration statement*)中说明它。没有说明的变量不能使用。

变量说明有三个目的：

1. 定义了变量名。
2. 定义了变量类型（整型、实型、字符型等）。
3. 向程序员描述该变量。

变量answer的说明如下：

```
int answer;    // the result of our expression
```

关键字int 告诉C++这个变量中可以放一个整型值。（整型在后面定义。）变量名是answer。分号表示语句结束。注释用来定义这个变量，它是给程序员的信息。

变量说明的一般形式是：

```
类型 名字;    // 注释
```

类型 (*type*) 指的是C++变量类型的一种 (int、float等)。名字 (*name*) 是任意的有效变量名。注释解释该变量是什么，它用于什么目的。变量说明位于程序开头，在main()一行的前面。（在第9章“变量作用域和函数”中，你将学习如何在其它地方说明一个局部变量。）

整 数

有一种变量类型是整型。整型（也叫整数）没有小数部分或是小数点。如1、87和-222都是整数。8.3就不是整数，因为它包含一个小数点。整型说明的一般形式是：

```
int 名字;    // 注释
```

8位的计算器只能处理99,999,999到-99,999,999之间的数。如果你想让99,999,999加1，则会出现上溢错误。计算机也有类似的限制。整数的限制依实现的不同而不同，就是说，电脑不同，整数的限制也不同。

计算器使用十进制数字（0~9）。计算机使用二进制数字（0~1），我们称之为位 (*bit*)。8个位组成一个字节 (*byte*)。存储整数的位数依机器的不同而不同。打印时二进制数将转换为十进制数。

在大部分UNIX机器上，整数是32位（4个字节）的，它们表示的范围为2,147,483,647



($2^{31}-1$) 到 -2,147,483,648 (-2^{31})。在PC机上的Turbo-C++系统下, 只用16位 (2个字节) 来表示整数, 所以表示的范围为32,767 ($2^{15}-1$) 到 -32,768 (-2^{15})。

问题4.1: 下列代码可以运行在UNIX机器上, 但在PC机上运行失败。

```
int zip;      // zip code for current address
.....
zip = 92126;
```

它为什么会失败? 在PC机上运行时的结果如何?

赋值语句

通过赋值语句 (assignment statement) 可以给变量赋一个值。变量在使用之前必须先说明。例如:

```
int answer;    // Result of a simple computation
```

然后, 该变量可以用在赋值语句中, 如:

```
answer = (1 + 2) * 4;
```

等号 (=) 左边的变量 `answer` 被赋以等号右侧表达式 $(1 + 2) * 4$ 的值。分号结束该语句。

当你说明一个变量时, C++ 就为该变量分配一个存储位置, 并放入一个未知值。你可以认为, 说明就是建立一个存放数据的盒子。开始时, 这个神秘的盒子内放的是一个未知量。如图4.1A所示。赋值语句计算表达式的值, 然后把这个值放到盒子中, 如图4.1B所示。

赋值语句的一般形式为:

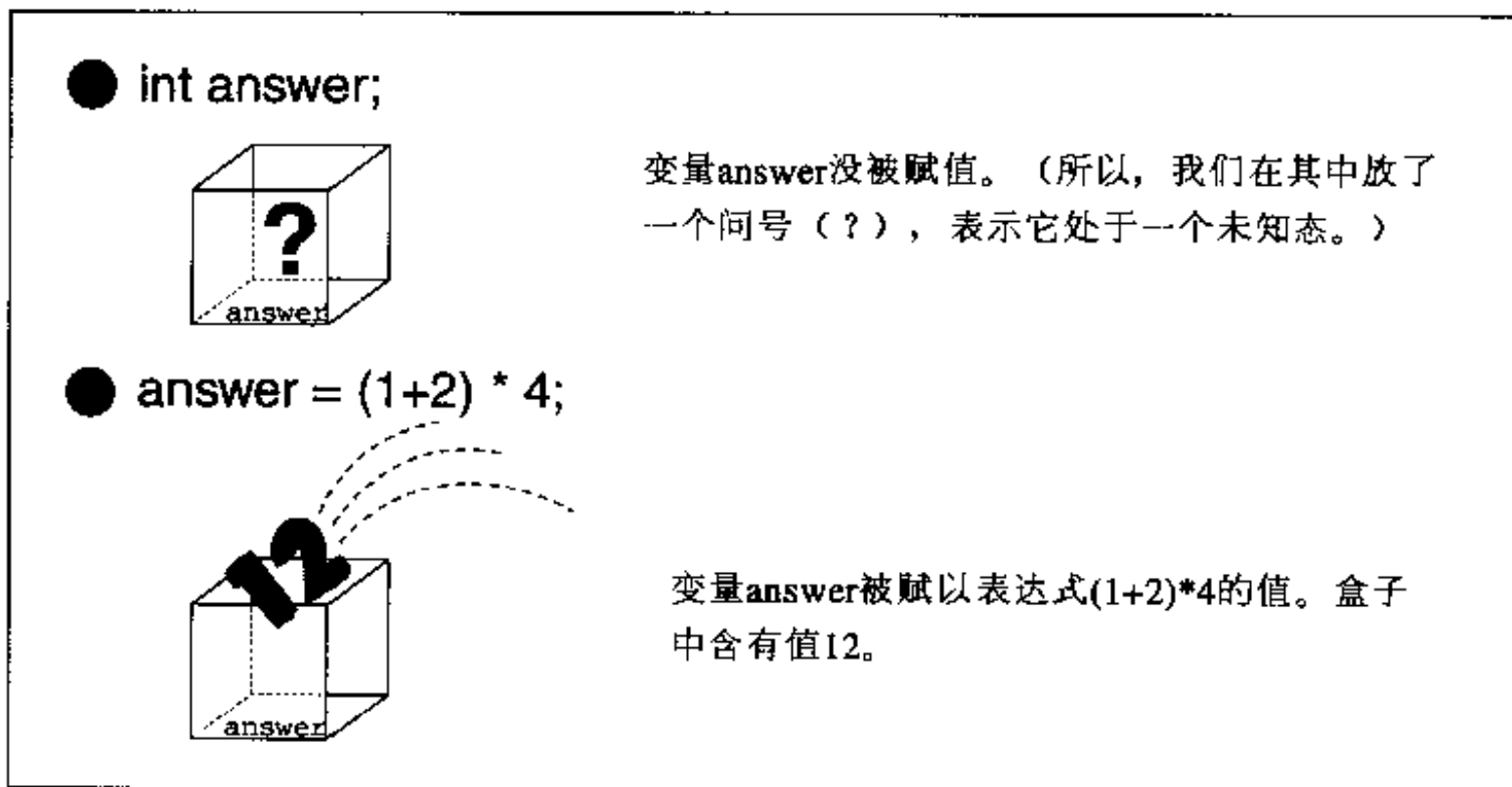


图 4.1 说明和赋值语句

变量 = 表达式;

等号(=)表示赋值, 而不是相等的意思。

在示例4.2中, 变量term中存储了一个整数值, 该值用在后面的两个表达式中。和表达式一样, 变量也可以使用输出运算符<<来输出, 所以我们用这个运算符来检验计算结果。

示例4.2 tterm/tterm.cc

```
#include <iostream.h>
int term;          // term used in two expressions
main()
{
    term = 3 * 5;
    cout << "Twice " << term << " is " << 2*term << "\n";
    cout << "Three times " << term << " is " << 3*term << "\n";
    return (0);
}
```

浮点数

实数带有小数部分。因为它内在的存储方式, 所以又被称做浮点数(*floating point number*)。5.5、8.3和-12.6都是浮点数。C++通过判定有无小数点来区分浮点数和整数, 所以5.0是一个浮点数, 而5是一个整数。浮点数必须包含一个小数点。如3.14159、0.5、1.0和8.88都是浮点数。

虽然我们有可能忽略小数点以前的数字, 把0.5写做.5, 但额外的0会使浮点数更清楚。用12.来表示12.0时也有类似的问题。浮点数零应该写为0.0。

另外, 浮点数可以包含指数形式 $e \pm \text{exp}$ 。

例如, 1.2e34表示 1.2×10^{34} 。

浮点数说明的一般形式为:

float 变量; // 注释

计算机能处理的浮点数也是有限制的。计算机不同, 它能处理的浮点数的范围也不同。浮点数精度要到第19章“浮点数”中再做进一步的讨论。使用cout可以输出浮点数。例如:

```
cout << "The answer is " << (1.0 / 3.0) << "\n";
```

浮点数除法与整数除法

除法运算符很特别。整数除法与浮点数除法有很大的不同。在整数除法中, 结果要被截断(即舍去小数部分)。例如, 整数除法19/10的结果为1。

如果除数或被除数中有一个是浮点数，那么将执行浮点数除法。19.0/10.0的结果为1.9。（19/10.0与19.0/10也是浮点数除法，但最好用19.0/10.0这种形式。）表4-2中还有几个例子。

表4.2 表达式示例

表达式	结 果	结果类型
1 + 2	3	整型
1.0 + 2.0	3.0	浮点型
19 / 10	1	整型
19.0 / 10.0	1.9	浮点型

C++允许把一个整数表达式的值赋给一个浮点变量。它自动进行整数 - 浮点数转换，然后再进行赋值。当把一个浮点数赋给整型变量时也要做类似的转换。当赋给整型变量时，要截短浮点数。

示例4.3 float1/float1.c

```
int    integer; // an integer
float floating; // a floating point number

main()
{
    floating = 1.0 / 2.0;          // assign floating 0.5
    integer = 1 / 3;              // assign integer 0
    floating = (1 / 2) + (1 / 2); // assign floating 0.0
    floating = 3.0 / 2.0;          // assign floating 1.5
    integer = floating;            // assign integer 1
    return (0);
}
```

注意表达式1/2是一个整型表达式，所以要执行整数除法，结果是一个整数0。

问题4.2: 示例4.4中为什么打印出“The value of 1/3 is 0”？对这个程序要进行哪些必要的修改？

示例4.4 float2/float2.cc

```
#include <iostream.h>

float answer; // the result of the divide

main()
{
    answer = 1/3;
```



```

    cout << "The value of 1/3 is " << answer << "\n";
    return (0);
}

```

字 符

`char`类型表示单个字符。字符说明的形式为：

```
char 变量;    //注释
```

字符要放到单引号（'）内。‘A’、‘a’、‘!’是字符常量。反斜杠（\）称作`escape`符（`escape character`）。它表明它后面的字符是一个特殊字符。例如，字符`\t`用来表示单个符号“tab”。`\n`是换行符，它使输出设备定位到下一行的行首，与打字机上的回车键类似。字符`\\`表示该符号本身。最后一点，可以用`\nnn`指定一个字符，这里`nnn`是对应于该字符的八进制码。表4.3罗列了一些特殊字符。在附录A中，列有全部字符的ASCII码。

表4.3 特殊字符

字 符	名 字	含 义
<code>\b</code>	回退	光标移到字符的左侧
<code>\f</code>	进纸	移到新页的开始
<code>\n</code>	换行	移到下一行
<code>\r</code>	回车	移到当前行的开始
<code>\t</code>	水平制表	移到下一个制表位（八列为界）
<code>\'</code>	撇号或单引号	字符'
<code>\"</code>	双引号	字符"
<code>\nnn</code>	字符 <code>nnn</code>	字符 <code>nnn</code> （八进制）
<code>\NN</code>	字符 <code>NN</code>	字符 <code>NN</code> （十六进制）

注 意

反的字符要包含在单引号（'）内；另一种不同的数据类型，即字符串，要包含在双引号（"）内。有一种好办法可以帮助我们记住这两种不同类型引号的使用法：单字符要放在单引号内。字符串可以含有任意个字符（包括双字符），所以它们要用双引号括起来。

示例4.5反转三个字符。

示例4.5 `print3/print3.cc`

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

```

char char1;    // first character
char char2;    // second character
char char3;    // third character

main()
{
    char1 = 'A';
    char2 = 'B';
    char3 = 'C';
    cout << char1 << char2 << char3 << " reversed is " <<
        char3 << char2 << char1 << "\n";
    return (0);
}

```

运行这个程序，它将输出：

```
ABC reversed is CBA
```

编程练习

练习4.1：编写一个程序，打印你的名字、身份证号和出生日期。

练习4.2：编写一个程序，打印一个用星号（*）组成的字母E。E有7行高，5个字符宽。

练习4.3：编写一个程序，计算3英寸宽、5英寸长的矩形的面积和周长。如果要计算的矩形是6.8英寸长、2.3英寸宽的话，需要做哪些修改呢？

练习4.4：编写一个程序，用字母组成“HELLO”，打印的每个字母7行高，5个字符宽。

答疑

问题4.1：在UNIX机器上，一个int中能够存放的最大数是2,147,483,647。当使用Turbo-C++时，这个数只有32,767。zip码中，92,126比32,767大。所以，程序不能运行，输出的结果为26,590。

用long int取代int就可以解决这个问题。我们将在第5章“数组、修饰符和读数”中讨论整数的不同种类。

问题4.2：这个问题涉及除法：1/3。数1和数3都是整数，所以这是一个整数除法。在整数除法中，小数要被截断。表达式应该写为：

```
answer = 1.0 / 3.0
```


本章内容:

- 数组
- 串
- 读数据
- 变量初始化
- 多维数组
- 整型
- 浮点型
- 说明
- 修饰符
- 十六进制与八进制常量
- 简捷运算符
- 编程练习

第5章

数组、修饰符 和读数

数 组

到目前为止，为了盖房子，我们给每块砖（变量）都取了名字。当砖的数目不大时，这还不错，但当我们盖更大的房子时怎么办呢？我们会指着一堆砖说：“这些砖用来垒墙。这是第一块砖、第二块砖、第三块砖……”。

数组允许我们对变量进行类似的处理。一个数组（array）是一组连续的内存位置，它们用来存放数据。数组中的每一项称为一个元素（element）。数组中元素的个数叫做数组的维（dimension）。典型的数组说明为：

```
// List of data to be sorted and averaged  
int    data_list[3];
```

它说明data_list是一个有三个元素的数组，这三个元素分别是data_list[0]、data_list[1]和data_list[2]，它们都是独立的变量。如要使用数组中的元素，就要利用一个称为下标（index）的数（方括号[]内的数）。C++是一个很有趣的语言，它喜欢从0开始计数，所以这三个元素的下标为0~2。

注 意

常识告诉你，当你说明data_list是有三个元素的数组时，data_list[3]是有效的。错了，data_list[3]是非法的。

示例5.1计算5个数的总和及平均值。

示例 5.1 five/five.cc

```
#include <iostream.h>

float data[5]; // data to average and total
float total;   // the total of the data items
float average; // average of the items

main()
{
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 46.5;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average = total / 5.0;
    cout << "Total " << total << " Average " << average << '\n';
    return (0);
}
```

该程序输出结果:

Total 211.5 Average 42.3

串

串是字符数组。特殊字符 ‘\0’ (NULL) 表示串的结束。

例如:

```
char name[4];

main()
{
    name[0] = 'S';
    name[1] = 'a';
    name[2] = 'm';
    name[3] = '\0';
    return (0);
}
```

该程序建立一个四个元素的字符数组。注意,我们必须为串结束符分配一个位置。

串常量由用双引号 (") 括起的正文组成。你可能已经注意到,我们用标准类cout进行输出时使用了很多的串常量。C++不允许数组直接赋值,所以下面的赋值语句是不对的:

```
name = "Sam";    // Illegal
```

你必须使用标准库函数strcpy，把一个串常量拷贝到变量中。（strcpy将拷贝完整的串，包括串结束符）。如果要把变量name初始化为“Sam”，你可以写下面的语句：

```
#include <string.h>

char name[4];

main()
{
    strcpy(name, "Sam");    // Legal
    return(0);
}
```

注意

#include <string.h>一行是必需的，它通知C++你要使用串函数库。

C++可以使用变长的串。例如，下面的说明：

```
#include <string.h>

char string[50];

main()
{
    strcpy(string, "Sam");
```

将建立一个数组（串），它最多可以含有50个字符。数组的大小是50，但串的长度为3。49个字符以内的任何一个串都可以存储在string内。（有一个位置留给串结束符NULL。）

有几个标准程序可以对串变量进行操作。这些程序列在表5-1中。

表5.1 字符串函数

函 数	描 述
strcpy(string1, string2)	把string2拷贝到string1中
strcpy(string1, string2)	把string2接到string1的后面
length =strlen(string)	返回string的长度
strcmp(string1, string2)	如果string1等于string2，则返回0；否则返回非零值

示例5.2说明strcpy的用法。

示例5.2 str/sam.cc

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

char name[30]; // First name of someone

main()
{
    strcpy(name, "Sam");
    cout << "The name is " << name << '\n';
    return (0);
}
```

示例5.3得到名字和姓，并把这两个字符串拼起来。程序把变量first初始化为名字（Steve）。而姓（Oualline）放到变量last中。我们把名字先拷贝到full_name中，用strcat函数加一个空格，再用该函数把姓接在后面，就得到了全名。

因为我们知道不会碰到多于99个字符的人名，所以串变量的大小取为100。（如果遇到名字多于99个字符，则程序将出错，并把名字一分为二）。

示例5.3 name2/man2.cc

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

char first[100]; // first name
char last[100]; // last name
char full_name[100]; // full version of first and last name

main()
{
    strcpy(first, "Steve"); // Initialize first name
    strcpy(last, "Oualline"); // Initialize last name

    strcpy(full_name, first); // full = "Steve"
                                // Note: strcat not strcpy
    strcat(full_name, " "); // full = "Steve "
    strcat(full_name, last); // full = "Steve Oualline"

    cout << "The full name is " << full_name << '\n';
    return (0);
}
```

该程序的输出结果为:

The full name is Steve Oualline

读数据

到目前为止，我们学习了如何计算表达式及输出结果。你还需要让你的程序能读数。输出类变量`cout`用运算符`<<`来输出数。而输入类变量`cin`使用运算符`>>`来读它们。例如，下列代码：

```
cin >> price >> number_on_hand;
```

读入两个数：`price`和`number_on_hand`。应该给这个程序送入两个数，两个数中间以空格来分隔。例如，如果你输入：

```
32 5
```

则`price`得到数32，而`number_on_hand`得到数5。

注意

在你输入数据时，并没有很精确的控制。对于简单的输入，C++的处理很合适。如果你的程序要读入一个数，而你键入的是`<enter>`，程序将跳过`<enter>`（它仅仅是一个空格）并等待你键入一个数。这样做有时会让你误认为程序死住了。

在示例5.4中，使用`cin`得到用户输入的一个数，然后倍乘这个数：

示例5.4 *double/double.cc*

```
#include <iostream.h>
char line[100]; // input line from console
int value;      // a value to double

main()
{
    cout << "Enter a value: ";
    cin >> value;
    cout << "Twice " << value << " is " << value * 2 << '\n';
    return (0);
}
```

该程序向用户要一个数，然后倍乘它。注意，在`Enter a value:`的后面并没有`\n`。因为我们不想在提示后换行。该程序运行时的情形如下：

```
Enter a value: 12
Twice 12 is 24
```

如果我们把Enter a value:换为Enter a value:\n, 则结果会是:

```
Enter a value:
12
Twice 12 is 24
```

问题5.1: 示例5.5计算给定了底和高的三角形面积。由于某种原因, 编译器认为我们没有说明变量width。但说明就在第二行, 即height变量说明之后。编译器为什么看不到它呢?

示例5.5 comment/comment.cc

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

```
int height;    /* the height of the triangle
int width;     /* the width of the triangle */
int area;      /* area of the triangle (computed) */
```

```
main()
{
    cout << "Enter width height? ";
    cin >> width >> height;
    area = (width * height) / 2;
    cout << "The area is " << area << '\n';
    return (0);
}
```

cin语句的一般形式是:

```
cin >> 变量;
```

这个语句可以处理所有的简单类型变量, 如int、float和char。读串时稍微困难点。要读串, 需使用如下的语句:

```
cin.getline(串, sizeof(串));
```

例如:

```
char name[100];    // The name of a person
cin.getline(name, sizeof(name));
```

我们将在第16章“文件输入/输出”中讨论getline和sizeof函数。

读串时, cin类处理一直到行尾的所有内容。示例5.6从键盘读入一行, 并报告该行的长度。

示例 5.6 len/len.cc

```
#include <string.h>
```

```

#include <iostream.h>

char line[100]; // A line of data

main()
{
    cout << "Enter a line:";
    cin.getline(line, sizeof(line));

    cout << "The length of the line is: " << strlen(line) << '\n';
    return (0);
}

```

运行该程序时，得到：

```

Enter a line:test
The length of the line is: 4

```

变量初始化

C++允许在说明语句中对变量进行初始化，即赋初值。例如，下列语句说明了一个整型变量counter，并给它赋初值0。

```
int counter(0);    // number cases counted so far
```

原来C语言的语法还支持下面的写法：

```
int counter = 0;    // number cases counted so far
```

用类似的方法也可以给数组赋初值。数组元素初值表必须用花括号（{ }）括起来。例如：

```

// Product numbers for the parts we are making
int product_codes[3] = {10, 972, 45};

```

它等价于：

```

product_codes[0] = 10;
product_codes[1] = 972;
product_codes[2] = 45;

```

花括号（{ }）中的元素个数不一定非与数组大小相一致。如果提供的数据太多了，会出现一条警告信息。如果数据不够，那么有的元素就得不到初值。

如果没有给出数组维数，C++将从初值表中元素的个数判定它。例如，可以用下列语句

给变量`product_codes`赋初值:

```
// Product numbers for the parts we are making
int product_codes[] = {10, 972, 45};
```

用类似的方法, 可以给串赋初值。下列语句给变量`name`赋初值 “Sam”:

```
char name[] = {'S', 'a', 'm', '\0'};
```

给串赋初值时, C++还有一种简写方式, 即用双引号 (") 来简化初值表。前一个例子可以如下书写:

```
char name[] = "Sam";
```

`name`的维为4, 因为C++在串尾分配了一个空间放 “\0” 字符。

C++使用变长的串。例如, 说明:

```
char string[50] = "Sam";
```

建立了一个(串)数组, 其中最多可含50个字符。数组的大小是50, 串的长度为3, 任何长度不超过49的串都可以存储在`string`中。(有一个位置留给串结束符NULL。)

注意

该语句只给`string`串的50个元素中的4个赋了初值。其它的46个元素没有赋初值, 相应的内容是随机数。

多维数组

数组可以有多维数组。二维数组的说明为:

```
type 变量[大小1][大小2]; // 注释
```

例如:

```
// a typical matrix
int matrix[2][4];
```

注: C++并没有使用其它语言中的惯用法`matrix[10,12]`。

要访问`matrix`中的元素, 可以用下列语句:

```
matrix[1][2] = 10;
```

C++对数组的维数并没有限制(只受机器的内存所限)。我们可以加另外的维。

```
four_dimensions[10][12][9][5];
```

给多维数组赋初值类似于给一维数组赋初值。每维元素值要放在一对花括号{}中。下列说明:

```
// a typical matrix
int matrix[2][4];
```

可以认为是说明了一个二个元素的数组, 每个元素又是一个有四个元素的数组。可用下列语句给该数组赋初值:

```
// a typical matrix
int matrix[2][4] =
{
    {1, 2, 3, 4},
    {10, 20, 30, 40}
};
```

它是下面写法的简化表示:

```
matrix[0][0]=1;
matrix[0][1]=2;
matrix[0][2]=3;
matrix[0][3]=4;
```

```
matrix[1][0]=10;
matrix[1][1]=20;
matrix[1][2]=30;
matrix[1][3]=40;
```

问题5.2: 下列程序为什么打印出错误的答案?

示例5.7 *array/array.cc*

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

```
int array[3][5] = { // Two dimensional array
    { 0, 1, 2, 3, 4 },
    {10, 11, 12, 13, 14 },
    {20, 21, 22, 23, 24 }
};
```

```
main()
{
    cout << "Last element is " << array[2,4] << '\n';
    return (0);
}
```

在Sun 3/50上运行该程序时产生下列信息:

Last element is 0x201e8

读者的回答可能是多种多样的。

整 型

C++被看作是一种中级语言，因为它允许使用者接触到机器的实际硬件。有些语言，像BASIC，不让使用者了解处理器的工作详情。这种做法使语言失掉了高效性。C++能让你详细了解要使用的硬件的情况。

例如，大多数机器允许使用不同长度的数。简单的BASIC仅允许程序员使用一种类型的数。虽然这简化了编程，但BASIC程序的效率非常低。C++允许程序员说明许多种不同的整数，因此，可以最大限度地利用硬件资源。

类型说明符int告诉C++使用最高效（对于你使用的机器而言）的整数。根据机器的不同，可以是2到4个字节。有时你需要额外的位来存放比一般的int所能表示的数更大的数。下面说明：

```
long int answer;           // the answer of our calculations
```

用来分配一个长整数。修饰词long通知C++你希望为一个整数分配额外的存储空间。如果你要使用小一点的数，并希望压缩存储空间，可以使用修饰词short。

```
short int year;           // Year including the 19xx part
```

C++保证了short的存储空间小于等于int的，而int的小于等于long的。在实际中，short几乎总是分配2个字节，long分配4个字节，而int分配2个或4个字节。（有关数范围的详细内容请参见附录B。）

长整型常量的结尾要有字符“L”。例如：

```
long int var = 1234L;      // Set up a long variable
```

实际上，“L”既可以大写，也可以小写。我们更愿意用大写，因为小写字母容易与数字“1”混淆。

```
long int funny = 12l;      // Is this 12<long> or one hundred twenty-one?
```

short int类型使用2个字节，或16位。其中15位用来放数字，1位放符号。它表示的数的范围是-32,768 (-2^{15}) 到32,767 ($2^{15}-1$)。unsigned short int使用所有的16位来放数，它表示的数的范围是0到65,535 ($2^{16}-1$)。所有的int说明都默认为signed，所以说明：

```
signed long int answer;    // final result
```

等同于：

```
long int answer;          // final result
```


还有一种极短整型：`char`类型。字符变量占用1个字节。它们可以用来表示-128到127或是0到255之间的数。和整数不一样，它们没有被默认为是signed的；缺省值由编译器来决定。*

问题：下面的字符变量是有符号的还是无符号的？

```
char foo;
```

答案：

- a. 它是有符号的。
- b. 它是无符号的。
- c. 它要由编译器来决定。
- d. 如果你总是指定了signed或是unsigned，就不必担心这样的问题。

读写短整型数时要运用点小技巧。如果你想在输出语句中使用`char`类型的变量，那么它将作为一个字符打印出来。你必须设法让C++把`char`变量认作是一个整型量。使用`int`运算符就可以实现这一点。示例5.8演示如何把极短整型量作为一个数打印出来。

示例5.8 two2/two2.cc

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

```
signed char ch; // Very short integer
                // Range is -128 to 127
```

```
int main()
{
    ch = 37;
    cout << "The number is " << int(ch) << '\n';
    return (0);
}
```

程序开始时，说明了一个字符型变量`ch`。该变量赋值37。实际上这是一个整数，而不是一个字符，但C++并不这样认为，在下一行，我们输出该变量的值。如果我们直接输出`ch`，C++将把它看作是一个字符。代码`int(ch)`告诉C++：“把这个字符看作是一个整数。”

不能读入一个极短整型量。你只能把它先作为一个`short int`型量读入，然后再把它赋给一个极短整数。

整型量小结：

`long int`类型说明允许程序员明确指明所需要的额外精度（以内存为代价）。`short int`数节省空间，但表示的数更受限制。最紧凑的整型是`char`类型。它们表示的数的范围最小。

* Turbo-C++中有一个命令行开关，用此开关可以把`char`类型的缺省值设为signed或是unsigned。

unsigned类型因为没有负数，所以使其能表示的数的范围扩大了一倍。这种类型的数在使用时要依赖于程序和存储环境。不同类型整数的表示范围列在附录B中。

浮点型

浮点类型也有许多种。float表示正常的精度（通常是4个字节）。double表示双精度（通常是8个字节）。双精度表示的数的范围和精度，均为单精度（float）变量的两倍。

修饰词long double表示再提高精度。在有些系统中，这种类型等同于double类型，所有类型的浮点数都是有符号的。

在大多数机器中，单精度浮点数的指令比双精度的指令运行得要快（但精度要低一些）。双精度是以运行时间和存储空间来换取运算精度的。大多数情况下，float类型已足够了；但是，如果考虑的就是精度问题，那么就使用double吧（请参见第19章“浮点数”）。

常量与引用说明

有时你会使用一个不会改变的值，如 π 。关键字const就表示一个永不改变的变量。我们用下面的语句说明pi这个值：

```
const float PI = 3.1415926;    // The classic circle constant
```

注意

按照约定，变量名用小写字母，而常量名用大写字母。但是语言中并没有要求必须这样做，而且有些系统使用完全不同的约定。

常量必须在说明时赋初值，而且这个值永远不能改变。例如，如果你想把PI的值重置为3.0，会产生一条错误信息：

```
PI = 3.0;    // Illegal
```

当说明数组的大小时，可以使用整型常量：

```
const int TOTAL_MAX = 50;    // Max. number of elements in the total
                               list
float total_list[TOTAL_MAX]; // Total values for each category
```

注意

C++允许程序员在说明数组时使用整型表达式。例如，可以说明total_list[10]或是total_list[7+3]。但是，有些编译器，如Borland C++ Version 3.1，不允许在这种表达式中出现整型常量。

例如：

```
const int first_part = 3;
const int second_part = 7;

float total1[3+7];      // Works even in Borland C++ Version 3.1
float total2[first_part + 7]; // Fails in Borland C++ Version 3.1
float total3[first_part + second_part]; // Also fails
```

另一类特殊的变量类型是reference类型。典型的引用说明是：

```
int count;                // Number of items so far
int &actual_count = count; // Another name for count
```

特殊字符“&”告诉C++，actual_count是一个引用变量。有了这个说明，count和actual_count就是同一个变量了。例如，下面两个语句是等价的：

```
count = 5;                // "Actual_count" changes too
actual_count = 5;         // "Count" changes too
```

换句话说，简单变量说明就是声明了一个要放数的盒子。一个引用变量就是给这个盒子再加上另一个名字，如图5.1所示。

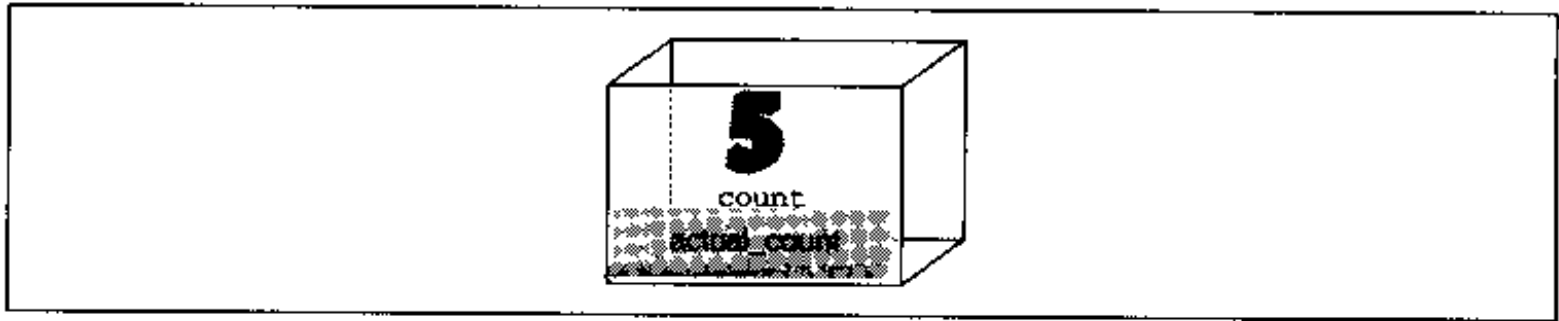


图 5.1 引用变量

这种形式的引用变量不太有用。事实上，在真正的程序中几乎从来不用。在第9章“变量作用域和函数”中，你将看到另一种形式的引用变量，这种很有用。

修饰符

正如你看到的，C++允许你在变量说明时指定多个修饰符。修饰符可以看作是形容词，它描述了紧随其后的类型。表5.2概述了各种修饰符。

表5.2 修饰符和简单类型

特殊的	类	规格	符号	类型
volatile	register	long	signed	int
空	static	short	unsigned	float

表5.2 修饰符和简单类型

特殊的	类	规格	符号	类型
	extern	double	空	char
	auto	空		空
	空			

特殊的

关键字volatile用于一些特殊的程序，如I/O设备驱动程序和共享内存应用程序等。它是个高级修饰符，它的使用已超出本书范围。

volatile

说明一个特殊的变量，它的值可能随时变化。

空

正常变量

类

变量的类将在第9章“变量作用域和函数”中详细讨论。下面简要叙述各种类：

register

它指明一类频繁使用的变量，这类变量应该存放在机器的寄存器中。见第17章“调试和优化”。

static

这个词的意思要依据上下文来定。在第9章“变量作用域和函数”及第23章“模块化程序设计”中将讨论该关键字。

extern

变量定义在另一个文件中。详细内容请参见第23章。

auto

从堆栈中分配的变量。该关键字几乎从来不用。

空

表明选择了缺省类（auto）。

规格

规格修饰符允许你为变量选择最有效的长度。

long

指明比正常的整型量要大。（有些非标准的编译器使用long double来表示一个很大的浮点数变量。）

short

比正常的整型量小。

double

双倍大小的浮点数。

空

表示一个正常规格的数。

符 号

数可以是有符号的，也可以是无符号的。该修饰符仅适用于char和int类型。浮点数总是有符号的。对于int类型，缺省值是有符号的，而对于字符型变量，缺省值是不确定的。

类 型

它指明变量的类型。简单类型包括：

int 整型

float 浮点数

char 单字符，也可以用于极短整型

十六进制和八进制常量

整数就是一串数字，如1234、88、-123等等。这些是十进制数（基为10）：174或17410。计算机处理的是二进制数（基为2）：10101110₂。八进制数（基为8）与二进制数之间很容易相互转换。三位二进制数（2³=8）可以转换为一位八进制数。所以10101110₂可以写作10 101 110₂，进而转换为八进制数256₈。十六进制数（以16为基数）有类似的规则，不过只是4位一组。例如，10000100₂是1000 0100，或84₁₆。

在表示八进制和十六进制数时，C++语言有其具体约定。以“0”开头表示一个八进制数。如，0123就是123（八进制）或83（十进制）。以“0x”打头的数是十六进制（以16为基准）常量。所以0x15就是21（十进制）。表5.3列出了三种进制下几个数的对应值。

表5.3 整数示例

十进制	八进制	十六进制
6	06	0x6
9	011	0x9
15	017	0xF

问题5.3：下面的程序为什么不能打印出zip码的正确值？它打印出什么内容？

```

long int zip;           // Zip code

main()
{
    zip = 02137L;       // Use the zip code for Cambridge MA

    cout << "New York's zip code is: " << zip << '\n';
    return(0);
}

```

简捷运算符

C++不仅提供了一组丰富的说明，它还提供了大量具有特殊目的的运算符。程序员常想把一个变量值增量（变量值加1）。用通常的赋值语句时，应如下书写：

```
total_entries = total_entries + 1;
```

对于这种常用的操作，C++提供了一个简捷方式。运算符++可用于增量操作。

```
++total_entries;
```

类似的运算符--可用于变量的减量（变量值减1）。

```

--number_left;
// Is the same as
number_left = number_left - 1;

```

但是，假定你想加2，而不是加1，则你要使用如下的写法：

```
total_entries += 2;
```

它等同于：

```
total_entries = total_entries + 2;
```

可用于这种方式的简单运算符列在表5.4中。

表5.4 简捷运算符

运算符	例子	等价语句
+=	x += 2;	x = x + 2;
-=	x -= 2;	x = x - 2;
*=	x *= 2;	x = x * 2;
/=	x /= 2;	x = x / 2;
%=	x %= 2;	x = x % 2;

副作用

不幸的是，C++允许程序员使用副作用(side effect)。副作用是指，语句中主操作执行时附带执行的操作。

例如，下面是合法的C++代码：

```
size = 5;
result = ++size;
```

第一条语句给size赋值5。第二条语句：

1. size增1（副作用）。
2. 把size的值赋给result（主操作）。

但是，它们的执行顺序如何？有四种可能的答案：

1. 把size的值（5）赋给result，然后size加1。最后，result是5，size是6。
2. size增1，然后把size的值（6）赋给result。最后，result是6，size是6。
3. 由编译器来决定，而且视计算机的不同，最后的结果也不同。
4. 如果你不写这样的代码，就不必担心这类问题。

正确答案是2。加1操作在赋值操作之前进行。但是，4是更好的回答。C++的主要操作已足够我们学的了，没有必要再去为副作用操心。

注意

有些程序员喜欢很紧凑的代码。这是从计算机发展的初期阶段遗留下来的，那时内存的价格非常昂贵。编程艺术发展到今天，依我的观点，简洁的代码比紧凑的代码更有价值。（许多人喜欢读的大部头小说，并不是用速记法来写的。）

实际上，C++提供了两种形式的++运算符。一种是变量++，另一种是++变量。第一种：

```
number = 5;
result = number++;
```

先进行表达式的赋值，然后number再增1，所以result的值为5。第二种：

```
number = 5;
result = ++number;
```

先进行增量计算，然后进行表达式的赋值。本例中，result的值为6。但是，这样使用++或--运算符可能会编出令人吃惊的代码：

```
o = --o - o--;
```


这行代码看上去像是有人在写电报码。程序员不认为这是一行语句，他得先译解它。如果你从不把++或--与别的语句混在一起，而总是把它们单独放在一行，那么两种形式的差别就不那么重要了。

注意

宁可使用前缀形式++变量，也不要使用后缀形式变量++；因为使用前缀形式，编译器能生成更简单些的代码。

更复杂的副作用有时连C++编译器也弄不明白。考虑下面的这段代码：

```
value = 1;
result = (value++ * 5) + (value++ * 3);
```

该表达式告诉C++执行下列各步：

1. value值乘5，并且value加1。
2. value值乘3，并且value加1。
3. 把两个结果相加。

和前面的例子不一样，第1步和第2步有相同的优先级，所以编译器可以按它自己决定的顺序来执行。假定它先执行第1步，如图5.2所示。

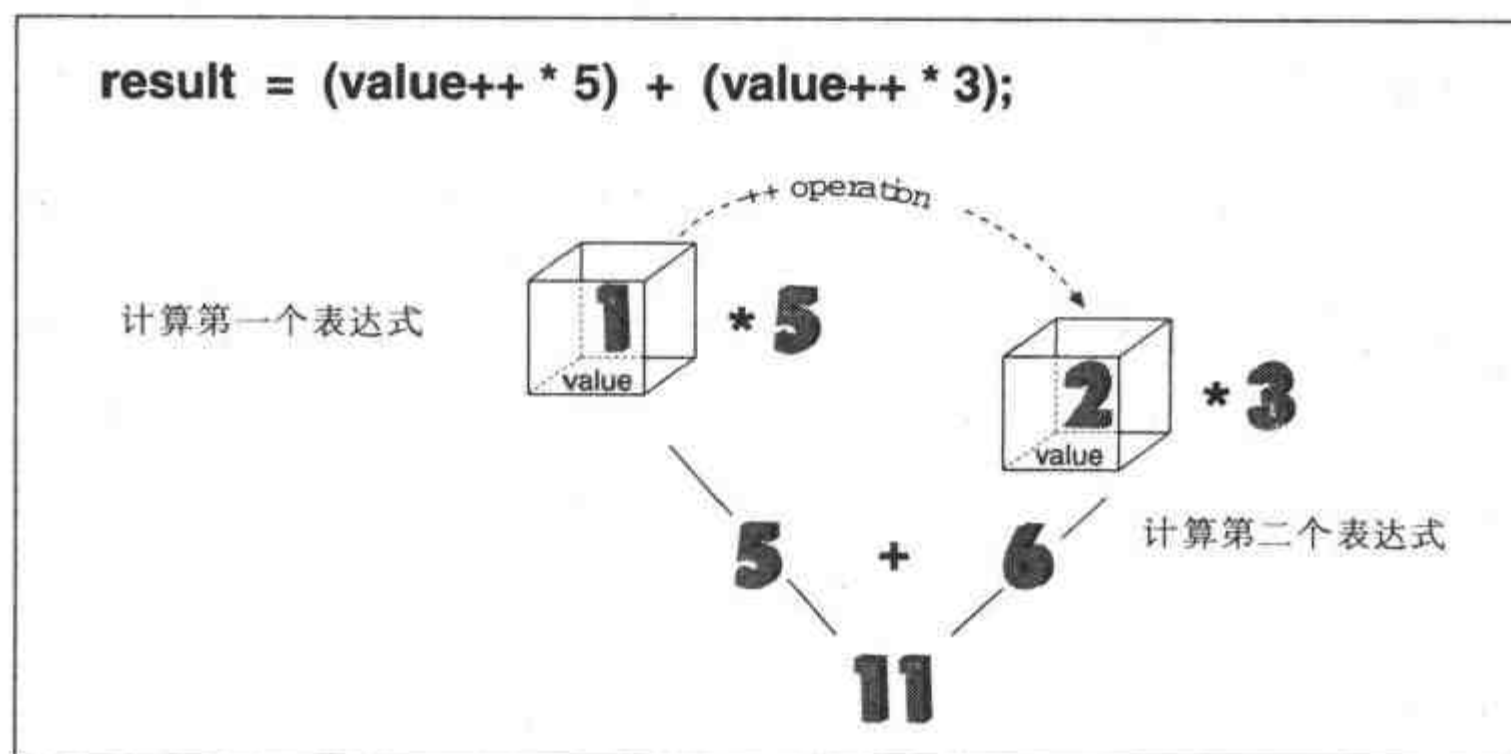


图 5.2 表达式赋值方法1

但它也可能先执行第2步，如图5.3所示。

使用第一种方法，结果为11；使用第二种方法，结果为13。因为该表达式的结果模棱两可，所以在较大的表达式中使用运算符++，会引出问题。（++和--不仅仅能带来这个问题。在第10章“C++预处理器”中我们还会遇到更多的麻烦。）

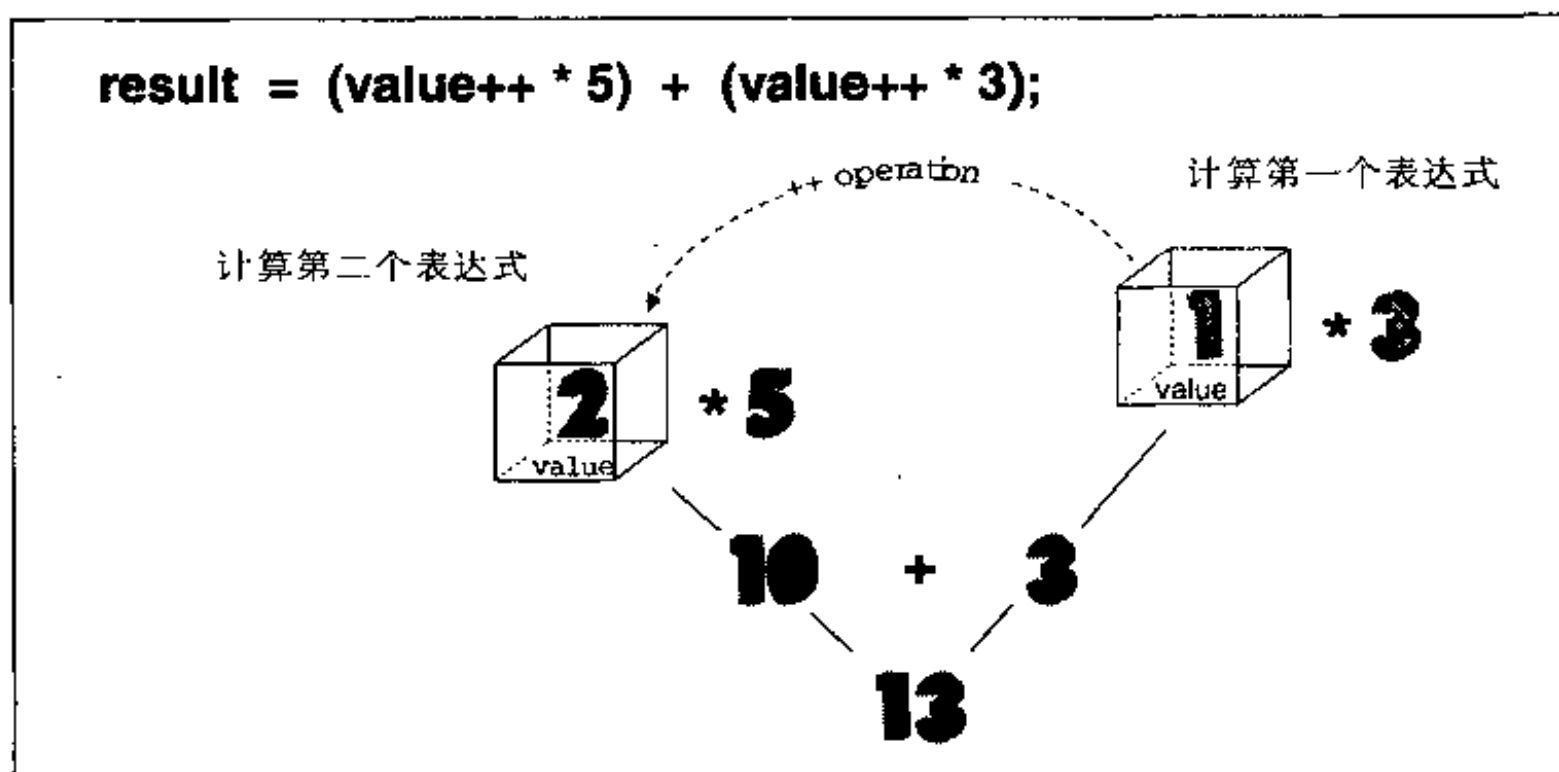


图 5.3 表达式赋值方法2

为避免这些问题，并保持程序简单，请永远把++和--单独放在一行。

编程练习

练习5.1: 编写一个程序，把摄氏温度转为华氏温度。 $F = \frac{9}{5}C + 32$

练习5.2: 编写程序计算球的体积。 $V = \frac{4}{3}\pi r^3$

练习5.3: 编写程序计算给定了长和宽的矩形的周长。 周长 = $2 * (\text{宽} + \text{长})$

练习5.4: 编写程序把每小时英里数转换为每小时公里数。 英里 = $(\text{公里} * 0.6213712)$

练习5.5: 编程序输入小时和分钟，然后计算总的分钟数。

练习5.6: 编程序输入分钟数，然后输出小时数和分钟数。

答疑:

问题5.1: 变量height后的注释中缺少注释结束符 (*/)。注释一直持续到下一行，把变量width的说明也包括在注释中了。示例5.9中的程序中，注释带有下列划线。

示例5.9 tempconv.c

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

```
int height;    /* The height of the triangle
int width;     /* The width of the triangle*/
int area;      /* Area of the triangle (computed) */
```

```
main()
```



```

{
    cout << "Enter width and height? ";
    cin >> width >> height;
    area = (width * height) / 2;
    cout << "The area is " << area << '\n';
    return (0);
}

```

问题5.2: 问题在于zip码的开头是一个0。这告诉C++，02137是一个八进制数。当我们打印它时，是用十进制数打印的。因为02137₈就是8503₁₀，所以程序输出：

```
New York's zip code is: 8503
```



本章内容:

- if语句
- else语句
- 为何不使用strcmp函数
- 循环语句
- while语句
- break语句
- continue语句
- 随处赋值的副作用
- 编程练习
- 答疑

第6章

判定和控制 语句

计算和表达式仅仅是计算机编程语言的一小部分。判定和控制语句也是必需的，这些语句指定语句的执行顺序。

到目前为止，你已经编写了线性程序(*linear program*)，这些程序中的语句一句接一句地执行。本章中，你将看到如何使用分支语句(*branching statement*)和循环语句(*looping statement*)改变一个程序的控制流(*control flow*)。分支语句可以根据条件子句(*conditional clause*)决定一段代码是否执行。循环语句可以数次重复执行一段代码，直到满足某些条件为止。

if语句

if语句允许在程序中加入某些判定。if语句的一般形式为：

if (条件)

语句;

如果表达式为真（非零），则执行语句。如果表达式为零，将不执行语句。例如，假定你编写一个帐单程序。如果顾客的结余没有了，或是他已经赊帐了（结余为负数），就输出一条信息。在C++中，相应的语句为：

```
if (total_owed <= 0)
    cout << "You owe nothing.\n";
```

运算符`<=`是关系运算符(*relational operator*)，它表示小于或等于。该语句读作“如果`total_owed`小于或等于零，则打印信息”。所有的关系运算符列在表6.1中。

表6.1 关系运算符

运算符	含 义
<=	小于或等于
<	小于
>	大于
>=	大于或等于
==	等于
!=	不等于

使用逻辑运算符(logical operator)，可以连接多个关系表达式。例如，语句：

```
if ((oper_char == 'Q') || (oper_char == 'q'))
    cout << "Quit\n";
```

使用逻辑或运算符(logical OR operator) (||)，使得当oper_char为小写的“q”或是大写的“Q”时，输出“Quit”。表6.2列出所有的逻辑运算符。

表6.2 逻辑运算符

运算符	用 法	含 义
逻辑或 ()	(表达式1) (表达式2)	表达式1或表达式2为真时，为真
逻辑与 (&&)	(表达式1)&& (表达式2)	表达式1和表达式2均为真时，为真
逻辑非 (!)	!(表达式)	如果表达式为真，则为假；如果表达式为假，则为真

if后面可以有多个语句，这些语句要放在花括号({})中。例如：

```
if (total_owed <= 0) {
    ++zero_count;
    cout << "You owe nothing.\n";
}
```

从可读性考虑，花括号中的语句要缩进一些。这样程序员可以快速分辨哪些语句的执行是有条件的。读者在后面将会看到，缩进错误会使程序的执行发生错误，而且程序的可读性很差。

else语句

if语句的另一种形式是：

```
if (条件)
```

语句;

else

语句;

如果条件为真，将执行第一条语句。如果条件为假，则执行第二条语句。在前面的示例中，当顾客没有结余时才输出信息。实际上，如果有结余金额的话，我们可能想告诉顾客他的结余还有多少：

```
if (total_owed <= 0)
    cout << "You owe nothing.\n");
else
    cout << "You owe " << total_owed << " dollars\n";
```

PASCAL程序员应注意：和PASCAL程序不一样，C++要求你在else之前的语句后加一个分号。

现在考虑下面的程序段：

```
if (count < 10)      // If #1
    if ((count % 4) == 2)  // If #2
        cout << "Condition:White\n";
    else // (Indentation is wrong)
        cout << "Condition:Tan\n";
```

这里有两个if语句和一个else语句。这个else语句是哪个if语句的？选择一个正确的答案：

1. 它属于第一个if。
2. 它属于第二个if。
3. 如果你从来不写这样的代码，就不必考虑这个问题。

正确的回答是3。根据C++语法规则，else属于最近的一个if。所以从语法上来讲，2是对的。但写这样的代码是违背KISS（即Keep It Simple, Stupid——让程序简单又简单）原则的。最好是写尽可能简单的程序。这段代码应该是这样的：

```
if (count < 10) {      // If #1
    if ((count % 4) == 2)  // If #2
        cout << "Condition:White\n";
    else
        cout << "Condition:Tan\n";
}
```

在第一个例子中，很难看清else子句是属于哪个if语句的，但是，增加一对花括号后，改善了程序的可读性，程序清楚易懂了。

为何不使用strcmp函数

strcmp函数比较两个串，如果它们相等，返回零；如果它们不相等，返回非零。要检查两个串是否相等，使用下面的代码：

```
// Check for equal
if (strcmp(string1, string2) == 0)
    cout << "Strings equal\n";
else
    cout << "Strings not equal\n";
```

有些程序员忽略了注释和 `== 0` 子句，写出了下面令人费解的语句：

```
if (strcmp(string1, string2))
    cout << ".....";
```

第一眼看上去，程序显然是在比较两个串，当它们相等时，执行cout语句。不幸的是，这个“显然”是错误的。如果串相等，strcmp返回零，不执行cout语句。因为strcmp有返回值，所以你在使用strcmp时应该非常小心，并注明使用目的。

循环语句

计算机不仅能计算，它还能一遍又一遍地计算。要让计算机重复它的工作，可以使用loop语句。循环语句有很多用处。例如，统计一个文档中的字数，或者统计帐目的收支数。

while语句

当程序重复执行某些操作时，可以使用while语句。while语句的一般形式为：

```
while (条件)
```

```
    语句;
```

程序重复执行while块中的语句，直到条件为假（0）时结束。（如果开始时条件为假，则语句一次也不执行。）

例如，示例6.1计算100以内的斐波那契数(Fibonacci number)。斐波那契数列为：

1 1 2 3 5 8 ...

这些项是通过下面的等式得到的：

```
1
1
```



2 = 1 + 1
3 = 1 + 2
5 = 3 + 2

等等。

项的一般计算公式为:

$$f_n = f_{n-1} + f_{n-2}$$

这是一个数学等式, 其中用到了数学风格的变量名 (f_n)。数学家用这种精练的方法命名变量。设计程序时, 这种精练命名法很危险, 所以我们还是用C++中冗长的名字来命名变量:

f_n 对应于 next_number
 f_{n-1} 对应于 current_number
 f_{n-2} 对应于 old_number

在C++中, 上述等式表示为:

```
next_number = current_number + old_number;
```

我们循环执行计算, 直到当前项大于或等于100时为止。while循环:

```
while (current_number < 100)
```

将重复计算并输出结果, 直到满足修饰条件时结束。

在while循环中, 我们计算current_number的值并输出。然后, 计算下一项的值。

循环体将完成这些工作。斐波那契数列的前两项是1和1。我们把前两项初始化为两个数。

程序执行时的情况如图6.1所示。开始时, current_number和old_number为1。我们输出当前项的值。然后计算变量next_number的值(值为2)。之后, 把next_number的值送给current_number, 把current_number的值送给old_number, 去计算下一项。重复这个过程, 直到计算出最后一项, 退出while循环。

示例6.1列出相应的C++程序。

示例6.1 fib/fib.cc

```
#include <iostream.h>
int    old_number;      // previous Fibonacci number
int    current_number;  // current Fibonacci number
int    next_number;     // next number in the series

main()
{
```



```

// start things out
old_number = 1;
current_number = 1;

cout << "1\n"; // Print first number

while (current_number < 100) {

    cout << current_number << '\n';
    next_number = current_number + old_number;

    old_number = current_number;
    current_number = next_number;
}
return (0);
}

```

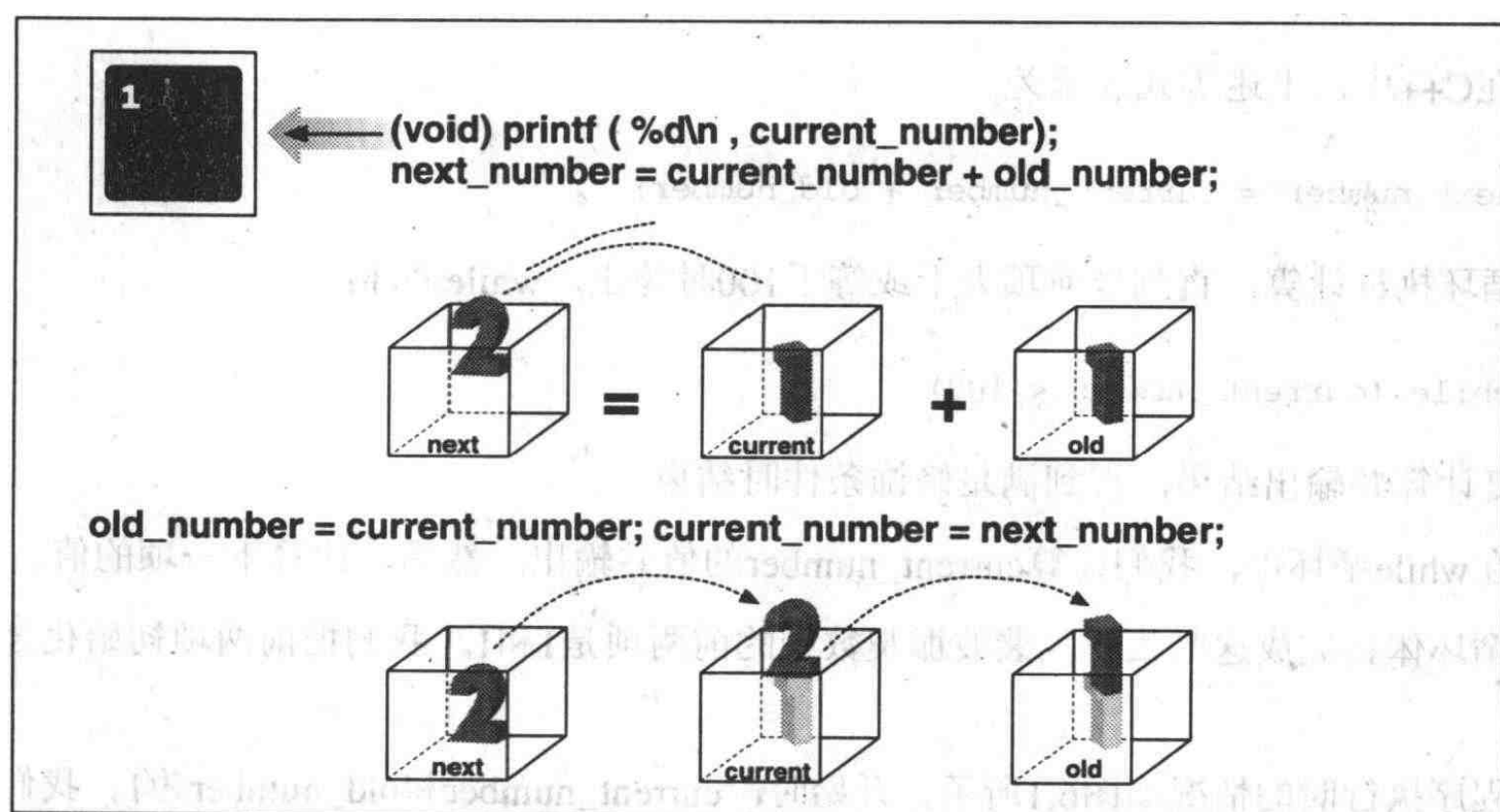


图 6.1 计算斐波那契数

break语句

我们已经使用while语句计算了100以内的斐波那契数。当开始条件变为假时，退出循环。还可以在任何时刻使用break语句退出循环。

假定你想累加一串数，但你并不知道要加多少项。你必须设法告诉程序它已经加完了所有的数。程序6.2中使用数0作为累加数的结束。

注意，while语句是这样开始的：

```
while (1) {
```

该程序将永远地循环下去，因为仅当表达式1为0时，才退出while循环。退出循环的唯一办法，是通过break语句。

当我们看到结束标志（0）时，使用语句：

```
if (item == 0)
    break;
```

退出循环。

示例6.2 total/total.cc

```
#include <iostream.h>
int total; // Running total of all numbers so far
int item; // next item to add to the list

main()
{
    total = 0;
    while (1) {
        cout << "Enter # to add \n";
        cout << " or 0 to stop:";
        cin >> item;

        if (item == 0)
            break;

        total += item;
        cout << "Total: " << total << '\n';
    }
    cout << "Final total " << total << '\n';
    return (0);
}
```

continue语句

continue语句很像break语句，所不同的是，它不是结束循环，而是跳到循环体的开头，再去执行它。例如，我们修改前面的程序，只累加大于0的数，程序如示例6.3所示。

示例6.3 total2/total2.cc

```
#include <iostream.h>
int total; // Running total of all numbers so far
int item; // next item to add to the list
int minus_items; // number of negative items

main()
```



```

{
    total = 0;
    minus_items = 0;
    while (1) {
        cout << "Enter # to add\n";
        cout << " or 0 to stop:";
        cin >> item;

        if (item == 0)
            break;

        if (item < 0) {
            ++minus_items;
            continue;
        }
        total += item;
        cout << "Total: " << total << '\n';
    }

    cout << "Final total " << total << '\n';
    cout << "with " << minus_items << " negative items omitted\n";
    return (0);
}

```

随处赋值的副作用

C++几乎允许在所有的地方使用赋值语句。例如，你可以把赋值语句放在另一个赋值语句中：

```

// don't program like this
average = total_value / (number_of_entries = last - first);

```

它等价于：

```

// program like this
number_of_entries = last - first;
average = total_value / number_of_entries;

```

第一种写法把`number_of_entries`的赋值藏在表达式中。但程序应该清楚简单，不应该隐藏任何成份。编程最重要的规则就是保持程序简单。

C++还允许你把赋值语句放在`while`条件中。例如：

```

// do not program like this
while ((current_number = last_number + old_number) < 100)

```

```
cout << "Term " << current_number << '\n';
```

避免编写这类程序。看看下面的写法，它的逻辑是多么简单：

```
// program like this
while (1) {
    current_number = last_number + old_number;

    if (current_number >= 100)
        break;
    cout << "Term " << current_number << '\n';
}
```

问题6.1： 因为某种奇怪的原因，示例6.4中的程序把所有人的结余都计算为0。为什么？

示例6.4 *balance/balance.cc*

```
#include <iostream.h>
int    balance_owed;    // amount owed

main()
{
    cout << "Enter number of dollars owed:";
    cin >> balance_owed;

    if (balance_owed = 0)
        cout << "You owe nothing.\n";
    else
        cout << "You owe " << balance_owed << " dollars.\n";

    return (0);
}
```

执行程序时输出：

```
Enter number of dollars owed: 12
You owe 0 dollars.
```

编程练习

练习6.1： 编程序计算两点间距离。

练习6.2： 教授用表6.3转换学生的评分等级。

表6.3 评分等级

百分成绩	评分等级
0~60	F
61~70	D
71~80	C
81~90	B
91~100	A

给出一个百分成绩，请输出对应的评分等级。

练习6.3：修改前面的程序，根据百分成绩中最后一位数的大小，在评分等级后输出+或-。评分等级修改内容参照表6.4。

表6.4 评分等级修改表

最后一位数字	评分等级修改为
1~3	-
4~7	无
8~0	+

例如，81=B-，94=A，68=D+。注意：F只有F。没有F+或F-。

注意

程序员常常要修改其他人编写的程序。把别人编写的示例6.4拿来，并修改它，这不失为一个好的练习。

练习6.4：给定一金额（小于1元钱），计算其中所含2角5分、1角、5分、1分各多少。

练习6.5：闰年是指年份能被4整除，并除去能被100整除但不能被400整除的年份。编写一个程序，判定某一年是否是闰年。

练习6.6：给定一名雇员的工作时间（小时）及每小时的工资数，编程序计算他的周工资额。40小时以上的工资数，按超过原工资标准半倍的加班工资来统计。

答疑

问题6.1：该程序讨论一个最常见的C++错误，这也是最令人灰心丧气的错误之一。问题起因于C++允许把赋值语句放在if条件中。语句：

```
if (balance_owed = 0)
```

使用了一个等号，而不是两个。C++将把balance_owed的值赋为0，然后检查结果（它为0）。如果结果非0（真），将执行if子句。因为结果为0（假），所以执行else子句，程序将输出错误的答案。

语句

```
if (balance_owed = 0 )
```

等价于

```
balance_owed = 0;  
if (balance_owed != 0)
```

语句应该写为：

```
if (balance_owed == 0)
```

这是初学程序设计的人最常犯的错误。也是最难找到的错误之一。

我曾经教过C语言程序设计课程。该课程结束后大约一个月的一天，我在街上遇见了从前的一名学生。他过来与我打招呼并说：“史蒂夫，我跟你说实话吧。上课时，我一直认为你太偏爱讨论=与==的错误了，但我现在不这样认为了。你知道，我刚刚写了第一个C程序作业，猜猜我犯了什么错误。”

许多程序员习惯于在==语句中先写常量。例如：

```
if (0 == balance_owed)
```

这样写，如果程序员写错了，把==写为=，即写成：

```
if (0 = balance_owed)
```

那么编译器会出错。（你不能把balance_owed赋给0。）

2



本章内容:

- 设置
- 规格说明
- 代码设计
- 原型
- Makefile
- 测试
- 调试
- 维护
- 修改
- 电子考古
- 编排程序
- 编程练习

第7章

程序设计

过程

编程不仅仅是写代码。软件有一个生存周期，面世、成长、进而成熟，最后消亡，被更新的产品所代替。理解这个循环很重要，因为作为一个程序员，只会花一小部分时间写新代码，大部分时间花在修改和调试已有代码上。软件并不存在于真空中，它一定要有文档、要被维护、被增强，并且被卖掉。在本章中，我们看一个由一名程序员编写的小程序项目。需要许多人编写的大项目将在第23章“模块化程序设计”中讨论。虽然最终的代码还不到100行，但所用的原理可以适用于设计有几千行代码的程序。图7.1所示的是软件的生存周期。

设计程序的主要步骤是:

- 需求。当有人有个想法，并要求你来实现它的时候，程序有了诞生的根基。需求文档用很一般的术语来描述用户的要求。
- 规格说明。描述程序要做什么。开始，初步的规格说明叙述程序要做什么。以后，随着程序越来越精练，规格说明也越来越详细。最后，当程序完成时，规格说明也就成为对程序的完整说明。
- 代码设计。程序员负责程序的总体设计。设计工作应该包括主要的算法、类定义、模块的规格说明、文件格式，以及数据结构。

有一件事情再怎么强调也不过分：“动手之前先动脑”。研究表明，好的设计产生的程序量，仅仅是坏设计产生的程序量的1/10。当使用C++时更是如此，因为在C++中，设计好的对象是编写好程序的关键。（在第13章“类的初步讨论”中，你将学习找出对象的方法。）

- 编码。下一步就是写程序了。先写一个原型，然后再填进代码，使之成为一个完整的程序。
- 测试。程序员应该设计一个测试计划，用它来测试程序。如果可能的话，让其他的

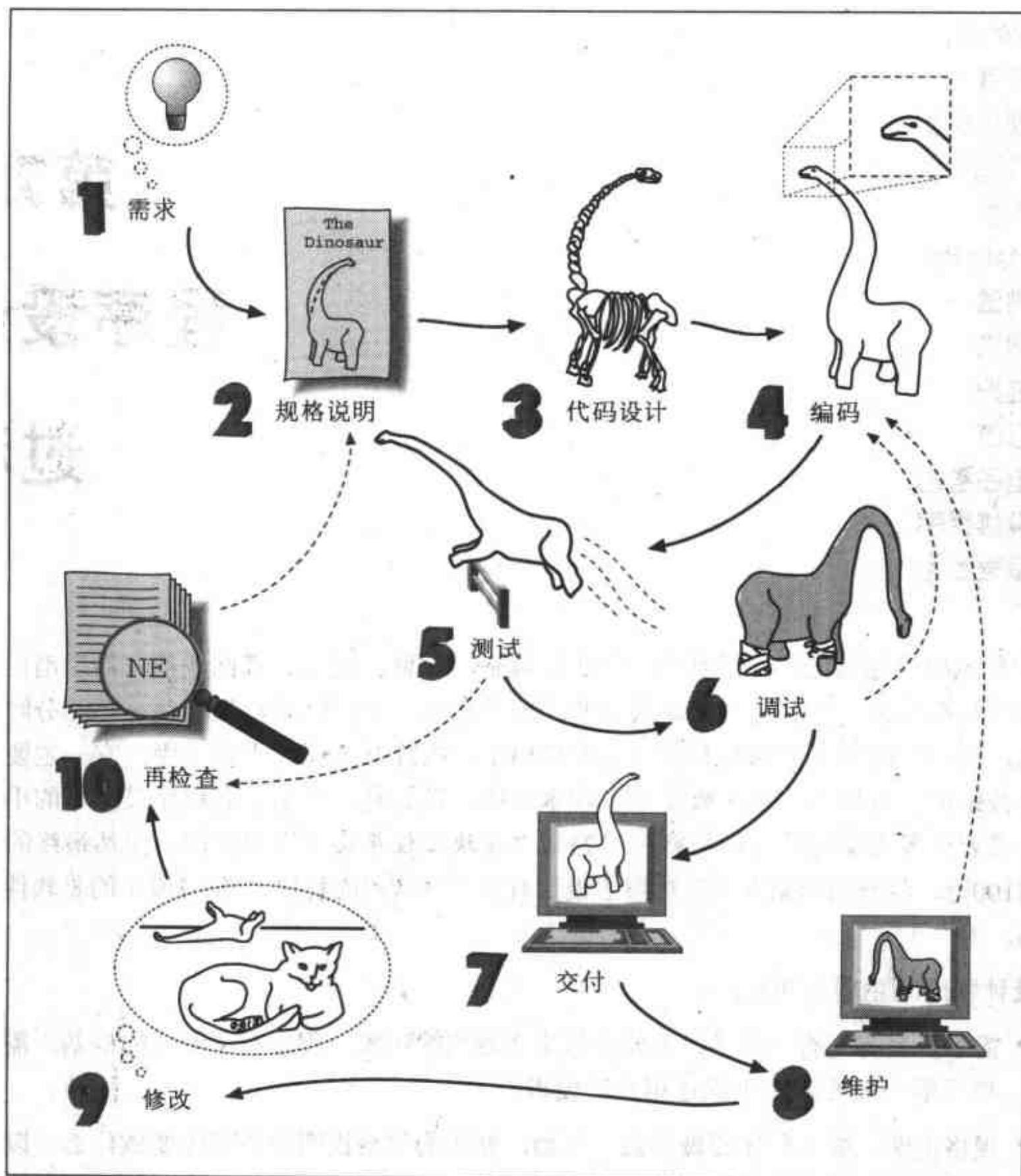


图 7.1 软件的生存周期

人来进行测试是一个好主意。

- 调试。不幸的是，很少有程序能一次通过。这些程序必须修改，然后再进行测试。
- 交付。包装程序，为它准备文档，并送到能发挥作用的地方。
- 维护。程序永远不会完美。总会发现错误，并需要进行修改。
- 修改和升级。程序工作一段时间后，用户可能想改变它，比如，想增加更多的特性或更智能的算法。此时将产生新的规格说明，程序设计过程再周而复始。

设置

操作系统允许你把文件组织在目录中。和文件夹把纸夹在一起放在文件柜中一样，目录把文件放在一起。在本章中，你将建立一个简单的计算器程序。这个程序中用到的所有文件都放在一个目录中，该目录名为`calc`。使用下列命令在UNIX下建立目录：

```
% cd ~  
% mkdir calc
```

在MS-DOS下，键入下列命令：

```
C:\> cd \  
C:\> mkdir calc
```

如果要告诉操作系统你想使用哪个目录，在UNIX下键入下列命令：

```
% cd ~/calc
```

在MS-DOS下，键入：

```
C:\> cd \calc  
C:\CALC>
```

目录的组织方法，可以参见操作系统文档。

规格说明

学习本章之前，我们假定你已接受了任务：“写一个模拟四功能计算器的程序”。一般来讲，这个规格说明不太明确，也不完整。你需要细化它的要求，精确地定义你要产生的程序。

第一步是写一个文档，它叫用户的初步说明(*The Preliminary User's Specification*)。它描述你的程序将做什么，以及它的使用方法。该文档不描述程序的内部结构以及你要使用的算法。四功能计算器的规格说明为：

Calc	Calc
一个四功能的计算器	A four-function calculator
初步的规格说明	Preliminary Specification
1994年12月10日 史蒂夫	Dec.10,1994 Steve Oualline

警告：这是一个初步的规格说明。任何软件之间的任何相似之处纯属巧合。

Calc程序让用户把1万美元的计算机变为不到2美元的四功能计算器。该程序只对整数

进行加、减、乘、除运算。

运行程序时，它把结果寄存器清0，并显示它的内容。然后用户可以键入一个运算符和一个数。结果会有变化，并显示出来。下面的运算是合法的：

运算符	含 义
+	加
-	减
*	乘
/	除

示例（用户输入的内容以黑体字表示）

calc

Result: 0

Enter operator and number: **+ 123**

Result: 123

Enter operator and number: **- 23**

Result: 100

Enter operator and number: **/ 25**

Result: 4

Enter operator and number: *** 4**

Result: 16

初步的规格说明有这样两个目的。第一，你应该把它呈给你的老板（或是顾客），以核实这确实是他的所思所想，并且得到他对你设计的认可。第二，你可以和同事们就这个设计进行商讨，看看他们有什么好的建议和修改。

讨论初步的规格说明，并加入注释：1) “你准备如何退出程序？”，2) “用0作除数时该怎么办？”

因此要加入一个新的运算符：

q - 退出

同时还要加入下面的内容：

用0作除数时显示一条错误信息，同时结果寄存器中的数不变。

IV+III=VII

有一位大学教师，有一次给他的学生布置了一项作业：“实现一个四功能的计算器”。一位学生注意到这个规格说明很不确切，所以决定和教授开个玩笑。教授没有说必须使用什么样的数，所以这位学生写了一个程序，它只能计算罗马数字（IV+III=VII）。程序还有一份完整的用户手册——用拉丁文写的。



代码设计

改进了初步的规格说明后，就可以着手设计代码了。在代码设计阶段，要有个计划。在进行有许多人参与的大项目时，往往每个程序员分到一块任务。此时，要定好文件的格式、数据结构和将采用的主要算法。

这个简单的计算器程序不使用文件，也不需要特殊的数据结构，那么，就只剩下设计主要算法这项工作了。算法用一种伪代码(*pseudo-code*)来描述，所谓伪代码是介于自然语言和真正代码之间的一种简写形式。算法描述为：

```
循环
    读一个运算符和一个数
    做计算
    显示结果
循环结束
```

原型

一旦代码设计完毕，就可以开始写程序了。但是，与其一次写一个完整的程序，然后再调试它，不如使用一个称为快速原型(*fast prototyping*)的方法。该方法是先完成规格说明中最小的部分，然后在此基础上再作努力。本例中，你可以先把四功能计算器简化为单功能计算器。一旦你的这个小程序工作了，就可以把其余的功能加上去。而且，原型还可以让老板先看到程序的样子。当你演示给他看时，他还可以给出进一步的指示。良好的交流是编制良好程序的关键，看你演示的人越多越好。四功能计算器的第一版代码显示在示例7.1中。

示例7.1 *calc/calc.cc*

```
#include <iostream.h>

int    result;    // the result of the calculations
char   oper_char; // the user-specified operator
int    value;     // value specified after the operator

int main()
{
    result = 0; // initialize the result

    // Loop forever (or till we hit the break statement)
    while (1) {
        cout << "Result: " << result << '\n';
```

```

        cout << "Enter operator and number: ";
        cin >> oper_char >> value;

        if (oper_char == '+') {
            result += value;
        } else {
            cout << "Unknown operator " << oper_char << '\n';
        }
    }
    return (0);
}

```

开始时，程序把变量`result`初始化为0。程序的主体是一个循环，它是从下面这条语句开始的：

```
while (1) {
```

当遇到`break`语句时，循环结束。代码：

```

    cout << "Enter operator and number: ";
    cin >> operator >> value;

```

要求用户输入一个运算符和一个数。输入的内容经分析后存入变量`operator`和`value`中。（在第16章“文件输入/输出”中将介绍像`<<`和`>>`这样的I/O运算符。）最后，检查运算符。如果运算符是加（+），就执行加法操作：

```

    if (operator == '+') {
        result += value;
    }

```

现在，这个程序只认识加法运算符。一旦它运行，你可以增加更多的`if`语句，从而为程序增加更多的运算符。

最后，如果输入的是非法运算符，则代码：

```

    } else {
        cout << "Unknown operator " << operator << '\n';
    }

```

将输出错误信息，告诉用户输入错误。

Makefile

输入源代码后，就要编译并连接它了。到目前为止，我们已经手工运行了编译器。这很乏味，而且容易出错。同时，大程序有许多模块，手工编译它们非常困难。所幸的是，UNIX和Turbo-C++都有一个称为`make`的实用工具，它能处理所有的编译工作。

现在，就拿这个例子作示范，用你的程序名取代`calc`。在第23章“模块化程序”设计中将详细讨论`make`程序。`make`先查找名为`Makefile`的文件，了解编译用户程序的方式，并据此运行编译程序。

在UNIX系统中，使用通用的CC编译器，`Makefile`为：

```
[File: calc1/makefile.unx]
#
# Makefile for many UNIX compilers using the
# "standard" command name CC
#
CC=CC
CFLAGS=-g
all: calc

calc: calc.cc
    $(CC) $(CFLAGS) -o calc calc.cc

clean:
    rm calc
```

如果使用的是Free Software Foundation的g++编译器，则`Makefile`为：

```
[File: calc1/makefile.gnu]
#
# Makefile for the Free Software Foundations g++ compiler
#
CC=g++
CFLAGS=-g -Wall
all: calc

calc: calc.cc
    $(CC) $(CFLAGS) -o calc calc.cc

clean:
    rm calc
```

在Turbo-C++系统中，`Makefile`为：

```
[File: calc1/makefile.tcc]
#
# Makefile for Borland's Turbo-C++ compiler
#
CC=tcc
#
# Flags
```

```

#      -N  -- Check for stack overflow
#      -v  -- Enable debugging
#      -w  -- Turn on all warnings
#      -ml -- Large model
#
CFLAGS=-N -v -w -ml
all: calc.exe

calc.exe: calc.cpp
        $(CC) $(CFLAGS) -ecalc calc.cpp

clean:
        erase calc.exe

```

使用Borland C++编译器时, *Makefile*与前一个相同, 只是编译器的名字是**bcc**。
最后, 在Microsoft Visual C++中, *Makefile*为:

```

[File: calc1/makefile.msc]
#
# Makefile for Microsoft Visual C++
#
CC=cl
#
# Flags
#      AL  -- Compile for large model
#      Zi  -- Enable debugging
#      Wl  -- Turn on warnings
#
CFLAGS=/AL /Zi /Wl
all: calc.exe

calc.exe: calc.cpp
        $(CC) $(CFLAGS) calc.cpp

clean:
        erase calc.exe

```

注意

Microsoft Visual C++在它的包装中提供了一个make程序; 但是make命令已改名为nmake。

只需执行命令make就可以编译程序了。(在Microsoft Visual C++下, 使用命令nmake。)make决定将使用哪些编译命令, 然后执行。

make根据文件的最后修改日期来判定是否需要编译它。编译产生一个目标文件。目标

文件的修改日期应该晚于其源文件的修改日期。如果源文件被编辑过，那么它的修改日期将更新，目标文件就过时了。`make`检查这些日期，如果源文件在目标文件生成之后被修改过，那么`make`就重新编译它。

测试

程序编译通过后，就可以进入测试阶段。现在要着手写测试计划了。这个文档很简单，只需列出你要执行的步骤，以确信程序能够工作就可以了。写这个计划有两个原因。

- 如果发现有一个错误，你想能重新产生它。
- 如果改过程序，你想重新测试程序，以保证新的代码不会打乱原来可以工作的程序。

测试计划为：

- + 123 结果应为123
- + 52 结果应为175
- * 37 应该显示出错信息

运行这个程序，得到

```
Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
Result: 175
Enter operator and number: * 37
Result: 212
```

有个地方明显地错了。输入“* 37”后应该得到一条出错信息，但实际上没有。程序中存在一个错误，所以你进入调试阶段。先做一个小的原型程序的好处是，可以尽早地找到错误。

调试

首先检查程序，看看你是不是能找到错误。在这样一个小程序中，这并不困难。但是我们假定，要调试的是一个有21页的程序，你面对的是一个有5000行代码的大得多的程序。这样的程序会使查错异常困难，所以需要进入到下一个步骤。

大多数系统都有C++调试程序，但每个调试器又都有所不同。另外，有些系统没有调试器。本例中，你要使用一条诊断打印语句。（更高级的调试手段将在第17章“调试和优化”中讨论。）这个作法很简单：在你确信数据还是正确的地方（要确保它真的是对的）放一条`cout`语句。然后在数据错误的地方再放一条`cout`语句。运行该程序，重复地放`cout`语句，直到你找到了错误之所在。有诊断`cout`语句的程序如下所示：


```

cout << "Enter operator and number: ";
cin >> value >> operator;

cout << "## after cin " << operator << '\n';

if (operator == '+') {
    cout << "## after if " << operator << '\n';
    result += value;
}

```

注意

每个cout语句中最前面的##表明该行是一个调试行。有了这个标志，就很容易区分临时调试的输出与程序真正的输出。而且，当你在编辑器中找错时，有了##标记后，寻找及移动调试行也容易多了。

再次运行程序，得到：

```

Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
## after cin +
## after if +
Result: 175
Enter operator and number: * 37
## after cin *
## after if +
Result: 212

```

从这里可以看出if语句出了错。不知何故，运算符变量输入的是*，而输出的是+。仔细检查发现，你犯了老毛病，用=代替了==。改正了这个错误后，程序运行正确了。在这个基础上，增加处理其它运算符-、*和/的代码，得到示例7.2。

示例7.2 calc3/calc3.c

```

#include <iostream.h>
int  result;    // the result of the calculations
char oper_char; // the user-specified operator
int  value;     // value specified after the operator
main()
{
    result = 0; // initialize the result

    // loop forever (or until break reached)
    while (1) {
        cout << "Result: " << result << '\n';
    }
}

```

```

        cout << "Enter operator and number: ";

        cin >> oper_char >> value;

        if ((oper_char == 'q') || (oper_char == 'Q'))
            break;

        if (oper_char == '+') {
            result += value;
        } else if (oper_char == '-') {
            result -= value;
        } else if (oper_char == '*') {
            result *= value;
        } else if (oper_char == '/') {
            if (value == 0) {
                cout << "Error: Divide by zero\n";
                cout << "    operation ignored\n";
            } else
                result /= value;
        } else {
            cout << "Unknown operator " << oper_char << '\n';
        }
    }
    return (0);
}

```

包含新运算符后的测试计划如下:

```

+123  结果应为123
+52   结果应为175
*37   应该显示出错信息
-175  结果应为0
+10   结果应为10
/5    结果应为2
/0    除数为0错误
*8    结果应为16
q     退出程序

```

测试这个程序, 你会大吃一惊, 它居然能够运行。从规格说明中删去“初步”一词, 程序、测试计划以及规格说明可以交付了。

维 护

好的程序员在交付程序之前，要让程序经过长期严格的测试。但当第一个用户几乎立刻就发现一个错误时，维护阶段开始了。修正错误，测试程序（确信修正没有破坏任何事情），然后再次交付程序。

修 改

虽然形式上程序写完了，但实际上，你的工作还没有结束。程序使用几个月之后，会有人找到你问：“你能不能加一个取模运算符？”这样你就得修正规格说明、修改程序、更新测试计划、测试程序，再次交付程序。

又过了一段时间，更多的人找到你，要求修改程序。很快，程序具有了三角计算功能、线性回归功能、统计功能、二进制算术功能以及财务计算功能。这个设计是基于单字符运算符的。不久，你会发现你的字符不够用了。此时，程序所能处理的事情远远多于你最初的设计。早晚你会觉得必须废弃这个程序，并写一个新程序取代它。这时，又开始写新的初步规格说明，这样周而复始。

电子考古

不幸的是，大多数程序员并不是从设计这一步开始工作。他们一头扎进维护或是修正工作。这就是说，程序员面对的是最糟糕的事情：理解并修改别人写的代码。

与流行的看法相反，大多数C++程序并不是由缺乏组织的人用神秘古怪的程序设计技术写的，它的注释也不是由世界语写的。它们只是看上去像是那样。电子考古是一门艺术，它研究已有代码，从中找出令人惊讶的东西（如这段代码如何以及为什么会执行）。

你的计算机可以有效地帮助你研究别人代码的真正含义。检测代码时有许多工具可用。（但应小心选择你的工具。许多C工具还没有升级到C++。有关修正程序的问题，请参看前面一节的内容。）这些工具包括：

- 交叉引用。这些程序都有像xref、cxref和cross那样的名字。UNIX系统V有一个实用工具cscope。它们能够打印变量表，还可以指出变量在什么地方使用过。
- 程序缩排器。像cb和indent这样的程序可以“正确”地缩排程序（正确缩排定义由工具提供者给出）。
- 漂亮的打印程序。像vgrind和cprint这样的打印程序可以对源代码进行排版，以方便在激光打印机上的输出。
- 调用图。在UNIX系统V上，程序cflow分析程序的结构。在其它系统上，有一个公共领域实用工具calls，它可以产生调用图，显示谁调用谁，以及谁被谁调用。

- 类浏览器。类浏览器可以显示类的继承关系，这样你可以指明用哪个组件建立类和它的结构。在第13章“类的初步讨论”中，将学习类的概念。

究竟该使用哪个工具？当然是使用适合你的那一个。不同的程序使用不同的方法。下面列出一些检查代码的方法。你可以从中选择一个来用。

编排程序

打印一份程序，并作笔记。使用红蓝墨水笔，以区分打印内容和自己注释的内容。使用亮条把重要的段落标记出来。这些注释都非常有用；把这些注释放到程序中，然后重新打印一份，再从头来一遍。



使用调试器

调试器是一个理解程序的强大工具。大多数调试器允许单步执行程序，即一次执行一行，并检查变量的值，从而明白程序真正的工作方式。一旦你找出代码的真正用意，就记下来，把它们作为注释加到程序中。

使用文本编辑器作为浏览器

从头至尾研究别人的程序时，最好的工具之一是你的文本编辑器。假定你想找出变量 `sc` 用在什么地方。使用查找命令，找出 `sc` 第一次使用的地方。再次查找，找到第二次使用的地方。一直下去，直到你明白该变量的含义为止。

假定你发现 `sc` 是作为数列计数器使用的。因为你已经在编辑器中，所以可以简单地进行全局查找-替换，把变量 `sc` 改为 `sequence_counter`。（**严重警告：**在进行替换之前，确信 `sequence_counter` 没有作为变量来定义，而且保证你进行的是字替换。否则，你会发现你把所有的 `sc` 都替换了，而这却不是你想做的。）给变量说明加注释，这样就可以逐步理解程序了。

加注释

不要害怕把你得到的信息加到注释中，哪怕这些信息微不足道。我使用过的注释包括：

```
int state; // Controls some sort of state machine
int rmxxy; // Something to do with color correction?
```

最后，还有一种引起注意的注释：

```
int idn; // ???
```

它的意思是：“我还没有想好这个变量做什么用。”即使不知道目的，此时也可以做标记，表明还需做更多的工作。

当你向别人的代码中加注释并改进编程风格时，结构将变得越来越清晰。插入注释，则以后的程序员就更容易理解这个程序了。

假定你面对的是如下这样一个程序，它的作者崇尚“越简洁越好”的设计风格。你的任务是指出程序的目的。首先，用铅笔加一些注释，如图7.2所示。

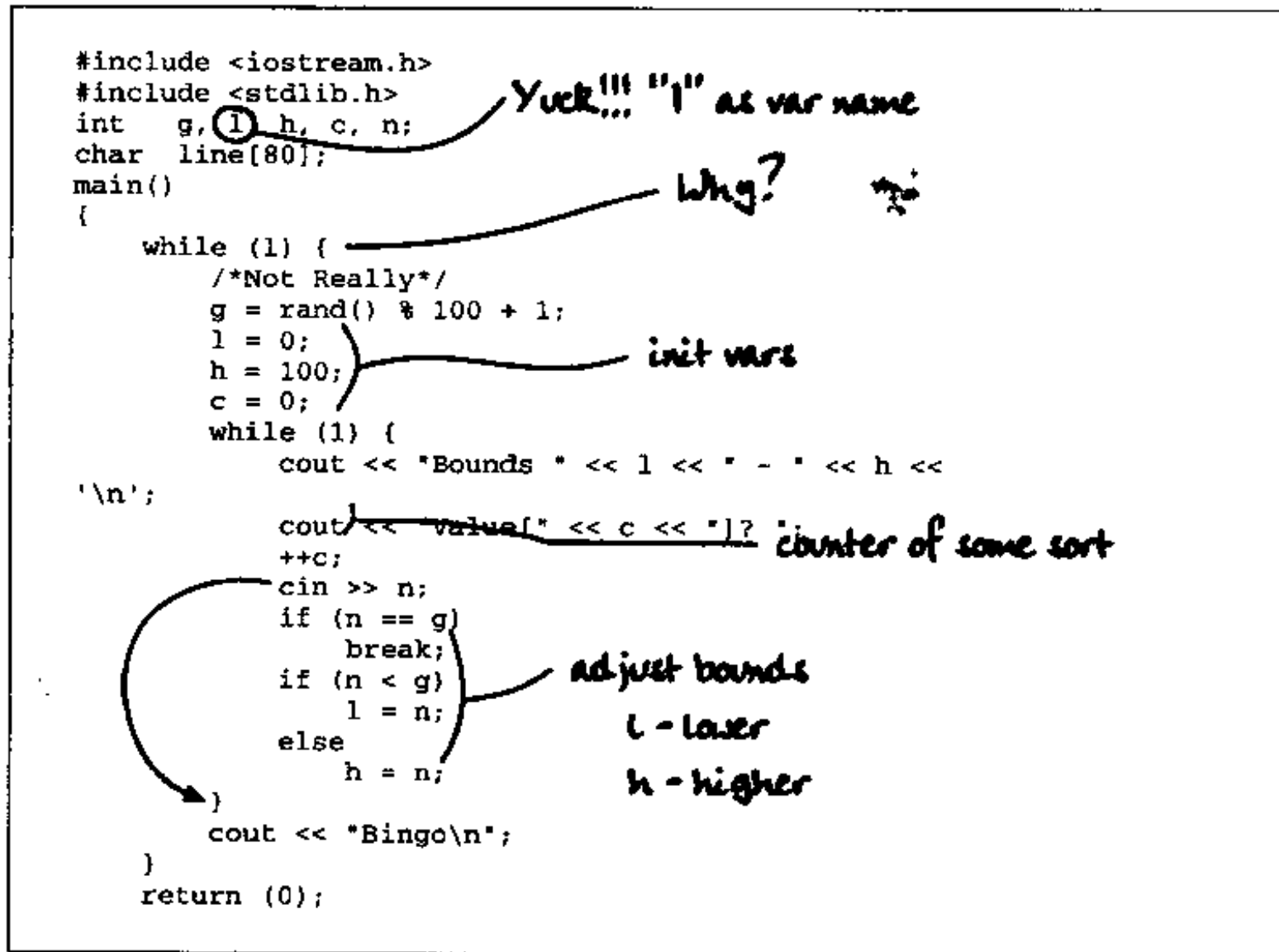


图 7.2 一个简洁的程序

这个程序需要再修改。通读一遍后，把它的原理描述下来，这样就得到一个有良好注释、易于理解的程序，见示例7.3。

```

/*****
 * guess -- a simple guessing game
 *
 * Usage:
 *     guess
 *
 * A random number is chosen between 1 and 100.
 * The player is given a set of bounds and
 * must choose a number between them.
 * If the player chooses the correct number, he wins*

```

```

*      Otherwise, the bounds are adjusted to reflect      *
*      the players guess and the game continues           *
*                                                         *
* Restrictions:                                           *
*      The random number is generated by the statement   *
*      rand() % 100.  Because rand() returns a number    *
*      0 <= rand() <= maxint  this slightly favors      *
*      the lower numbers.                                *
* *****/
#include <iostream.h>
#include <stdlib.h>
int  number_to_guess;  // Random number to be guessed
int  low_limit;        // Current lower limit of player's range
int  high_limit;       // Current upper limit of player's range
int  guess_count;      // Number of times player guessed
int  player_number;    // Number gotten from the player
char line[80];         // Input buffer for a single line
main()
{
    while (1) {
        /*
         * Not a pure random number; see restrictions
         */
        number_to_guess = rand() % 100 + 1;

        // Initialize variables for loop
        low_limit = 0;
        high_limit = 100;
        guess_count = 0;

        while (1) {
            // Tell user what the bounds are and get his guess
            cout << "Bounds " << low_limit << " - " << high_limit << '\n';
            cout << "Value{" << guess_count << "}? ";

            ++guess_count;

            cin >> player_number;

            // Did he guess right?
            if (player_number == number_to_guess)
                break;

            // Adjust bounds for next guess
            if (player_number < number_to_guess)

```

```

        low_limit = player_number;
    else
        high_limit = player_number;

    }
    cout << "Bingo\n";
}
return (0);
}

```

编程练习

对于每一个练习，请按照软件生存周期，完成从规格说明到交付的各步。

练习7.1：编写一个程序，把英制单位转换为公制单位（如：英里转为公里，加仑转为升，等等）。包括规格说明和代码设计。

练习7.2：编写一个程序计算天数，如90年6月1日到92年8月3日之间有多少天。包括规格说明和代码设计。

练习7.3：串行传输线每秒钟可以传送960个字符。编写一个程序，计算传送一个给定大小的文件时需要多长时间。用一个400MB（419,430,400字节）的文件来测试这个程序。使用合适的单位。（一个400MB的文件要按天来计算。）

练习7.4：编写一个程序，对一给定金额，加上8%的税，并对结果在分一级进行四舍五入。

练习7.5：编写一个程序，判定一个数是否是素数。

练习7.6：编写一个程序，统计一串数中正数和负数的个数。

第二部分

简单的程序

设计

本章内容:

- for语句
- switch语句
- switch、break和continue
- 编程练习
- 答疑

第8章

再次讨论 控制语句

for 语句

for语句允许你指定一段代码的执行次数。for语句的一般形式是:

for (初始语句; 条件; 重复语句)

语句体;

它等价于:

初始语句;

while (条件) {

语句体;

重复语句;

}

例如, 示例8.1使用while循环累加5个数。

示例8.1 *total6/total6w.cc*

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

```
int total;          // Total of all the numbers
int current;        // Current value from the user
int counter;        // While loop counter
```

```
main() {
```



```

    total = 0;

    counter = 0;
    while (counter < 5) {
        cout << "Number? ";

        cin >> current;
        total += current;

        ++counter;
    }
    cout << "The grand total is " << total << '\n';
    return (0);
}

```

该程序还可以用for语句重写，见示例8.2。

示例8.2 *total6/total6.cc*

```

#include <iostream.h>

int total;      // Total of all the numbers
int current;    // Current value from the user
int counter;    // For loop counter

main() {
    total = 0;
    for (counter = 0; counter < 5; ++counter) {
        cout << "Number? ";

        cin >> current;
        total += current;
    }
    cout << "The grand total is " << total << '\n';
    return (0);
}

```

注意，counter从0变到4。一般情况下，程序将累加1、2、3、4、5。如果你从0开始考虑累加，将更适合于C++；这样，累加的将是0、1、2、3、4。（从1开始累加，是导致数组上溢错误的主要原因。见第5章“数组、修饰符和读数”。）

仔细研究这两种写法，你会发现它们很类似，详见图8.1。

许多程序设计语言不允许你改变循环中控制变量的值（本例中是counter）。C++并没有严格要求这一点。你可以随时改变控制变量的值——你可以转入循环，也可以从循环中转出，去做PASCAL和FORTRAN程序员不敢做的那些事情。（虽然C++给了你做这些傻事的自由，但不意味着你就要去做。）

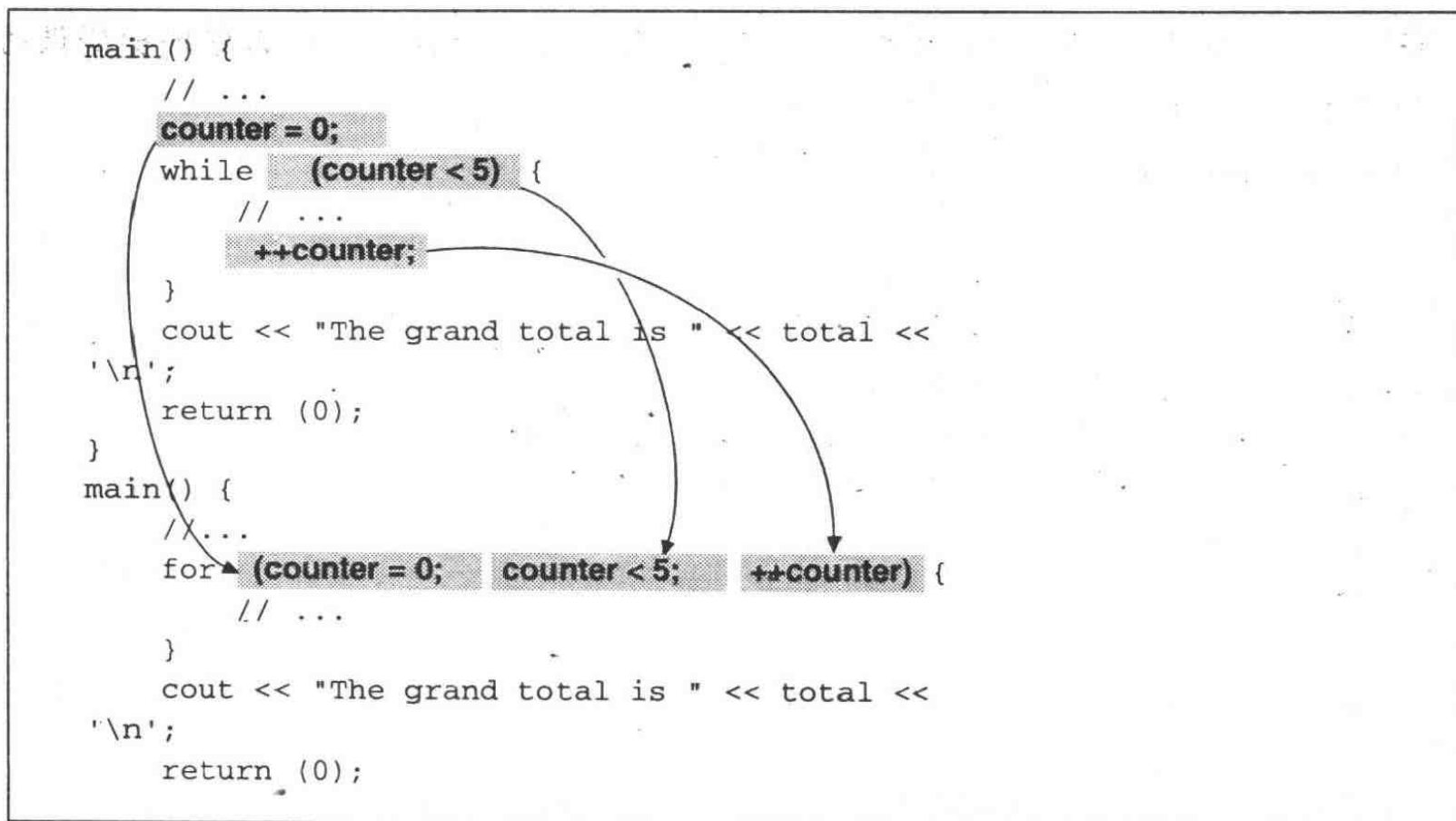


图 8.1 while和for的相似点

问题8.1: 示例8.3包含了一个错误。

示例8.3 cent/cent.cc

```

#include <iostream.h>
/*
 * This program produces a Celsius to Fahrenheit conversion
 * chart for the numbers 0 to 100.
 *
 * Restrictions:
 * This program deals with integers only, so the
 * calculations may not be exact.
 */

// The current Celsius temperature we are working with
int celsius;
main() {
    for (celsius = 0; celsius <= 100; ++celsius);
    cout << "Celsius: " << celsius <<
        " Fahrenheit: " << ((celsius * 9) / 5 + 32) << '\n';
    return (0);
}

```

运行时, 程序打印出:

Celsius: 101 Fahrenheit: 213

就再没有别的了。这是为什么?

问题8.2: 示例8.4读入5个数, 并统计其中出现的3和5的个数。为什么它给出错误的答案?

示例8.4 `seven/seven.cc`

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

```
int seven_count;    // Number of sevens in the data
int data[5];        // The data to count 3 and 7 in
int three_count;    // Number of threes in the data
int index;          // Index into the data
```

```
main() {
    seven_count = 0;
    three_count = 0;

    cout << "Enter 5 numbers\n";
    cin >> data[1] >> data[2] >> data[3] >>
        data[4] >> data[5];

    for (index = 1; index <= 5; ++index) {
        if (data[index] == 3)
            ++three_count;
        if (data[index] == 7)
            ++seven_count;
    }
    cout << "Threes " << three_count << " Sevens " << seven_count << '\n';
    return (0);
}
```

运行该程序时, 我们输入数据: 3 7 3 0 2, 得到的结果为:

Threes 4 Sevens 1

(你的答案可能多种多样。)

switch语句

switch语句类似于一串if-else语句。switch语句的一般形式为:

```
switch (表达式) {
    case 常量1:
        语句
        ....
        break;
```

```

    case 常量2:
        语句
        ....
        // Fall through
default:
    语句
    ....
    break;
case 常量3:
    语句
    ....
    break;
}

```

switch语句计算表达式的值，然后转入case标号之一。标号是不允许重复的，所以只能选中一个case标号。表达式的结果必须为整数、字符或是枚举量。

case标号必须是常量，但次序不限。default标号可以放在switch中的任何地方。

当C++遇到switch语句时，它先计算表达式的值，然后查找匹配的case标号。如果没有找到，就使用default标号。如果没有default，该语句就什么都不做。

switch中的break语句告诉计算机继续执行switch之后的语句。如果没有break语句，则继续执行下一条语句。

注 意

switch语句类似于PASCAL中的case语句。主要的差别是，PASCAL中只允许在标号后放一个语句，而C++允许写许多条。C++可以一直执行下去，直到它遇到break语句为止。在PASCAL中，你不能从一个case中一直执行到另一个case中。但在C++中，却可以。

在第7章“程序设计过程”中有个计算器程序，其中含有一系列的if-else语句。

```

if (operator == '+') {
    result += value;
} else if (operator == '-') {
    result -= value;
} else if (operator == '*') {
    result *= value;
} else if (operator == '/') {

```

```

        if (value == 0) {
            cout << "Error: Divide by zero\n";
            cout << "    operation ignored\n";
        } else
            result /= value;
    } else {
        cout << "Unknown operator." << operator << '\n';
    }
}

```

这段代码很容易改写为switch语句。在switch中，对每一种操作，使用不同的case语句。default子句将处理所有非法的操作。

使用switch语句重写这个程序，不仅使程序更简单，也更易读，程序见示例8.5。

示例8.5 calc-sw/calc3.cc

```

#include <iostream.h>
int    result;        // The result of the calculations
char   oper_char;     // The user-specified operator
int    value;         // Value specified after the operator

main()
{
    result = 0;                // Initialize the result

    // Loop forever (or until break reached)
    while (1) {
        cout << "Result: " << result << '\n';
        cout << "Enter operator and number: ";
        cin >> oper_char >> value;

        if ((oper_char == 'q') || (oper_char == 'Q'))
            break;

        switch (oper_char) {
            case '+':
                result += value;
                break;
            case '-':
                result -= value;

                break;
            case '*':
                result *= value;
                break;
            case '/':
                if (value == 0) {

```

```

        cout << "Error: Divide by zero\n";
        cout << "    operation ignored\n";
    } else
        result /= value;
    break;
default:
    cout << "Unknown operator " << oper_char << '\n';
    break;
}
}
return (0);
}

```

在case的结尾并不一定要有break语句。如果没有break语句的话，将继续执行下一条语句。

例如：

```

control = 0;

// A not so good example of programming
switch (control) {
    case 0:
        cout << "Reset\n";
    case 1:
        cout << "Initializing\n";
        break;
    case 2:
        cout << "Working\n";
}

```

本例中，当control == 0时，程序打印出：

```

Reset
Initializing

```

在case 0中没有break语句。打印完"Reset"后，程序继续执行下一条语句（case 1），并打印"Initializing"。

但这样的语法有个问题。你不能肯定是程序原本被设计为从case 0执行到case 1，还是程序员忘记写break语句了。为了消除这个疑虑，case部分最好还是写上break语句，或者写上注释"//fall through"。

```

// A better example of programming
switch (control) {
    case 0:

```



```

        cout << "Reset\n";
        // Fall through
    case 1:
        cout << "Initializing\n";
        break;
    case 2:
        cout << "Working\n";
}

```

因为case 2在最后，所以不一定非得要break语句。break语句总是使程序跳到switch的结尾，但现在我们已经执行到那了。

假定我们稍微改动一个这个程序，在switch中增加另一个case语句：

```

// We have a little problem
switch (control) {
    case 0:
        cout << "Reset\n";
        // Fall through
    case 1:
        cout << "Initializing\n";
        break;
    case 2:
        cout << "Working\n";
    case 3:
        cout << "Closing down\n";
}

```

这时，如果control == 2，程序将输出：

```

Working
Closing down

```

这很令人讨厌。问题出在case 2不再是最后一个case语句了。我们继续执行了。（我们并不想这样，或者说，我们应该加上一条“// Fall through”的注释。）break语句现在是必需的。如果你总是写上break语句，就没有必要担心是不是真地需要它了。

```

// Almost there
switch (control) {
    case 0:
        cout << "Reset\n";
        // Fall through
    case 1:
        cout << "Initializing\n";
        break;
    case 2:

```

```

        cout << "Working\n";
        break;
    }

```

最后，我们问这样一个问题：当`control == 5`时，会出现什么情况？本例中，因为没有相匹配的`case`或是`default`子句，所以跳过整个的`switch`语句。

本例中，程序员没有包含`default`语句，因为`control`不会是0、1、2以外的值。但是，变量可以被赋以很奇怪的值，所以我们要再加上一些语句，以防备这种情况的出现。

```

// The final version
switch (control) {
    case 0:
        cout << "Reset\n";
        // Fall through
    case 1:
        cout << "Initializing\n";
        break;
    case 2:
        cout << "Working\n";
        break;
    default:
        cout << "Internal error, control value " << control <<
            " impossible\n";
        break;
}

```

虽然并不需要`default`语句，但应该在每个`switch`语句中写一条。哪怕`default`语句仅仅像下面这样：

```

default:
    // Do nothing
    break;

```

所以，还是应该有才好。这表明你会忽略掉范围以外的所有数据。

switch、break和continue

`break`语句有两种用法。一种是在`switch`语句中，它使程序退出`switch`语句。另一种是在`for`或是`while`循环中，它使程序退出循环。`continue`语句只用在循环中，它使程序跳到循环的开头。

让我们修改计算器程序，来说明这些语句是如何使用的。只有当输入合法数据后新程序才输出结果，它还有一条`Help`命令。

Help命令很特别。我们不想在Help命令之后输出结果，所以，在Help case后我们没有放break语句，而是放了一条continue语句。continue强迫程序跳到循环的开头。

当输入一个未知运算符时，我们输出一条出错信息。在Help case中，使用一条continue语句，跳过结果的输出。

最后，还有一条特殊命令：quit。该命令在switch外进行处理。它执行循环开头的break语句。因为break在switch的外面，所以它属于while循环，并使程序退出while循环。

该程序的控制流如图8.2中所示。

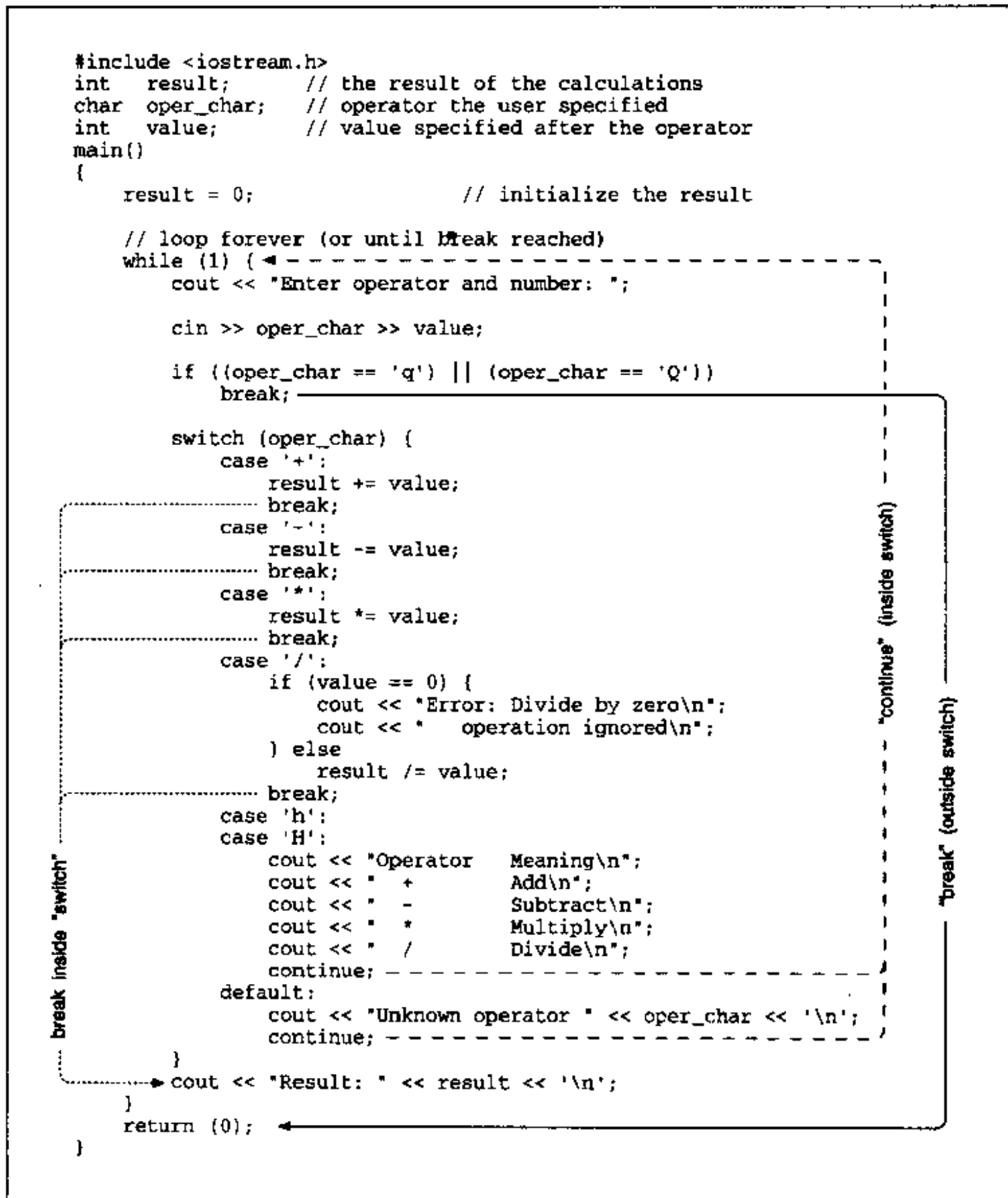
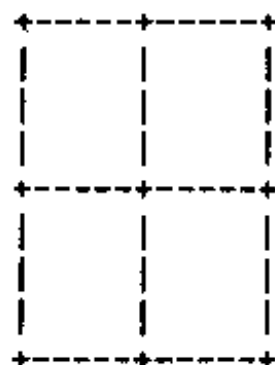


图 8.2 switch/continue

编程练习

练习8.1: 打印一张棋盘 (8*8)。每个方块为5个字符宽，3个字符高。下面是2*2棋盘的示例:



练习8.2: n 个并联电阻的总电阻的计算公式为:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \dots + \frac{1}{R_n}$$

假定我们有两个并联电阻组成的电网，这两个电阻的阻值分别为 $400\ \Omega$ 和 $200\ \Omega$ 。计算公式为:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$$

代入值得:

$$\frac{1}{R} = \frac{1}{400} + \frac{1}{200}$$

$$\frac{1}{R} = \frac{3}{400}$$

$$R = \frac{400}{3}$$

所以这个电网的总电阻为 $133.3\ \Omega$ 。

编写一个程序计算任意个电阻并联后的总电阻。

练习8.3: 编写程序计算 n 个数的平均数。

练习8.4: 编写程序打印乘法表。

练习8.5: 编写程序，读入一个字符，判定它是元音字母还是辅音字母，并输出结果。

练习8.6: 编写程序，把数转换为字。例如：895转换为“eight nine five”。

练习8.7: 修改练习8.6中的程序, 使得在输出结果时, 不仅仅输出对应的数, 还要输出它所在的位, 例如: 输入13时, 输出thirteen; 输入100时, 输出one hundred。

答 疑

问题8.1: 问题出在for语句的结尾有一个分号。for的语句体是在右圆括号与分号之间。本例中, 语句体为空。虽然cout语句是缩进的, 但也不是for语句的一部分。缩进是错误的, C++编译器并不看它是否缩进。在表达式:

```
centigrade <= 100
```

等于假 (centigrade == 101) 之前, 程序什么也不做。然后执行cout语句。

问题8.2: 问题出在我们把数读到data[1]到data[5]中。在C++中, 合法的数组范围是0到<数组大小>-1, 也就是说, 是从0到4。data[5]是非法的。当我们使用非法的数组元素时, 会出现令人不可思议的怪现象; 在本例中, 变量three_count的值被改变了。解决的方法是, 使用data[0]到data[4]。

本章内容:

- 作用域和存储类
- 函数
- 参数类型一览
- 结构化程序设计基础
- 递归
- 编程练习
- 答疑

第9章

变量作用域 和函数

到目前为止,你一直都在使用全局变量(global variable)。这些变量几乎可以在程序的任何地方赋值并使用。本章中,你将了解其它类型的变量,并学习它们的使用方法。本章还将介绍如何把代码划分为函数。我们将讨论函数的有关内容,包括函数过载、用函数建立结构化的程序以及递归函数的调用方法。

作用域和存储类

所有的变量都有两个属性:作用域(scope)和存储类(storage class)。一个变量的作用域是指程序中变量有效的区域。一个全局变量从说明开始到程序的结尾都是有效的。局部变量(local variable)的作用域局限在说明它的块内,在这个块外就不能访问(赋值或读出)它了。一个块(block)是括在花括号({})内的一段代码。图9.1说明了局部和全局变量的差别。

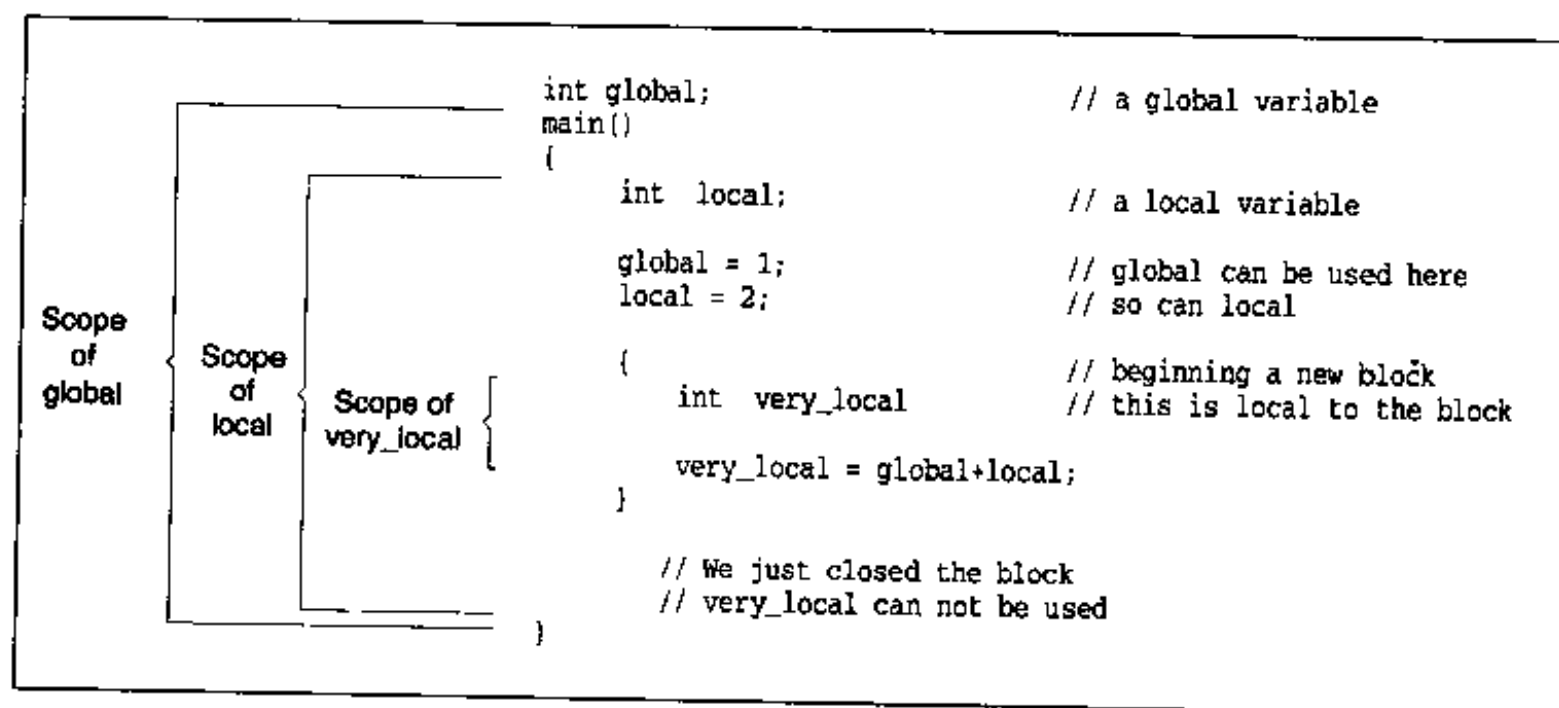


图 9.1 局部和全局变量

局部变量可能会与全局变量重名。一般地，变量count（图9.2中第一次说明的那一个）的作用域应该是整个程序。第二个说明，即局部count，在它被说明的这个小块内要优先于全局变量。在这个块内，全局count被隐藏起来。你还可以嵌套局部说明，从而隐藏局部变量。这些“非常局部”的变量比“一般的局部”变量有更小更局部的作用域。（前面介绍的简洁原则会让你记起：使用嵌套的办法来隐藏局部变量并不会使程序更易懂。）图9.2说明一个被隐藏的变量。

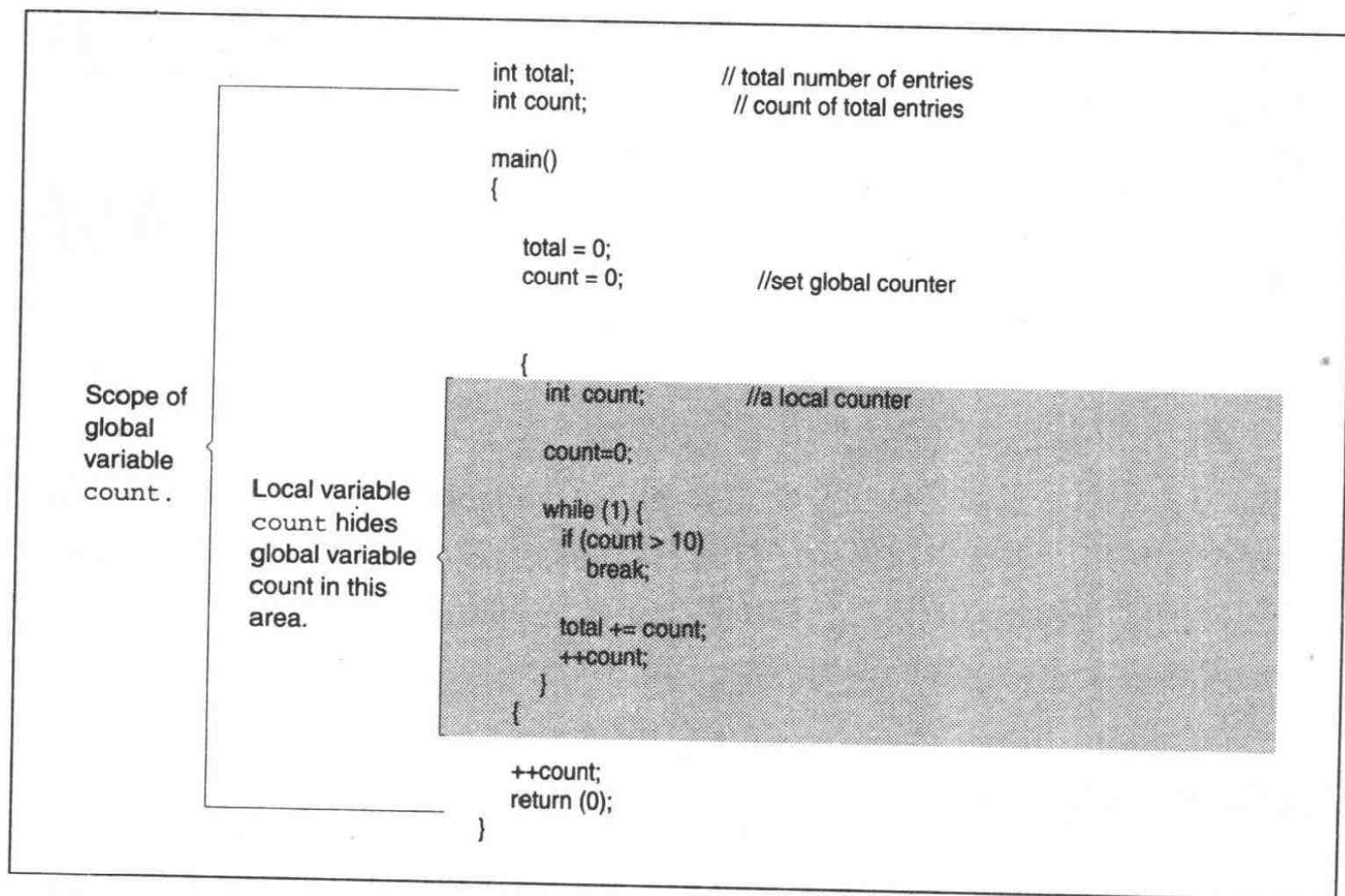


图 9.2 被隐藏的变量

变量count作为局部变量和全局变量都被说明了。一般地，count（全局）的作用域是整个程序，但当在一个块内说明了一个变量时，变量的实例就变为活动的那一个。在这个块内，全局count已被局部count所隐藏。图中阴影部分表示全局count被隐藏的区域。

隐藏变量并不是好的编程方法。当你写这样的语句时，问题就暴露出来了：

```
count = 1;
```

很难判定你指的是哪个count。它是在main开头说明的那个全局count，还是在while循环中间说明的那个变量？最好给这些变量取不同的名字，如total_count、current_count以及item_count。

变量的存储类既可以是永久（permanent）的，也可以是暂时（temporary）的。全局变量永远是永久的。它们在程序启动之前建立并被赋初值，并且一直保留到程序的结束。一个块开始时说明的临时变量从被称为栈(stack)的一段内存中分配空间。如果你分配了太多的临时变量，会导致栈上溢错误(stack overflow error)。在块结束时，临时变量使用的空间将

返回给栈。每次进入这个块时，临时变量都将被初始化。

栈的大小依赖于使用的系统和编译器。在许多UNIX系统中，程序自动地尽最大可能分配栈空间。在其它系统中，缺省的栈大小可以由编译器开关来改变。在Turbo-C++中，栈空间不能大于64,000字节。看上去这是个很大的空间，但几个大数组很快会把它装满。你应该考虑把所有的大数组说明为永久变量。

如果局部变量没有用static来说明的话，它将是临时的。

注 意

当static用在全局变量时，它的意义完全不同。（这表明一个变量局部于当前文件。）见第23章“模块化程序设计”。有关static不同含义的详细讨论，请见表14.1。

示例9.1说明了临时和永久变量的不同。我们选择使用含义明确的变量名：**temporary**是一个临时变量，而**permanent**是永久变量。在每次建立**temporary**变量时（在for语句块的开始），C++都要给它赋初值；而**permanent**只在程序启动时，一次性地初始化。

在循环中，两个变量都进行加1操作。但到循环开头时，**temporary**被初始化为1。

示例9.1 perm/perm.cc

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

```
main() {
    int counter;    // Loop counter

    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;
        static int permanent = 1;

        cout << "Temporary " << temporary <<
            " Permanent " << permanent << '\n';
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

The output of this program is:

Temporary 1 Permanent 1

Temporary 1 Permanent 2

Temporary 1 Permanent 3

注 意

临时变量有时又称作自动变量(*automatic variable*)，因为编译器自动为它们分配存储空间。修饰词**auto**可以用来表明一个临时变量；但在实际中，**auto**几乎从来不用。

表9.1列出一个变量说明时的几种不同方式。

表9.1 变量的说明

说 明	作用域	存储类	初始化
在所有的块外	全局	永久	一次性
在所有的块外并用static	全局	永久	一次性
在一个块内	局部	临时	每次进入块时
在一个块内并用static	局部	永久	一次性

注 意

关键字static是C++中过载意义最多的运算符。根据它的使用方式，它有很多意义。表14.1中列出了它的全部用法。

函 数

使用函数(function)，你就可以把常用的代码组织为一个紧凑的单元，这个单元可以重复地使用。你已经见过一个函数了，这就是main。它是一个特殊的函数，在程序的开始时被调用。其它所有的函数都直接或间接地从main调用。

假定你想写一个程序，计算三个三角形的面积。你可以写三次计算公式；或者，可以建立一个函数，然后三次调用这个函数。每个函数都应该有包含下列内容的注释块：

名字 函数名

描述 对函数功能的描述

参数 对函数中每个参数的描述

返回值 对函数返回值的描述信息

还可以加入另外的内容，如文件格式、引用或是注解。其它的内容，请参见第3章“风格”。

计算三角形面积的函数的开头为：

```
/******  
* Triangle -- compute area of a triangle *  
*  
* Parameters *  
* width -- width of the triangle *  
* height -- height of the triangle *  
* .. *  
*/
```

```

* Returns                                     *
*  area of the triangle                       *
*****/

```

函数本身以下面这一行开始：

```
float triangle(float width, float height)
```

`float`是函数类型。它定义了函数返回值的类型。`width`和`height`是函数的临时变量。参数是局部于函数的变量，它们用来向函数传递一些信息。

注 意

在C++中，函数类型并不是必需的。如果没有说明函数类型，缺省的类型为`int`。但是，如果你不写函数类型，就会弄不清是想让函数使用缺省的`int`类型，还是忘记写函数类型了。为了避免有这样的疑惑，记住永远写清函数类型，而不要使用缺省值。

函数使用下列语句计算面积：

```
area = width * height / 2.0;
```

剩下的问题就是把结果传给调用者。使用`return`语句可以完成这项工作：

```
return (area)
```

完整的三角函数见示例9.2。

示例9.2: `tri/tri-sub.cc`

```

/*****
 * Triangle -- compute area of a triangle *
 *                                           *
 * Parameters                               *
 *  width -- width of the triangle         *
 *  height -- height of the triangle       *
 *                                           *
 * Returns                                  *
 *  area of the triangle                   *
 *****/

float triangle(float width, float height)
{
    float area; // Area of the triangle

    area = width * height / 2.0;

```



```
    return (area);
}
```

代码行:

```
size = triangle(1.3, 8.3)
```

来调用triangle函数。当C++见到这个调用时，它执行下面的操作:

Triangle的变量 width = 1.3

Triangle的变量 height = 8.3

开始执行triangle函数的第一行。

这一类的参数传递，在技术上被称为“值调用”。当调用函数时，才进行参数的赋值操作，所以参数的数据流向只有一个：输入。

函数中的数据通过return语句传出来。在triangle这个例子中，函数给局部变量area赋值5.4，然后执行return (area)语句。这样，函数的返回值就是5.4。这个值赋给size。

```
return (area);    5.4 (The value of area)
// .....
size = triangle(1.3, 8.3)
```



示例9.3计算三个三角形的面积。

示例9.3 tri/tri.cc

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

```
main()
```

```
{
```

```
    // Function to compute area of triangle
```

```
    float triangle(float width, float height);
```

```
    cout << "Triangle #1 " << triangle(1.3, 8.3) << '\n';
```

```
    cout << "Triangle #2 " << triangle(4.8, 9.8) << '\n';
```

```
    cout << "Triangle #3 " << triangle(1.2, 2.0) << '\n';
```

```
    return (0);
```

```
}
```

```
/* *****
```

```
* Triangle -- compute area of a triangle *
```

```
*
```

```
* Parameters *
```

```
* width -- width of the triangle *
```

```
* height -- height of the triangle *
```

```
*
```

```
* Returns *
```

```

*   area of the triangle                               *
*****/
float triangle(float width, float height)
{
    float area; // Area of the triangle

    area = width * height / 2.0;
    return (area);
}

```

函数也要像变量一样被说明。说明可以把函数的返回值和参数的信息告诉C++编译器。函数说明有两种方式。一种是，在函数使用之前，写一个完整的函数。另一种是，定义函数原型(*function prototype*)，函数原型为编译器提供了调用函数时的足够信息。函数原型看上去很像函数的第一行，它是没有函数体的函数。例如，函数triangle的原型为：

```
float triangle(float width, float height);
```

注意，这一行的最后要有一个分号。它告诉C++这是一个函数原型，而不是真正的函数。

在说明原型时，C++还允许你省掉参数名字。可以仅简单地这样写：

```
float triangle(float, float);
```

但是，这种写法不常使用。因为建立原型有一种非常简单的方法，就是在编辑器中拷贝函数的第一行，然后放在说明原型的地点。（在头文件中会出现多次，我们将在第23章“模块化程序设计”中介绍头文件。）而且，在原型中写名字会使读程序的人得到额外的有用信息。

说明没有参数的函数时，其参数列表为void。例如：

```
int get_value(void);
```

void结构延续了原来C中的写法，在C中，空参数表“()”表明是原来K&R风格的C函数原型。实际上，C++既允许你写一个空的参数表，也允许你写void说明，但void更好一些。void就表示：“该函数没有参数”。

关键字void还用来指明没有返回值的函数（类似于FORTRAN的SUBROUTINE或是PASCAL的Procedure）。例如，下面的函数只打印出一个结果，它没返回任何值：

```

void print_answer(int answer)
{
    if (answer < 0) {
        cout << "Answer corrupt\n";
        return;
    }
}

```

```

        cout << "The answer is " << answer '\n';
    }

```

const参数和返回值

如果一个参数被说明为是**const**类型的，则表明这个参数在这个函数内是不能改变的。普通的参数可以在函数内部被改变，但改变的值不会回传给调用函数。

例如，在**triangle**函数中，我们从不改变**width**或**height**的值。它们就可以被说明为是**const**的。因为返回值也是不能被改变的，所以也可以说明为是**const**。**const**说明可以提醒程序员：参数在函数内部不被修改。如果你试图改变一个**const**参数，编译器会出错。示例9.4是使用了**const**说明的**triangle**函数。

示例9.4 *tri/tri-sub2.cc*

```

const float triangle(const float width, const float height)
{
    float area; // Area of the triangle

    area = width * height / 2.0;
    return (area);
}

```

正像它表明的那样，返回值的**const**说明仅仅是个摆设。在下一节中，将介绍如何返回引用参数的值，并使**const**返回说明有用。

引用参数和返回值

在第4章“基本说明和表达式”中，我们讨论过引用参数。引用参数为变量说明了又一个名字。对于全局和局部变量来说，引用参数不太有用。但是，当用作参数时，它们有完全不同的含义。

假定你想写一个让**counter**加1的程序，如果你的代码如示例9.5那样，那么它将不会运行。

示例9.5 *value/value.cc*

```

#include <iostream.h>
// This function won't work
void inc_counter(int counter)
{
    ++counter;
}

main()
{
    int a_count = 0;    // Random counter

```

```

    inc_counter(a_count);
    cout << a_count << '\n';
    return (0);
}

```

为什么呢？因为C++中缺省的调用方式是值调用。这意味着，值只能传进不能传出。

如果参数counter变为引用参数又如何呢？引用是给同一个变量命名两个名字的另一方法。当调用inc_counter时，counter变为a_count的一个引用变量。这就是说，对counter的任何操作都反映在a_count上。使用引用参数的示例9.6工作正常。

示例9.6 value/ref.cc

```

#include <iostream.h>
// Works
void inc_counter(int &counter)
{
    ++counter;
}

main()
{
    int a_count = 0;    // Random counter

    inc_counter(a_count);
    cout << a_count << '\n';
    return (0);
}

```

检查该程序，我们发现它很像示例9.5，只是在counter的前面加了一个“&”符号。这个“&”告诉C++，counter是一个引用参数，它不是通常的值调用参数。

引用说明还可用在返回值上。例如，示例9.7寻找数组中的最大元素。

示例9.7 value/big.cc

```

int &biggest(int array[], int n_elements)
{
    int index; // Current index
    int biggest; // Index of the biggest element

    // Assume the first is the biggest
    biggest = 0;
    for (index = 1; index < n_elements; ++index) {
        if (array[biggest] < array[index])
            biggest = index;
    }
}

```

```

        return (array[biggest]);
    }

```

如果想打印数组中的最大元素，所要做的就是：

```

int item_array[5] = {1, 2, 5000, 3, 4}; // An array

cout << "The biggest element is " <<
    biggest(item_array, 5) << '\n';

```

我们仔细看看这些代码。首先考虑，当你产生一个引用变量时会出现什么现象：

```

int &big_reference = item_array[2]; // A reference to element #2

```

引用变量`big_reference`是`item_array[2]`的另一个名字。你可以用这个引用变量来输出一个值：

```

cout << big_reference << '\n';    // Print out element #2

```

但是因为它是一个引用变量，你还可以把它用在赋值号的左边：

```

big_reference = 0;    // Zero the largest value of the array

```

函数`biggest`返回一个引用变量。当用于`item_array`时，它返回`item_array[2]`的引用变量。记住，在下面的代码中，`biggest(item_array, 5)`就是`item_array[2]`。下面三段代码执行的操作是等价的。真正的变量`item_array[2]`没有被改变，但使用引用的方式却有所改变。

```

// Using the actual data
cout << item_array[2] << '\n';
item_array[2] = 0;

// Using a simple reference
int big_reference = &item_array[2];
cout << big_reference << '\n';
big_reference = 0;

// Using a function that returns a reference
cout << biggest(item_array, 5) << '\n';
biggest(item_array, 5) = 0;

```

因为使用`biggest`的代码段中返回的是一个引用变量，所以它可以用在赋值号(=)的左边。但假定你不想这样，那么你可以这样写，即返回一个`const`型的引用。

```

const int &biggest(int array[], int n_elements);

```

这行代码告诉C++即使你返回的是一个引用变量，其结果也不能被改变。所以下面这样

的代码是非法的:

```
biggest(item_array, 5) = 0;    // Now it generates an error
```

悬空的引用

当使用“引用返回”时,应该非常小心。如果不小心,可能会引用到一个已不存在的变量。示例9.8说明了这个问题。

示例9.8 *ref/ref.cc*

```
1 #include <iostream.h>
2
3 const int &min(const int &i1, const int &i2)
4 {
5     if (i1 < i2)
6         return (i1);
7     return (i2);
8 }
9
10 main()
11 {
12     int &i = min(1 + 2, 3 + 4);
13
14     return (0);
15 }
```

第3行定义了函数`min`。它返回对两个数中较小者的引用。在第12行,我们调用这个函数。在调用`min`之前,C++建立一个整型的临时变量来存放表达式`1 + 2`的值。对该临时变量的引用作为参数`i1`传送给`min`函数。C++为`i2` 参数建立另一个临时变量。然后调用函数`min`,并返回对`i1`的引用。但`i1`指的是什么呢?它是C++在`main`中建立的一个临时变量。当调用语句结束时,C++已经取消了所有的临时变量。

让我们再仔细地研究一下对`min`的调用(第12行)。下面是第12行代码所对应的伪代码,包含了C++编译器没有向程序员公开的详细内容:

```
create integer tmp1, assign it the value 1 + 2
create integer tmp2, assign it the value 3 + 4
bind parameter i1 so it refers to tmp1
bind parameter i2 so it refers to tmp2
call the function "min"
bind main's variable i so it refers to
    the return value (i1-a reference to tmp1)
// At this point i is a reference to tmp1
destroy tmp1
destroy tmp2
```



```
// At this point i still refers to tmp1
// It doesn't exist, but i refers to it
```

在第12行结束时，我们遇到了这样一个处境：i指向一个已消失的临时变量。换句话说，i指向了不存在的东西。这就叫悬空的引用(*dangling reference*)，程序员应该避免使用。

数组参数

到目前为止，你见到的都只是简单参数。C++在处理数组参数时略有不同。首先，在原型说明时，你不必写数组大小。例如：

```
int sum(int array[]);
```

C++使用称为“地址调用”的参数传递机制来传递数组。另一方面，C++自动地把所有的数组参数转为引用参数。这样就能传递任意大小的数组。我们刚刚说明的函数sum，可以接受的数组大小为3、43、5000或者任意。

但是，你也可以写上数组大小。不论你写的是什么，C++都忽略掉。不过写上的话，可以提醒读程序的人，这个函数只接受固定大小的数组。

```
int sum(int array[3]);
```

对于多维数组，你必须把除最后一维外的所有维的大小都写上。之所以这样，是因为C++要使用这些维数来计算数组中每个元素的分配地址。

```
int sum_matrix(int matrix1[10][10]); // Legal
int sum_matrix(int matrix1[10][]);   // Legal
int sum_matrix(int matrix1[][10]);    // Illegal
```

问题9.1： 示例9.9中的函数应该计算一个串的长度*。但它把所有串的长度都计算为0。为什么？

示例9.9 length/length.cc

```
/* *****
 * length -- compute the length of a string
 *
 * Parameters
 *     string -- the string whose length we want
 *
 * Returns
 * ***** */
```

* 该函数（当它正确工作时）与库函数strlen的功能相同。

```

*           the length of the string           *
*****/
int Length(char string[])
{
    int index;           // Index into the string

    /*
     * Loop until we reach the end-of-string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */
    return (index);
}

```

函数重载

下面我们定义一个简单的函数，它返回一个整数的平方：

```

int square(int value) {
    return (value * value);
}

```

我们还想计算浮点数的平方：

```

float square(float value) {
    return (value * value);
}

```

现在我们有二个函数，它们有相同的名字。这样做非法吗？在像C和PASCAL这样的语言中，这确实是非法的。在C++中，却是对的。C++允许函数重载(*function overloading*)，也就是说，你可以定义具有相同名字的二个函数。所以你可以定义一个square函数，它处理所有的类型：int、float、short int、double。如果我们知道一个字符的平方是什么意思的话，甚至还可以定义字符的平方。

为了保证代码的一致性，使用同一名字的所有函数都应该执行同一种基本功能。例如，你可以定义下面二个square函数：

```

// Square an integer
int square(int value);

// Draw a square on the screen
void square(int top, int bottom, int left, int right);

```

这是完全正确的C++代码，但是读程序的人会迷惑不解。

函数重载还有一个限制：C++必须能区分函数。例如，下面的代码是非法的：


```
int get_number(void);  
float get_number(void); // Illegal
```

问题在于，C++使用参数表来区分函数。但是两个`get_number`函数的参数表是一样的，`(void)`。这样，C++就分不清两个函数了，所以第二个说明是错误的。

缺省参数

假定你想定义一个在屏幕上画矩形的函数。该函数还必须能按需要成比例地缩放矩形。函数定义为：

```
void draw(const rectangle &rectangle; double scale)
```

这个函数用过一段之后，你会发现在90%的情况下，你不使用`draw`中按比例缩放功能。换句话说，90%的情况下，缩放因子为1.0。

这时，C++允许你为`scale`指定一个缺省值。语句：

```
void draw(const rectangle &rectangle; double scale = 1.0)
```

告诉C++：“如果没有指定`scale`的值，它就取1.0”。所以下面的两行是等价的：

```
draw(big_rectangle, 1.0); // Explicitly specify scale  
draw(big_rectangle);      // Let it default to 1.0
```

使用缺省参数时，在设计风格上有点问题。研究下面的代码：

```
draw(big_rectangle);
```

你能说出程序员是想让`scale`等于1.0呢，还是忘了写这个参数了呢？虽然缺省参数有时是有用的，但还是不要滥用。

未用过的参数

如果你定义了一个参数，但没有用它，大多数好的编译器会输出一条警告信息。例如：

```
void exit_button(Widget &button) {  
    cout << "Shutting down\n";  
    exit (0);  
}
```

上面这段代码将产生下面的警告信息：

```
Warning: line 1. Unused parameter "button"
```

但是当你真地不想用这个参数时怎么办？有没有办法让C++缄口不说，不来打扰你？有。办法就是不写参数名。

```
// No warning, but style needs work
void exit_button(Widget &) {
    cout << "Shutting down\n";
    exit (0);
}
```

这段代码对C++合适，但对读程序的人来说，就不合适了。我们只能明白exit_button有一个Widget &参数，但参数是什么？解决这个问题的办法是，用注释写清参数的名字。

```
// Better
void exit_button(Widget & /*button*/) {
    cout << "Shutting down\n";
    exit (0);
}
```

有人认为这样写不好看，并且容易混淆。他们是对的，这段代码确实不易读。本该有更好的办法；我希望我能想出来。

现在你可能会问一个问题：“为什么我不能写这样的代码？为什么就不能不写参数名？”

其原因是，许多程序设计系统都使用回调函数(callback function)。例如，你可以告诉X Windows系统，“当按下‘EXIT’按钮时，调用exit_button函数”。你的回调函数可能处理许多按钮，这样，知道按下的是哪个按钮就很重要了。所以X Windows把按钮作为函数的一个参数。

如果你知道只有唯一的一个按钮可以使X调用exit_button时怎么办？好，X仍然把它传给你，你忽略它就是了。这就是有些函数有未使用的参数的原因。

内联函数

再回头看用于整数的square函数，我们看到它是个非常短的函数（只有一行）。当C++调用一个函数时，都会有些额外开销。这包括参数入栈、进入和退出函数以及函数返回后栈的整理。

例如，下面的代码：

```
int square(int value) {
    return (value * value);
}
main() {
    // .....
    x = square(x);
}
```

在68000机器上会生成如下的汇编代码（示意性的）。

```

label "int square(int value)"
    link a6,#0                // Set up local variables

    // The next two lines do the work
    movel a6@8,d1             // d1 = value
    muls1 a6@8,d1             // d1 = value * d1

    movel d1,d0               // Put return value in d0
    unlk a6                   // Restore stack
    rts                       // Return(d0)
label "main"
//....
//    x = square(x)
//
    movel a6@(-4),sp@         // Put the number x on the stack
    jbsr "void square(int value)"
                                // Call the function

    addqw #4,sp               // Restore the stack
    movel d0,a6@4             // Store return value in X
// ...

```

正如你从代码中看到的，对于两行工作代码，要多负担8行间接代码。C++使用内联函数就可以减少间接代码的开销。**inline**关键字告诉C++这个函数非常小。也就是说，C++编译器把完整的函数体放到代码流中，比生成对函数的调用要更简单也更容易。

```

inline int square(int value) {
    return (value * value);
}

```

修改square函数：

```

label "main"
// ...
//    x = square(x)
//
    movel d1,a6@(-4)          // d1 = x
    movel a6@4,d0             // d0 = x
    muls1 d0,d0                // d0 = (x * x)

    movel d0,a6@4             // Store result

```

把函数说明为内联的，可以减少额外的8行代码，使运行更快了。

inline修饰符给C++提供了一条线索，编译器在生成代码时可以使用。**inline**告诉编译器代码非常小而且简单。和**register**一样，**inline**也是一个暗示性的修饰符。如果C++编译器没

有产生内联函数，它就把这个函数作为一个普通函数来对待。

参数类型一览

表9.2列出了不同的参数类型。

表9.2 参数类型

类 型	说 明
按值调用	<code>function(int var)</code> 把值传送给函数，且值在函数内可以被修改，但这些修改并不回传给调用函数。
常量按值调用	<code>function(const int var)</code> 把值传送给函数，但值不能被修改。
引用	<code>function(int &var)</code> 把引用传送给函数。对参数的任何修改都映射到调用函数中的对应量。
常量引用	<code>function(const int &var)</code> 在函数内值不能被修改。对于复杂数据类型来说，这种形式的参数比常量按值调用的效率更高。（见第12章“高级类型”。）
数组	<code>function(int array[])</code> 值传入函数内，并且可能要被改变。C++自动地把数组转为引用参数。
按地址调用	<code>function(int *var)</code> 把指针传给一个项。指针将在第15章“简单指针”中介绍。

结构化程序设计基础

计算机科学家花了许多年的时间研究究竟应该如何设计程序。结果，他们提出了许多确实不错的程序设计方法——一个月有一种。这些方法包括：流程图、自顶向下的程序设计、自底向上的程序设计、结构化的程序设计和面向对象的程序设计。

现在你已经了解了函数，我们可以用结构化程序设计技术(*structured programming technique*)来设计程序。这种方法是把一个程序划分为更小更容易设计的函数。这样，程序易写也易理解。我认为这不是最好的方法，但却是最适合于我的方法。如果其它的方法适合于你，那就采用吧。

结构化的程序设计把注意力集中在程序的代码上。在后面你将学习如何把代码和数据

结合为类，从而转向面向对象的程序设计。

程序设计的第一步决定你要做什么。这已在第7章“程序设计过程”中讨论过。下一步，决定如何构造你的数据。

最后，是代码设计阶段。写论文时，我们先写一份提纲，论文中的每个部分都用一句话来描述；稍后再补充详细的内容。写程序与此类似。你要先写一份提纲，不过这份提纲是你的main函数。细节可以在其它函数内完成。例如，示例9.10可以解决世界上的所有问题。

示例9.10 A global solution

```
main()
{
    void init(void);
    void solve_problems(void);
    void finish_up(void);

    init();
    solve_problems();
    finish_up();
}
```

当然，细节留待以后解决。

从写main函数着手，不应该超过两页纸。如果它变长了，考虑把它划分为两个更小更简单的函数。函数的大小不应该超过三页纸，因为人一次短期记忆的内容最多就只有这么多。在main函数完成之后，可以着手解决其它的函数。这样的结构化程序设计称为自顶向下的程序设计(top-down programming)。从顶(main)开始，向下开展工作。

另一种设计方法是自底向上的程序设计方法(bottom-up programming)。这种方法的第一步是先写最底层的函数，测试之后，把这些函数组合在一起。当我编写一个以前没有用过的新标准函数时，我试着使用了这种方法。这就是我们在第7章中编写计算器程序时使用过的方法。

实际上，两种方法都是有用的。大多数情况下采用自顶向下的方法，部分情况下采用自底向上的方法。计算机科学家对此有一个术语：混用。编程时应该遵从的一条原则是：“使用最奏效的那种。”

递 归

递归就是函数直接或间接地调用它自己。有些函数本身很适于递归算法，比如计算阶乘。

递归函数必须遵从下面两条原则：

1. 它必须有结束点。

2. 使问题变得更简单。

阶乘的定义为:

```
fact(0) = 1
fact(n) = n * fact(n-1)
```

对应的C++函数为:

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* else */
    return (number * fact(number-1));
}
```

它满足上述的两条原则。第一, 它有一个结束点 (当 `number == 0` 时)。第二, 它简化了问题, 因为 `fact(number-1)` 比 `fact(number)` 更简单。

当 `number >= 0` 时, 阶乘是有定义的。但如果我们要计算 `fact(-3)` 时, 会发生什么事情? 程序会因栈上溢而异常中断。`fact(-3)` 调用 `fact(-4)`, 而 `fact(-4)` 调用 `fact(-5)`, 依此类推, 没有结束点。这就是无穷递归错误 (*infinite recursion error*)。

有许多需要重复做的事情都可以采用递归来实现, 比如累加一个数组中的所有元素值。你可以定义一个函数, 它累加数组中下标从 `m` 到 `n` 的各元素:

如果你只有一个元素, 则累加操作很简单。否则, 把第一个元素的值和其余元素的累加和加起来。

对应的C++函数为:

```
int sum(int first, int last, int array[])
{
    if (first == last)
        return (array[first]);
    /* else */
    return (array[first] + sum(first + 1, last, array));
}
```

例如:

```
Sum(1 8 3 2) =
  1 + Sum(8 3 2) =
    8 + Sum(3 2) =
      3 + Sum(2) =
        2
      3 + 2 = 5
```



```
8 + 5 = 13
1 + 13 = 14
Answer = 14
```

编程练习

练习9.1: 编写一个程序，统计一个串中字的个数。（你的文档中应该准确描述字的定义。）编写一个程序，测试这个新过程。

练习9.2: 编写一个函数**begins(string1, string2)**，如果string1的开始是string2的话，返回值为真。编写一个程序测试这个函数。

练习9.3: 编写一个函数**count(number, array, length)**，它统计number出现在array中的次数。array是一个有length个元素的数组。函数应该采用递归方法。编写一个测试程序，测试这个函数。

练习9.4: 编写一个程序，它读入一个字符串，并把串中每个字符的值累加起来作为该串的简单哈希码，返回该值。

练习9.5: 编写一个程序，返回数组中的最大值。

练习9.6: 编写一个函数，扫描一个串，把其中的“_”替换为“_”。

答疑

问题9.1: 程序员在解释这个什么也不做（除了index加1）的for循环时会遇到很多麻烦。因为在for的最后没有分号，所以C++读入一个完整的语句（此处是return(index)），并把它放到for循环中。示例9.11是改进版本。

示例9.11 *length/rlen.cc*

```
int length(char string[])
{
    int index;          // index into the string

    /*
     * Loop until we reach the end of string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */ ;
    return (index);
}
```

本章内容:

- #define语句
- 条件编译
- #include文件
- 参数宏
- 高级特征
- 小结
- 编程练习
- 答疑

第10章

C++预处理器

第一个C编译器没有常量或是内联函数。但还在开发C的时候,就已有迹象表明,它需要处理命名的常量、宏,并能包括文件。解决的办法是建立一个预处理器,在程序提交给C编译器之前,先由预处理器来处理。预处理器就是一个专用的文本编辑器。它的语法完全不同于C,它完全不懂C的结构。它只是一个简单的文本编辑器。

预处理器很有用,并且很快就被合并到主要的C编译器中。C++编译器也保留了这个预处理器。在有些系统中,例如UNIX中,它仍然是一个独立的程序,它由编译器包cc自动执行。有些较新的编译器,像Turbo-C++,有内置的预处理器。

#define语句

#define语句用来定义一个常量。例如,下面两行执行类似的功能:

```
#define SIZE 20          // The array size is 20
const int SIZE = 20;    // The array size is 20
```

实际上,第一行是给预处理器的一条命令,把SIZE的值永远改为20。这样就不用再变值了。

预处理器的所有命令的第一列都是井号(#)。C++是自由格式的语言。语言元素可以放在一行的任何地方,行尾处理为一个空格。预处理器却不是自由格式的。它根据第一列的井号(#)来进行判断。正像你在下面将看到的,预处理器并不了解C++的语法,它也可以编辑C++以外的任何程序。

警告

预处理器并不是C++编译器的一部分。它使用了完全不同的语法,而且要想用好它,也应采取完全不同于C++的态度。出现的大多数问题都是因为你把它当作C++了。

在预处理器中,行的结束即意味着语句的结束。在C++中,分号才表示语句的结束。而

分号并不表示预处理器中语句的结束，语句中加入分号，只能使结果出人意料。在行尾写反斜杠（\），表示该行没有结束。预处理器最简单的用法是定义一个替换宏。例如，命令：

```
#define FOO bar
```

会使预处理器在每次遇到“FOO”时都用“bar”来取代。使用全大写的字符表示宏名字是常用的编程方法。这样可以容易地区分变量（全小写）与宏（全大写）。

简单的define语句的一般格式为：

```
#define 名字 替换内容
```

名字可以是任意的C++合法标识符。替换内容可以是单行内放得下的任意内容。替换内容可以包括空格、运算符和其它字符。

下面的定义：

```
#define FOR_ALL for (i = 0; i < ARRAY_SIZE; ++i)
```

可以这样使用：

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

宏的这种使用方式并不好。它们混淆了程序的基本控制流。本例中，如果程序员想了解循环做什么，他就必须翻到程序的开头，查看FOR_ALL的定义。

定义宏来大规模地替换C++的基本程序结构，也不是好方法。例如，你可以如下定义：

```
#define BEGIN {
#define END }
...
    if (index == 0)
        BEGIN
            cout << "Starting\n";
        END
```

这样，你不是用C++来编程，而是用半C++半PASCAL的混合语言编程。

因为预处理器不检查C++程序的语法是否正确，所以可能引起出人意料的问题。例如，示例10.1在第11行产生一个错误。

示例10.1 *big/big.cc*

```
1 #define BIG_NUMBER 10 ** 10
2
```

```

3 main()
4 {
5     // Index for our calculations
6     int    index;
7
8     index = 0;
9
10    // Syntax error on next line
11    while (index < BIG_NUMBER) {
12        index = index * 8;
13    }
14    return (0);
15 }

```

问题出在第1行`#define`语句，但错误信息指向第11行。因为有第1行的定义，所以预处理器把第11行展开为：

```
while (index < 10 ** 10)
```

因为`**`是一个非法的运算符，所以出现语法错。

问题10.1：下面的程序得到的结果是47，而不是预计的144。为什么？（提示在下面。）

示例10.2 `first/first.cc`

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

```

#define FIRST_PART    7
#define LAST_PART     5
#define ALL_PARTS     FIRST_PART + LAST_PART

```

```

main() {
    cout << "The square of all the parts is " <<
        ALL_PARTS * ALL_PARTS << '\n';
    return (0);
}

```

提示：

```
CC -E prog.cc
```

把预处理器的输出送给标准输出。

在MS-DOS/Windows中，命令：

```
cpp prog.cpp
```

会建立一个名为`prog.i`的文件，它含有预处理器的输出。

预处理器运行示例10.1后，会给出信息：

示例10.3 first/first-ed.out

```
# 1 "first.cc"
# 1 "/usr/local/lib/g++-include/iostream.h" 1 3

// About 900 lines of #include stuff omitted

inline ios& oct(ios& i)
{ i.setf(ios::oct, ios::dec|ios::hex|ios::oct); return i; }

# 1 "first.cc" 2

main() {
    cout << "The square of all the parts is " <<
        7 + 5 * 7 + 5 << '\n';
    return (0);
}
```

注意

C++预处理器的输出包含了许多信息，大部分可以忽略掉。本例中，你要查看这些输出内容，直到找到cout行为止。检查这一行内容，就会知道引起错误的原因了。

问题10.2： 示例10.4产生一条警告信息：counter is used before it is set。这有点令人吃惊，因为 for 循环应该设置了它的值。你还会得到一条非常奇怪的警告：“null effect”，它指向第11行。

示例10.4 max/max.cc

```
1 // Warning, spacing is VERY important
2
3 #include <iostream.h>
4
5 #define MAX=10
6
7 main()
8 {
9     int counter;
10
11     for (counter = MAX; counter > 0;
12         --counter)
13         cout << "Hi there\n";
14
15     return (0);
16 }
```

提示：看看预处理器的输出。

问题10.3：示例10.5计算size的值是错误的。为什么？

示例10.5 size/size.cc

```
#include <iostream.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
main()
{
    int size; // Size to really use

    size = FUDGE;
    cout << "Size is " << size << '\n';
    return (0);
}
```

问题10.4：下面的程序应该在得到坏数据时，输出“Fatal Error: Abort”信息，并退出。但它在得到好数据时也退出了。为什么？

示例10.6 dis/dis.cc

```
1 #include <iostream.h>
2 #include <stdlib.h> /* ANSI Standard only */
3
4 #define DIE \
5     cerr << "Fatal Error: Abort\n"; exit(8);
6
7 main() {
8     // A random value for testing
9     int value;
10
11     value = 1;
12     if (value < 0)
13         DIE;
14
15     cout << "We did not die\n";
16     return (0);
17 }
```

#define与const

const关键字相对新一些。在const之前，#define是定义常量的唯一方法，所以大多数原来的代码使用的是#define命令。但是，const的使用更优于#define。有下面几个原因。首先，C++立即检查const语句的语法；而在宏使用之前不会检查#define。第二，const使用C++语

法，**#define**有其自己的语法。最后一个原因，**const**遵从C++一般的作用域规则，而由**#define**定义的常量永远都可用。

大多数情况下，**const**语句要比**#define**语句好。定义常量有两种方法。

```
#define MAX 10 // Define a value using the pre-processor
               // (This can easily cause problems)
const int MAX = 10; // Define a C++ constant integer
               // (Safer)
```

#define命令只局限于定义简单常量。**const**语句可以定义C++中几乎所有类型的常量，包括结构类。例如：

```
struct box {
    int width, height; // Dimensions of the box in pixels
};
const box pink_box(1.0, 4.5); // Size of a pink box to be used for
                               input
```

但是对于条件编译和其它特殊用途方面，**#define**命令却是非常重要的。

条件编译

编程时有一个这样的问题，即写出的代码需能在许多不同机器上运行。理论上，C++代码是可移植的，但实际上，许多机器都有需要解决的一些问题。例如，本书介绍了UNIX、MS-DOS和Windows编译器。虽然它们几乎相同，但也有若干不同点，这些问题将在第25章“移植问题”中介绍。

在修改使用条件编译(*conditional compilation*)生成的代码时，预处理器给了用户最大的灵活性。假定你想在调试阶段加入调试代码，而在最终提交的产品中删除调试代码，那么，你可以在程序中加入如下的语句：

```
#ifdef DEBUG
    cout << "In compute_hash, value " << value << " hash " << hash <<
        "\n";
#endif /* DEBUG */
```

注意

你不必非在**#endif**后面写上**/* DEBUG */**，但它是一条很有用的注释。

如果程序的开头有下列命令：

```
#define DEBUG /* Turn debugging on */
```

则程序中就会包含前面提到的**cout**语句。如果程序包含如下的命令：

```
#undef DEBUG      /* Turn debugging off */
```

则cout语句就被略过了。

严格地讲，**#undef DEBUG**并不是必需的。如果没有**#define DEBUG**语句，自然就不用定义**DEBUG**。**#undef DEBUG**语句用来明确地告诉阅读代码的人，**DEBUG**用于条件编译，并且现在处于关闭状态。

如果符号没有定义，则命令**#ifndef**会编译随后的代码。**#else**把条件变反。例如：

```
#ifdef DEBUG
    cout << "Test version. Debugging is on\n";
#else /* DEBUG */
    cout << "Production version\n";
#endif /* DEBUG */
```

程序员或许会临时删去一段代码。常用的方法是把这段代码用**/* */**框住。有时，这样做会带来一些问题，看下面的例子：

```
/* **** Comment out this section
    section_report();
    /* Handle the end-of-section stuff */
    dump_table();
**** End of commented out section */
```

在第5行有一个语法错误。为什么？

使用**#ifdef**结构把这段代码删去，这样做会更好一些。

```
#ifdef UNDEF
    section_report();
    /* Handle the end-of-section stuff */
    dump_table();
#endif /* UNDEF */
```

（当然，如果别人定义了**UNDEF**，这段代码也不会被删去，但一般不会有这么巧。）

编译器开关**-D**符号允许在命令行定义这个符号。例如，命令：

```
CC -DDEBUG -g -o prog prog.cc
```

将编译程序prog.c，这其中包括出现在**#ifdef DEBUG**到**#endif /* DEBUG */**之间的所有代码，而不管在程序中有没有**#define DEBUG**。Turbo-C++中等价的命令是：

```
tcc -DDEBUG -g -N -eprog.exe prog.c
```

该选项的一般形式是**-D**符号或是**-D**符号=值。例如下面的语句给**MAX**赋值10：

```
CC -DMAX=10 -o prog prog.c
```


大多数C++编译器自动地定义某些与系统有关的符号。例如，Turbo-C++定义了符号 `__TURBOC__`，而MS-DOS定义了 `__MSDOS__`。ANSI标准编译器C定义了 `__STDC__`，C++编译器定义了 `__cplusplus`。大多数UNIX编译器定义了一个用于系统的名字（如，Sun、VAX、celerity等等）。但是，在文档中很少提到这些。符号 `unix` 为所有的UNIX机器所使用。

注意

命令行选项仅指定一个符号的初始值。程序中的任何 `#define` 和 `#undef` 命令都可以改变这个符号的值。例如，下列命令：

```
#undef DEBUG
```

将取消DEBUG的定义，不管你是否使用了-DDEBUG。

#include 文件

`#include` 命令可以让程序使用其它文件中的源代码。例如，在程序中可以使用如下的命令：

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

它通知预处理器将文件 `iostream.h` 插入到当前程序中。包含在其它程序中的文件叫头文件 (*header file*)。 (大部分 `#include` 命令都放在程序的开头。) 尖括号表明该文件是一个标准的头文件。在UNIX中，这些文件通常放在 `/usr/include` 目录下。在MS-DOS/Windows中，这些文件放在Turbo-C++目录下 (按照安装时指定的名字)。

标准的包含文件用来定义库函数所用的数据结构和宏。例如，`cout` 是个标准类 (就你目前所知)，它在标准输出设备上输出信息。`cout` 使用的 `ostream` 类定义及相关的程序都在 `iostream.h` 中有定义。

有时，用户可能要写自己的包含文件。当一个程序扩展在几个文件中时，局部包含文件特别有用，它们用来存放要用的常量和数据结构。当一组程序员共同开发一个项目时，为了共享信息，就要使用局部包含文件。(见第23章“模块化程序设计”。)

局部包含文件的文件名一般由双引号 (") 来指定。

```
#include "defs.h"
```

文件名 ("defs.h") 可以是任何有效的名字。它可以是一个简单的文件 "defs.h"，也可以含有相对路径 `../data.h`，或者是一个绝对路径 `/root/include/const.h`。(在MS-DOS/Windows中，可以用反斜杠 (\) 来代替斜杠 (/) 用作目录分隔符。)

包含文件有可能嵌套，因此也可能带来问题。假定你在文件 `const.h` 中定义了几个有用的常量，如果文件 `data.h` 和 `io.h` 都包含 `const.h`，程序写法如下：

```
#include "data.h"
#include "io.h"
```

因为预处理器两次设置const.h中的定义，所以产生错误。一个常量定义了两次并不是个致命的错误；但是，两次定义一种数据结构或是联合就是个错误了，应该避免。

解决这个问题的一种办法是检查const.h，看看是否已包含了它，并且不要定义已定义过的任何符号。

看下面的代码：

```
#ifndef _CONST_H_INCLUDED_

/* Define constants */

#define _CONST_H_INCLUDED_
#endif /* _CONST_H_INCLUDED_ */
```

当包含const.h时，它定义了符号_CONST_H_INCLUDED_。如果这个符号已经定义过了（因为前面已包含过这个文件），#ifndef条件就把其它的定义隐藏起来，这样就不会发生问题了。

注意

有时也可能把代码放在头文件中，但这种方法不好。按惯例，代码应该放在.cc文件中，而把定义、说明、宏和内联函数放在.h文件中。

参数宏

我们仅讨论了简单的#define和宏。宏可以带有参数。下面的宏计算一个数的平方：

```
#define SQR(x) ((x) * (x)) /* Square a number */
```

使用时，宏用跟随其后的自变量取代x。例如，SQR(5)变为((5)*(5))。请记住，在宏的参数两侧要放一对括号。否则有可能导致错误，见示例10.7：

示例10.7 sqr/sqr.cc

```
#include <iostream.h>
#define SQR(x) (x * x)
{
    int counter;    // Counter for loop

    for (counter = 0; counter < 5; ++counter) {
        cout << "x " << counter + 1 <<
            " x squared " << SQR(counter + 1) << '\n';
```



```

    }
    return (0);
}

```

问题10.5: 上面的程序会输出什么内容? (试着在计算机上运行一次。) 为什么会输出这样的信息? (检查预处理器的输出。)

使程序简单的编程规则不让我们使用增量(++)和减量(--)运算符, 除非它们单独在一行中。当在表达式中使用时, 它们会带来副作用, 从而得出意外的结果, 见示例10.8。

示例10.8 *sqr-i/sqr-i.cc*

```

#include <iostream.h>
#define SQR(x) ((x) * (x))

main()
{
    int counter;    /* Counter for loop */

    counter = 0;
    while (counter < 5)
        cout << "x " << counter + 1 <<
            " x squared " << SQR(++counter) << '\n';
    return (0);
}

```

为什么它没有得出预计的结果? 每次counter加的是几?

示例10.8中的SQR(++counter)扩展为((++counter) * (++counter))。这样, 在每次循环中, counter的值增2。这个表达式的实际结果依系统而定。

问题10.6: 运行下面的程序时显示有一个没有定义的变量, 可我们只有一个变量counter。为什么?

示例10.9 *rec/rec.cc*

```

#include <iostream.h>
#define RECIPROCAL (number) (1.0 / (number))

main()
{
    float counter;

    for (counter = 0.0; counter < 10.0;
        counter += 1.0) {

        cout << "1/" << counter << " = " <<
            RECIPROCAL(counter) << "\n";
    }
}

```

```

    }
    return (0);
}

```

#运算符

#运算符用在条件宏中，用来把自变量转为一个串。例如：

```

#define STR(data) #data
STR(hello)

```

产生

"hello"

有关这个运算符的详细用法，请参见第26章“组合在一起”。

参数宏与内联函数

在大多数情况下，最好使用内联函数而不要用参数宏，这样可以避免参数宏隐藏的问题。但是，有些情况下，参数宏比内联函数更好。例如，SQR宏可以用于float和int数据类型。而如果我们要用内联函数实现的话，就必须写两个函数。

```

#define SQR(x) ((x) * (x)) // A parameterized macro
// Works, but is dangerous

// Inline function to do the same thing
inline int sqr(const int x) {
    return (x * x);
}

```

高级特征

本书并不打算介绍C++预处理器的所有命令。更高级的特征包括用于条件编译的#if命令的高级形式，和用于在文件中插入依赖于编译器的指令的#pragma命令。有关这些特征的详细信息请参见C++参考手册。

小 结

C++预处理器是C++语言中一个非常有用的部分。它所表现的形式与C++完全不同。它必须与C++编译器区别对待。

宏定义中出现的问题常常不表现在定义宏的地方，而往往在程序内部才出错。遵从下面的简单规则，可以减少问题的发生。

1. 在每一项的两侧都放一对括号。特别是#define中的常量和宏参数的两侧。
2. 当用多个语句定义一个宏时，要把代码放在花括号{}内。
3. 预处理器不是C++编译器。不要使用=或是;。

```
#define X = 5 // Illegal
#define X 5;  // Illegal
#define X = 5; // Very illegal

#define X 5    // Correct
```

最后，皆大欢喜。如果你做到了这些，你会为不再出问题而高兴。

编程练习

练习10.1: C++没有布尔类型。使用#define来定义BOOLEAN、TRUE和FALSE的值，建立布尔类型。

练习10.2: 编写一个宏，如果它的参数可以被10除尽，则返回真，否则返回假。

练习10.3: 编写一个宏 is_digit，如果它的自变量是一个十进制数字，返回真。编写第二个宏 is_hex，当它的自变量是一个十六进制数字（0~9，A~F，a~f）时，返回真。第二个宏可以引用第一个。

练习10.4: 编写一个预处理器宏，它交换两个整数。（如果你是一个计算机迷，可以不使用在宏外说明的临时变量。）

答疑

问题10.1: 程序经预处理器运行后，cout语句被扩展为：

```
cout << "The square of all the parts is " << 7 + 5 * 7 + 5 << '\n';
```

等式 $7 + 5 * 7 + 5$ 的结果为47。在宏中表达式的所有项的两侧放一对括号() 是一条好的编程规则。如果你把ALL_PARTS的定义改为：

```
#define ALL_PARTS (FIRST_PART + LAST_PART)
```

则程序将正确执行。

问题10.2: 预处理器是一个思维简单的程序。当它定义宏时，标识符后的所有内容都是宏的一部分。本例中，MAX的定义是“= 10”。当展开for语句时，变为：

```
for (counter==10; counter > 0; --countner)
```



C++允许计算一个结果，然后把它扔掉。对该条语句，程序检查counter的值是不是10后就会扔掉答案。在定义中去掉=，问题迎刃而解。

问题10.3：与前面的问题一样，预处理器并不使用C++的语法约定。本例中，程序员用一个分号来结束语句，但预处理器把它作为size定义的一部分。对size的赋值语句展开为：

```
size = 10; -2;;
```

结尾的两个分号不会影响什么，但中间的一个分号却是一个“杀手”。这行代码告诉C++做两件事：1)把10赋给size，2)计算值-2后把它扔掉（由此会有“null effect”的警告信息）。修改程序，删去分号，就可以了。

问题10.4：预处理器的输出为

```
void exit();

main() {
    int value;

    value = 1;
    if (value < 0)
        cout << "Fatal Error: Abort\n"; exit(8);

    cout << "We did not die\n";
    return (0);
}
```

问题出在if后面的两个语句中。一般地，它们应该放在两行。如果缩进正确的话，应该是下面的形式：

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

main() {
    int value; // A random value for testing

    value = 1;
    if (value < 0)
        cout << "Fatal Error: Abort\n";

    exit(8);

    cout << "We did not die\n";
    return (0);
}
```

从这段代码中容易看出，我们为什么总是退出。事实上，if语句后面的两个语句隐藏在

单一的预处理器宏中。纠正这个错误的方法是在多语句宏的所有语句外加一对花括号。

```
#define DIE \  
    {cout << "Fatal Error: Abort\n"; exit(8);}
```

问题10.5: 答案是，每次循环后counter增2。这是因为宏调用：

```
SQR(++counter)
```

扩展为：

```
((++counter) * (++counter))
```

问题10.6: 参数宏和不带参数的宏的唯一区别是，参数宏的宏名字后立即跟有一个括号。本例中，**RECIPROCAL**的定义后是一个空格，所以它不是一个参数宏。它只是一个简单的文本替换宏，把**RECIPROCAL**替换为：

```
(number) (1.0 / number)
```

删去**RECIPROCAL**和(number)中间的空格，问题自会解决。

本章内容:

- 位运算符
- 与运算符
- 按位或
- 按位异或
- 非运算符
- 左移与右移运算符
- 设置、清除和检测位
- 位图图形
- 编程练习
- 答疑

第11章 位运算

本章讨论面向位的运算符。一个位是信息的最小单位。一般地，它由1和0来表示。（其它表示方法有开/关、真/假和是/否。）位操作用来在最底层控制机器，它们允许程序员接触到机器的内部。许多高级程序从来不需要位操作。如设备驱动程序或是像素级的图形程序这样的低级代码，才需要位操作。如果你只想写高级程序，这一章可以跳过。

八个位放在一起组成一个字节，表示为C++中的数据类型char。一个字节可以包含如下的位：01100100。

这个字节也可以写为十六进制数0x64。（C++用前缀“0x”表示十六进制数（基为16）。）用十六进制表示二进制数时很方便，因为每个十六进制数代表4个二进制位。表11-1列出十六进制（hex）与二进制的对应表：

表 11.1 十六进制与二进制

十六进制	二进制	十六进制	二进制
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

位运算符

位运算符或叫按位运算符可以让程序员对单个的位进行操作。例如，一个短整数可存放16位（在大多数机器上）。位运算符把这些位都看成是毫不相关的。相反，加运算符把这16位看成是一个由16位组成的单一的数。

位运算符允许你对位进行设置、清除、检测操作，还可以执行其它的运算。位运算符列在表11.2中。这些运算符可以作用于任何的整型或是字符型数据。

表11.2 位运算符

运算符	含 义
&	按位与
	按位或
^	按位异或
~	非
<<	左移
>>	右移

与运算符（&）

与运算符比较两个位。如果它们都为1，则结果为1。与运算符的定义如表11.3所示。

表11.3 与运算符

Bit1	Bit2	Bit1 & Bit2
0	0	0
0	1	0
1	0	0
1	1	1

当两个8位的变量（字符变量）进行与操作时，与运算符独立地对每一位进行操作。下面的程序描述了这个操作：

```
int    c1, c2;

c1 = 0x45;
c2 = 0x71;
```



```
cout << "Result of " << Hex << c1 << " & " << c2 << " = " <<
      (c1 & c2) << dec << '\n';
```

该程序的输出为:

Result of 45 & 71 = 41

运算过程是:

c1 = 0x45	二进制	01000101
& c2 = 0x71	二进制	01110001
= 0x41	二进制	01000101

按位与 (&) 类似于逻辑与 (&&)。在逻辑与中, 如果两个操作数都为真 (非0), 则结果为真 (1)。在按位与 (&) 中, 如果两个操作数中对应的位都为真 (1s), 则结果的对应位即为真 (1s)。所以说按位与 (&) 独立地对位进行操作, 而逻辑与 (&&) 把操作数看成一个整体。但是, &与&&毕竟不是同一个运算符, 示例11.1中说明这个问题:

示例11.1 *and/and.cc*

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

```
main()
```

```
{
```

```
    int i1, i2; // Two random integers
```

```
    i1 = 4;
```

```
    i2 = 2;      // Set values
```

```
    // Nice way of writing the conditional
```

```
    if ((i1 != 0) && (i2 != 0))
```

```
        cout << "Both are not zero #1\n";
```

```
    // Shorthand way of doing the same thing
```

```
    // Correct C++ code, but rotten style
```

```
    if (i1 && i2)
```

```
        cout << "Both are not zero #2\n";
```

```
    // Incorrect use of bitwise AND resulting in an error
```

```
    if (i1 & i2)
```

```
        cout << "Both are not zero #3\n";
```

```
    return (0);
```

```
}
```


问题：为什么进行第三种检测时程序不输出Both are not zero #3?

答案：运算符&是按位与。按位与的结果为0。

i1=4	00000100
i2=2	00000010
&	00000000

因为按位与的结果为0，所以条件不满足。如果程序员使用第一种形式：

```
if ((i1 != 0) && (i2 != 0))
```

同时把&&错误地写为&：

```
if ((i1 != 0) & (i2 != 0))
```

则程序也会正确地执行。

```
(i1 != 0) is true(result = 1)
```

```
(i2 != 0) is true(result = 1)
```

1与1进行按位与运算，结果还是1，所以表达式为真。

注意

我发现这个程序中说明的这个小问题后不久，我对办公室的同事说：“现在理解了&和&&之间的不同了”，他也明白我的意思。我一直不知该如何理解语言，我说了这样一句话，而且竟有人明白，这确实令我吃惊。

你可以使用按位与运算符来判定一个数是奇数还是偶数。基数为2时，所有偶数的最后一位数字是0，所有奇数的最后一位数字是1。下列函数使用按位与运算截取最后一位数字。如果它是0（偶数），函数返回真。

```
inline int even(const int value)
{
    return ((value & 1) == 0);
}
```

按位或 (|)

相容析取运算符（也称为或运算）比较两个操作数。如果其中有一个位是1，则结果也为1。表11.4a列出或运算的真值表。

表11.4a 或运算符

Bit1	Bit2	Bit1 Bit2
0	0	0
0	1	1
1	0	1
1	1	1

对一个字节执行或运算的示例为:

i1=0x47	01000111
i2=0x53	01010011
57	01010111

按位异或 (^)

当两个操作数中有一个且只有一个为1时, 不可兼析取运算 (也称为异或) 的结果也为1。异或运算的真值表见表11.4b。

表11.4b 异或运算

Bit1	Bit2	Bit1 ^ Bit2
0	0	0
0	1	1
1	0	1
1	1	0

对一个字节执行异或运算的示例为:

i1=0x47	01000111
i2=0x53	01010011
^ 14	00010100

非运算符 (~)

非运算符（也称为取反运算符）是一元运算符，它返回操作数的反，如表11.5所列。

表11.5 非运算符

Bit	~Bit
0	1
1	0

对一个字节执行非运算的示例为：

c=	0x45	01000101
~c=	0xBA	10111010

左移和右移运算符 (<<, >>)

左移运算符把数据向左移动若干位。移出的所有位都丢失了，右侧新增加的位为0。右移运算符与此类似，只是方向不同。例如：

	c=0x1C	00011100
c << 1	c=0x38	00111000
c >> 2	c=0x07	00000111

向左移动1位 ($x \ll 1$) 等同于被2乘 ($x * 2$)。向左移动2位 ($x \ll 2$) 等同于被4乘 ($x * 4$, 或 $x * 2^2$)。由此可以得出一个公式。向左移动n位等同于乘上 2^n 。为什么用移位取代乘法？因为移位操作比乘法运算更快，所以

```
i = j << 3;    // Multiply j by 8 (2**3)
```

比：

```
i = j * 8;
```

执行得更快。或者说，如果编译器没能聪明地把乘法转换为左移的话，前一种方式执行得更快。

有许多聪明的程序员用这种技巧加快程序执行的速度，但却失掉了简明性。请不要这

样做。编译器会自动地完成这样的加速。也就是说，用左移运算符不会给你带来任何好处，反而会使程序不清楚。

左移运算是乘法；那么，右移运算就是除法了。所以：

```
q = i >> 2;
```

就是：

```
q = i / 4;
```

而且，这种技巧也不要用在程序中。

详谈右移运算

右移运算特别复杂。当右移一个变量时，C++需要填充其左边的空格。对于有符号的变量来说，C++用其符号位来填充。对于无符号变量来说，C++用0来填充。表11.6说明一些典型的右移操作。

表11.6 右移示例

	有符号字符	有符号字符	无符号字符
表达式	9 >> 2	-8 >> 2	248 >> 2
二进制值 >> 2	0000 1010 ₂ >>2	1111 1000 ₂ >>2	1111 1000 ₂ >>2
结果	??000010 ₂	??11 1110 ₂ >>2	??11 1110 ₂ >>2
填充	符号位 (0)	符号位 (1)	0
最后结果 (二进制)	0000 0010 ₂	1111 1110 ₂	0011 1110 ₂
最后结果 (短整型)	2	-2	62

设置、清除和检测位

一个字符包含8个位。每个位都可以看作是一个独立的标志。位操作可以用来截取单字节中的单个位。例如，假定你正写一个底层通信程序。你准备在一个8K的缓冲区中存储字符，留待稍后再用。对每个字符，你还将存储一组状态标志。标志列在表11.7中。

表11.7 通信程序状态值

名 字	描 述
ERROR	如果设置了任何错误，则为1
FRAMING_ERROR	本字符发生结构错误
PARITY_ERROR	字符奇偶错误

表11.7 通信程序状态值 (续)

名 字	描 述
CARRIER_LOST	传送信号消失
CHANNEL_DOWN	通信设备的电源信号消失

你可以在每个字符变量中存储每个标志。这表明, 对每个缓存字符, 都需要5个字节的
状态存储空间。对于一个大的缓冲区来说, 这个数目太大了。我们可以把每个标志位都放
入一个8位的状态字符中, 这样所用空间只需原来的1/5。

你可以按表11.8给标志赋值。

表11.8 位赋值

位	名 字
0	ERROR
1	FRAMING_ERROR
2	PARITY_ERROR
3	CARRIER_LOST
4	CHANNEL_DOWN

按惯例, 位的顺序是76543210。表11.9中列出每个位对应的常量。

表11.9 位值

位	二进制值	十六进制常量
7	10000000	0x80
6	01000000	0x40
5	00100000	0x20
4	00010000	0x10
3	00001000	0x08
2	00000100	0x04
1	00000010	0x02
0	00000001	0x01

可以如下定义:

```
// True if any error is set
const int ERROR = 0x01;
```

```

// A framing error occurred for this character
const int FRAMING_ERROR = 0x02;

// Character had the wrong parity
const int PARITY_ERROR = 0x04;

// The carrier signal went down
const int CARRIER_LOST = 0x08;

// Power was lost on the communication device
const int CHANNEL_DOWN = 0x10;

```

位的这种定义方法不太清晰。你能不看表说出哪个位代表常量0x10吗？表11.10显示如何使用左移运算符（<<）来定义位。

表11.10 左移运算符和位的定义

C++ Representation	Base 2 Equivalent	Result (Base 2)	Bit Number
1 << 0	00000001 ₂ << 0	00000001 ₂	Bit 0
1 << 1	00000001 ₂ << 1	00000010 ₂	Bit 1
1 << 2	00000001 ₂ << 2	00000100 ₂	Bit 2
1 << 3	00000001 ₂ << 3	00001000 ₂	Bit 3
1 << 4	00000001 ₂ << 4	00010000 ₂	Bit 4
1 << 5	00000001 ₂ << 5	00100000 ₂	Bit 5
1 << 6	00000001 ₂ << 6	01000000 ₂	Bit 6
1 << 7	00000001 ₂ << 7	10000000 ₂	Bit 7

虽然很难说出哪个位代表0x10，但很容易就能说出哪个位表示1 << 4。

可以如下定义标志：

```

// True if any error is set
const int ERROR = (1 << 0);

// A framing error occurred for this character
const int FRAMING_ERROR = (1 << 1);

// Character had the wrong parity
const int PARITY_ERROR = (1 << 2);

// The carrier signal went down
const int CARRIER_LOST = (1 << 3);

```



```
// Power was lost on the communication device
const int CHANNEL_DOWN = (1 << 4);
```

现在你已经定义了位，可以操纵它们了。若要置一个位的值，要使用|运算符。例如：

```
char flags = 0; // Start all flags at 0
flags |= CHANNEL_DOWN; // Channel just died
```

要检测一个位，需使用&运算符把位“屏蔽出来”。

```
if ((flags & ERROR) != 0)
    cerr << "Error flag is set\n";
else
    cerr << "No error detected\n";
```

清除一个位有点困难。假定你想清除位PARITY_ERROR。在二进制中，这个位是00000100。你要建立一个屏蔽，它能屏蔽掉所有的位，但只保留下你想清除的位(11111011)。这可以用非运算符(~)实现。然后，屏蔽与数进行与运算，以清除那一位。

PARITY_ERROR	00000100
~PARITY_ERROR	11111011
flags	00000101
flags & ~PARITY_ERROR	00000001

在C++中，表示为：

```
flags &= ~PARITY_ERROR; // Who cares about parity
```

问题11.1： 在下面的程序中，HIGH_SPEED标志起作用，但DIRECT_CONNECT标志不起作用。为什么？

```
#include <iostream.h>

const int HIGH_SPEED = (1<<7);    /* modem is running fast */
                                   // we are using a hardwired connection
const int DIRECT_CONNECT = (1<<8);

char flags = 0;                    // start with nothing

main()
{
    flags |= HIGH_SPEED;           // we are running fast
    flags |= DIRECT_CONNECT;       // because we are wired together

    if ((flags & HIGH_SPEED) != 0)
        cout << "High speed set\n";
```

```

    if ((flags & DIRECT_CONNECT) != 0)
        cout << "Direct connect set\n";
    return (0);
}

```

位图图形

现在，越来越多的计算机可以处理图形了。PC中，已有了像EGA和VGA卡这样的图形设备。UNIX中，有了X窗口系统。

在位图图形中，屏幕上的每个像素都由内存中的1个位来表示。例如，图11.1显示的是一个14*14的位图在屏幕上的样子以及放大后的示意图。放大后，我们可以看清它的各个位。

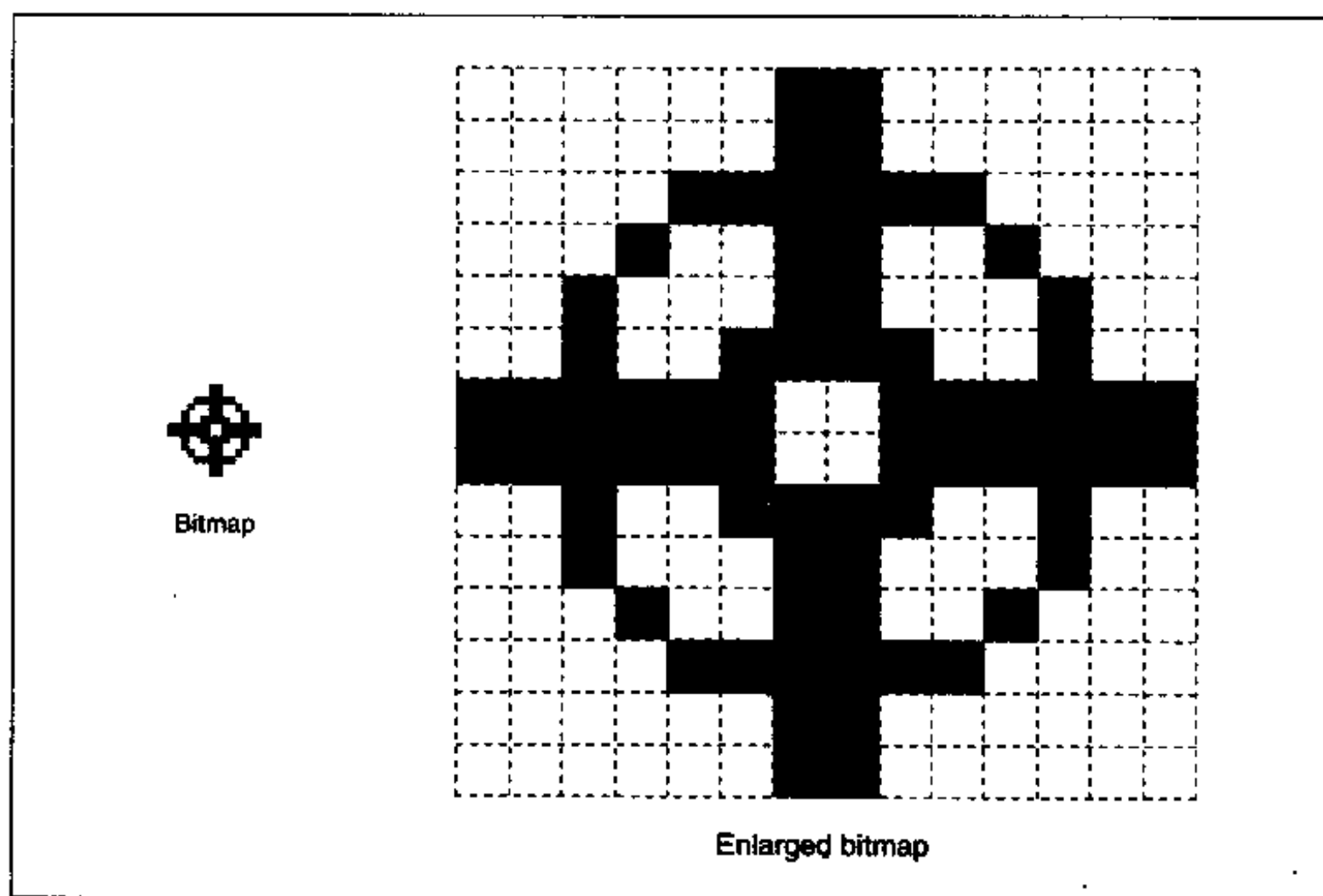


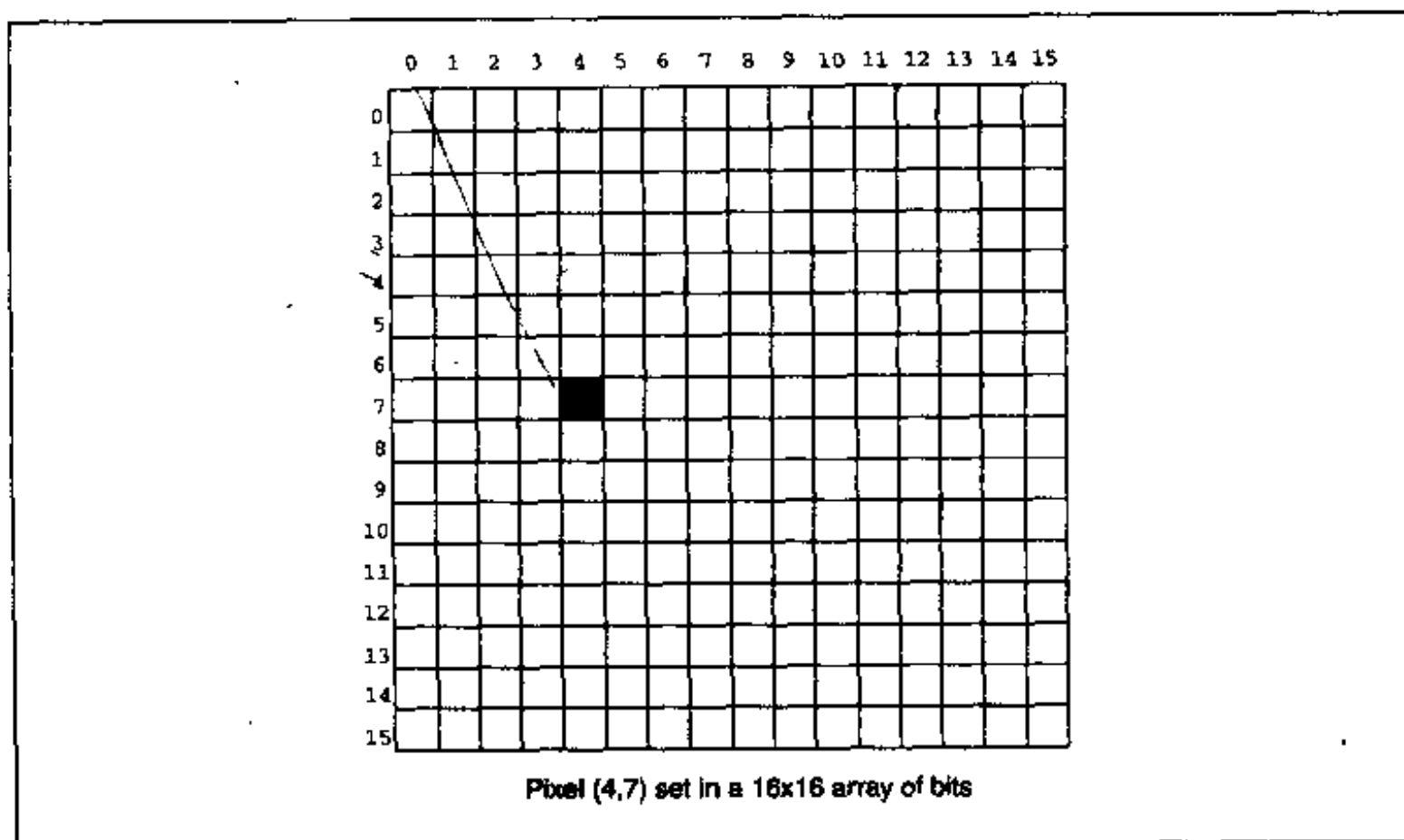
图 11.1 实际的尺寸和放大后的位图

假定我们有一个小型图形设备——一个16*16像素的单显。我们想设置(4,7)那个位置的位。这个设备的位图是一个位矩阵，如图11.2中显示的那样。

这里我们遇到了一个问题。在C++中没有用于位矩阵的数据类型，最接近的是一个字节数组。所以，16*16的位矩阵就变成了2*16的字节数组，如图11.3所示。

要设置(4,7)位置的像素，我们需要设置(0,7)字节的第四个位。要设置这个位，使用语句：`bit_array[0][7] |= (0x80 >> (4));`常量0x80是最左位。

我们可以建立一个宏，它设置(x,y)位置的位(像素)，从而使这个过程一般化。这里需要计算两个值：字节的坐标和字节内位的位置。位的位置是(x,y)。字节由8个位组成，



图·11.2 位矩阵

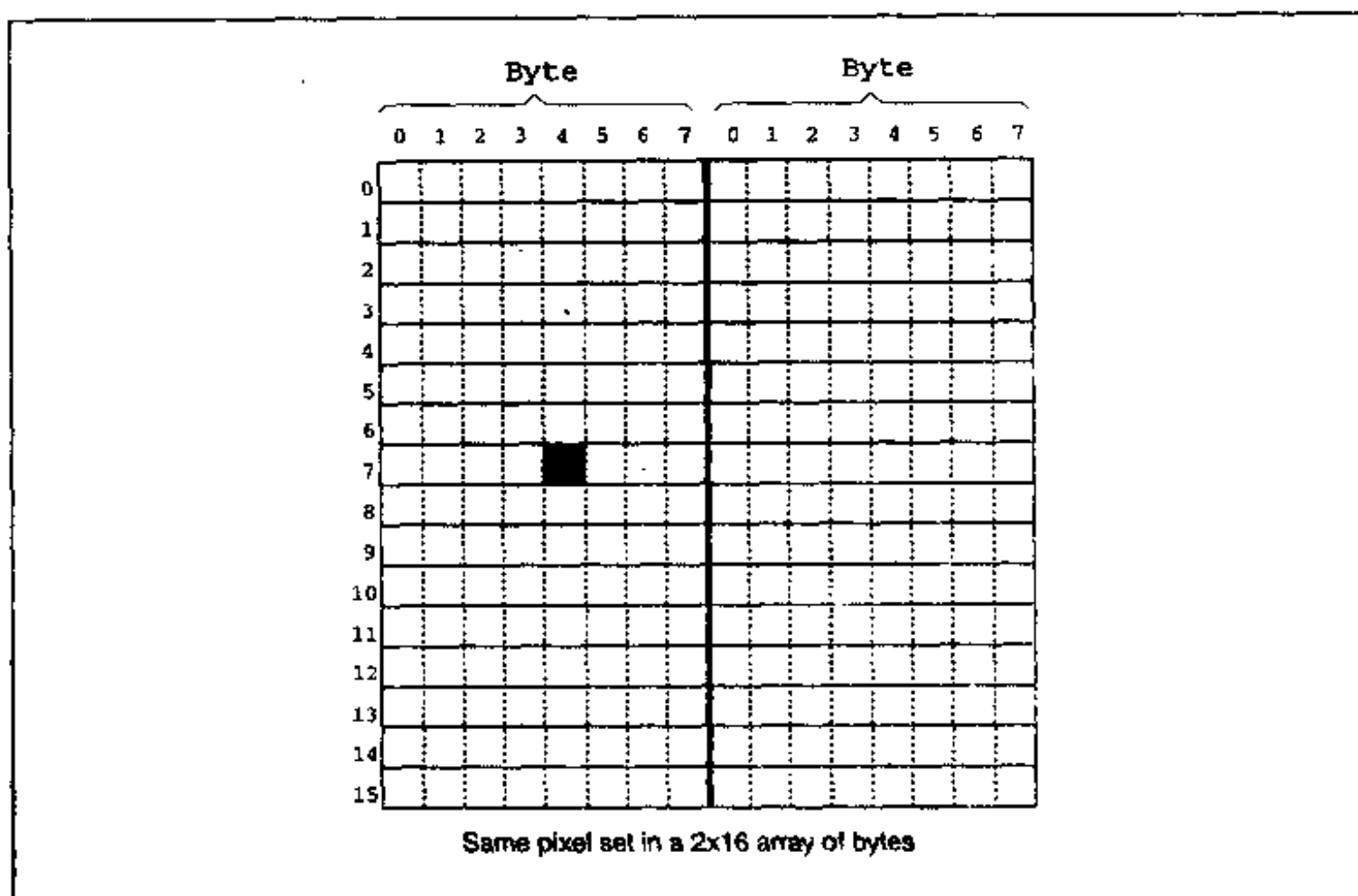


图 11.3 字节数组

所以，字节地址就是 $(x/8, y)$ 。字节内的位可不这样简单。我们想建立一个由要设置的单个位组成的屏蔽。对于最左位来说，这个屏蔽应该是 $1000\ 0000_2$ ，或者表示为 $0x80$ 。此时， $(x\%8) == 0$ 。下一位是 $0100\ 0000_2$ ，或表示为 $(0x80 \gg 1)$ ，此时， $(x\%8) == 1$ 。所以建立屏蔽时使用表达式 $(0x80 \gg (x\%8))$ 。

现在，我们已有了字节位置和位屏蔽，剩下的就是要设置位了。下面的函数设置位图数组`graphics`中的给定位。

```
void inline set_bit(const int x,const int y)
{
    graphics[(x)/8][y] |= (0x80 >> ((x)%8))
}
```

示例11.2画出图形数组的对角线，并在终端上输出该数组。

示例11.2 `graph/graph.cc`

```
#include <iostream.h>

const int X_SIZE = 40; // Size of array in the X direction
const int Y_SIZE = 60; // Size of array in the Y direction
/*
 * We use X_SIZE/8 since we pack 8 bits per byte
 */
char graphics[X_SIZE / 8][Y_SIZE]; // The graphics data

/*****
 * set_bit -- set a bit in the graphics array
 *
 * Parameters
 *      x,y -- location of the bit
 *****/
inline void set_bit(const int x,const int y)
{
    graphics[(x)/8][y] |= (0x80 >> ((x)%8));
}

main()
{
    int loc; // Current location we are setting
    void print_graphics(void); // Print the data

    for (loc = 0; loc < X_SIZE; ++loc)
        set_bit(loc, loc);

    print_graphics();
    return (0);
}

/*****
 * print_graphics -- print the graphics bit array
 *
 *      as a set of X and .'s
 *****/
```

```

void print_graphics(void)
{
    int x;      // Current x byte
    int y;      // Current y location
    int bit;    // Bit we are testing in the current byte

    for (y = 0; y < Y_SIZE; ++y) {

        // Loop for each byte in the array
        for (x = 0; x < X_SIZE / 8; ++x) {
            // Handle each bit
            for (bit = 0x80; bit > 0; bit = (bit >> 1)) {
                if ((graphics[x][y] & bit) != 0)
                    cout << 'X';
                else
                    cout << '.';
            }
        }
        cout << '\n';
    }
}

```

程序中定义了一个位图数组：

```
char graphics[X_SIZE / 8][Y_SIZE]; // The graphics data
```

因为我们共有X_SIZE个位，转为字节就是X_SIZE/8，所以要用到常量X_SIZE/8。

主for循环：

```

for (loc = 0; loc < X_SIZE; ++loc)
    set_bit(loc, loc);

```

画出一条斜穿图形数组的对角线。

因为我们没有位图设备，所以我们用子程序print_graphics来仿真它。循环：

```

for (y = 0; y < Y_SIZE; ++y) {
    ...

```

可以输出每一行。循环：

```

for (x = 0; x < X_SIZE / 8; ++x) {
    ...

```

则输出行中的每个字节。在每个字节中有8位，这由下面的循环来处理：

```
for (bit = 0x80; bit > 0; bit = (bit >> 1))
```

它使用了一个独特的循环计数器。该循环让变量bit从7位（最左位）开始。对循环的每一次重复，位向右移一位，即 $\text{bit} = (\text{bit} \gg 1)$ 。当移完了位后，循环退出。循环计数器走了一圈。

最后，在循环的内部有如下的代码：

二进制	十六进制
0000 0000 1000 0000	0x80
0000 0000 0100 0000	0x40
0000 0000 0010 0000	0x20
0000 0000 0001 0000	0x10
0000 0000 0000 1000	0x08
0000 0000 0000 0100	0x04
0000 0000 0000 0010	0x02
0000 0000 0000 0001	0x01

```
if ((graphics[x][y] & bit) != 0)
    cout << "X";
else
    cout << ".";
```

它测试一个单独的位，如果该位已被设置了，则输出“X”；如果没被设置，则输出“.”。

问题11.2：在下面的程序中，HIGH_SPEED标志起作用，但DIRECT_CONNECT标志不起作用。为什么？在下面的程序中，第一个循环工作，但第二个不工作。为什么？

示例11.3 loop/loop.cc

```
#include <iostream.h>
main()
{
    short int i;

    // works
    for (i = 0x80; i != 0; i = (i >> 1)) {
        cout << "i is " << hex << i << dec << '\n';
    }

    signed char ch;

    // Fails
    for (ch = 0x80; ch != 0; ch = (ch .. 1)) {
        cout << "ch is " << hex << int(ch) << dec << '\n';
    }
}
```

```

    }
    return (0);
}

```

编程练习

练习11.1: 编写一组内联函数: `clear_bit`和`test_bit`, 与示例11.2中定义的`set_bit`函数一起工作。编写一个主程序来测试这些宏。

练习11.2: 编写一个程序, 画一个10*10的位图正方形。

练习11.3: 修改示例11.2, 在黑背景下画一条白对角线。

• **练习11.4:** 编写一个程序, 统计一个整数中设置位的个数。例如, 数5 (十进制) 表示为00000000000000101 (二进制), 它有两个设置位。

练习11.5: 编写一个程序, 它把一个32位的整数 (`long int`), 划分为四个4位的数。(小心处理符号位。)

练习11.6: 编写一个程序, 它把一个数中的所有设置位移到最左边。例如, 01010110 (二进制) 变为11110000 (二进制)。

答疑

问题11.1: `DIRECT_CONNECT`由表达式 $(1 \ll 8)$ 定义为第8位, 但是, 字符变量的8个位的序号是76543210, 没有第8位。解决的办法是把`flags`说明为16位的短整型。

问题11.2: 问题出在`ch`是一个字符 (8位)。值0x80用8位表示就是1000 0000₂。第一位是符号位, 它为1。当右移1位时, 用符号位来填充, 所以, 1000 0000₂ >> 1得到1100 0000₂。虽然变量`i`有符号, 但也得出了结果, 因为它有16位。所以, 用16位表示时, 0x80就是0000 0000 1000 0000₂。注意, 设置的位不是符号位。

解决的办法是把`ch`说明为一个无符号的变量。

第三部分

高级类型

和类

本章内容:

- 结构
- 联合
- typedef
- enum类型
- 位域或紧缩结构
- 结构数组
- 编程练习

第12章

高级类型

C++提供了一组丰富的数据类型。通过使用结构、联合、enum和类等类型，程序员可以用新的类型来扩展语言。

结 构

假定你正为仓库写一个盘货程序。仓库里堆满了箱子，箱子内的东西也分类整理好了。箱子内所有的类都作了标记，所以你不必担心箱子或是其中的东西弄混了。

对每个箱子你需要了解：

- 内部各部分的名字（30个字符长）。
- 箱内数量（整数）。
- 价格（以分为单位）。

在前一章，我们已经用数组存放过一组数据类型相似的数据；但在这个例子中，你的数据是不同的类型：两个整数和一个串。

我们不使用数组，而使用一种新的数据类型，名叫结构(structure)。在数组中，所有的元素都是同一类型的，并且它们有次序。在结构中，每个元素，或叫域(field)，都有名字，并有自己的数据类型。

结构定义的一般形式为：

```
struct 结构名 {  
    域类型 域名 // 注释  
    域类型 域名 // 注释  
    ....  
} 变量名;
```

例如，你想定义一个装打印机电缆的箱子。结构定义如下：


```

main()
{
    typedef int group[10];    // Create a new type "group"

    group totals;            // Use the new type for a variable

    // Initialize each element of total
    for (i = 0; i < 10; ++i)
        totals[i] = 0;
}

```

enum类型

枚举 (*enum*) 数据类型用来定义只可取有限个值的变量。这些值都被称为名字 (标签)。在内部, 编译器为每个标签赋一个整数值, 例如一个星期中的各天。你可以直接用 `#define` 来建立一周内各天的值 (`day_of_the_week`), 例如:

```

typedef int day_of_the_week;    // Define the type for days of the week

const int SUNDAY    = 0;
const int MONDAY    = 1;
const int TUESDAY   = 2;
const int WEDNESDAY = 3;
const int THURSDAY  = 4;
const int FRIDAY    = 5;
const int SATURDAY  = 6;

/* Now to use it */
day_of_the_week today = TUESDAY;

```

这种方法不太方便。更好的方法是使用枚举类型:

```

enum day_of_the_week {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY};

/* Now use it */
enum day_of_the_week today = TUESDAY;

```

枚举语句的一般形式为:

enum 枚举名 {标记1, 标记2,...} 变量名

和结构一样, 枚举名或是变量名可以省略。标记可以是任意有效的C++标识符; 不过, 标记通常要大写。

使用enum类型的好处是，C++可以约束enum说明中所列变量中存储的值。编译器检查下面的代码时会发现一个错误：

```
today = 5; // 5 is not a day_of_the_week
```

enum类型也有令人失望的地方，那就是，enum变量不能用作数组的下标。下面的代码是不对的：

```
enum day_of_the_week today = TUESDAY;

// Define string versions of our days of the week
char day_names[7][] = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};

...

/*
 * The following line generates a warning
 * because today is not an integer
 */
cout << "Today is " << day_names[today] << '\n';
```

为了解决这个问题，你需要告诉C++把today看作是一个整数。用cast或是typecast运算符就可以了。表达式int(today)告诉C++：“我知道today不是一个整数，但要把它看作是一个整数。”使用下面的语句来解决刚才的问题：

```
cout << "Today is " << day_names[int(today)] << '\n';
```

在表达式中，cast也非常有用，它可以确保变量有正确的类型。一般地，你可以用下面的表达式修改几乎所有表达式的类型：

类型(表达式)

当混用整数和浮点数时，这非常有用。

```
int won, lost; // # games won/lost so far
float ratio; // Win/loss ratio

won = 5;
lost = 3;
```



```

struct item {
    unsigned int list:1;    // True if item is in the list
    unsigned int seen:1;    // True if this item has been seen
    unsigned int number:14; // Item number
};

```

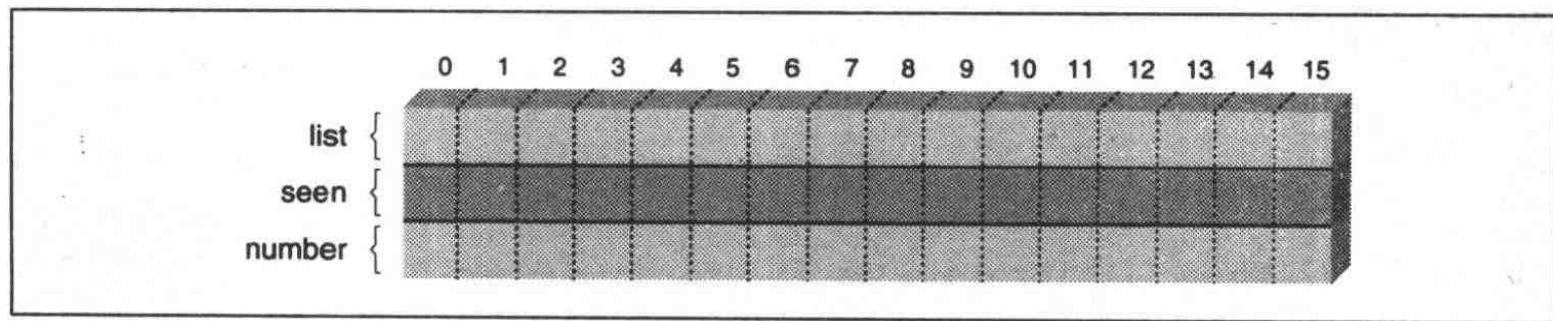


图 12.3 非紧缩结构

如上的定义告诉编译器用1位来表示list，用1位来表示seen，用14位来表示number。用这样的方法，可以把数据压缩到2个字节内，见图12.4。

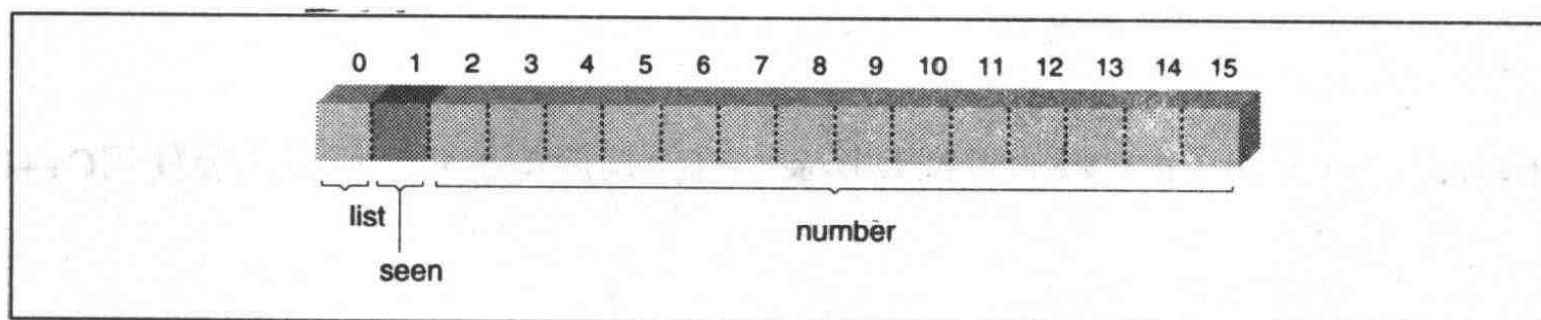


图 12.4 紧缩结构

使用紧缩结构的时候要小心。相对来说，从位域中取数据的机器代码要多一些，而且要慢一些。除非存储非常紧张，否则不要使用紧缩结构。

在第10章“C++预处理器”中，需要存储字符数据和8000个字符对应的5个状态标志。在这个例子中，对每个标志都使用了不同的字节，所以耗费了许多的存储空间（每个读入的字符占5个字节）。你可以使用按位运算，把5个字节压缩到1个字节中。或者，可以用一个紧缩结构完成同样的功能：

```

struct char_and_status {
    char character;    // Character from device
    int error:1;       // True if any error is set
    int framing_error:1; // A framing error occurred
    int parity_error:1; // Character had the wrong parity
    int carrier_lost:1; // The carrier signal went down
    int channel_down:1; // Power was lost on the channel
};

```

使用紧缩结构来表示标志，比使用按位运算更清楚，也不易出错。但是，按位运算有更大的灵活性。你可以使用对你来说最清楚最简单的方法。

结构数组

结构和数组可以组合在一起。假如你想记录一名运动员四圈赛跑中每一圈跑的时间。定义一个结构存放时间：

```
struct time {
    int hour;    // Hour (24-hour clock)
    int minute;  // 0-59
    int second;  // 0-59
};

#define MAX_LAPS 4 /* We will have only 4 laps */

/* The time of day for each lap */
struct time lap[MAX_LAPS];
```

语句：

```
struct time lap[MAX_LAPS];
```

把lap定义为一个有4个元素的数组。每个元素包含一个time结构。

使用方法如下：

```
/*
 * Runner just past the timing point
 */

lap[count].hour = hour;
lap[count].minute = minute;
lap[count].second = second;
++count;
```

这个数组可以在运行时进行初始化。结构数组的初始化类似于多维数组的初始化。

```
struct time start_stop[2] = {
    (10, 0, 0),
    (12, 0, 0)
};
```

假如你想写一个处理邮件表的程序。邮件标签高5行宽60个字符。用一个结构来存放名字和地址。邮件表打印输出时将按名排序，而对于真正的邮件，则以邮编顺序来排列。邮件表的结构如下：

```
struct mailing {
    char name[60];    // Last name, first name
```

```

    char address1[60]; // Two lines of street address
    char address2[60];
    char city[40];     // Name of the city
    char state[2];     // Two-character abbreviation
    long int zip;      // Numeric zip code
};

```

现在可以说明一个存放邮件表的数组了：

```

/* Our mailing list */
struct mailing list[MAX_ENTRIES];

```

编程练习

练习12.1：编写一个程序，把若干个名字和地址分类，产生一组邮件标签。

练习12.2：设计一个存放时间和日期的结构。编写一个函数，计算两个时间相差的分钟数。

练习12.3：设计一个飞机票预订数据结构，其中存储下列数据：

航班号

起始地代码（3个字符）

目的地代码（3个字符）

启程时间

到达时间

编写一个程序，列出由用户指定的起始和目的地两机场间所有可能的飞行计划。

表13.1 栈操作

操作	操作后的栈
Push (4)	4
Push (5)	5 4
Push (8)	8 5 4
Pop (returns 8)	5 4
Pop (returns 5)	4
Push (9)	9 4
Pop (returns 9)	4
Pop (returns 4)	<empty>

设计一个栈

栈的设计从设计数据结构着手。这个结构要有一个地方存放数据（名叫data），还需要存放当前栈中的项数（名叫count）：

```
const int STACK_SIZE = 100;    // Maximum size of a stack

// The stack itself
struct stack {
    int count;                  // Number of items in the stack
    int data[STACK_SIZE];      // The items themselves
};
```

下一步，要建立处理压栈和出栈操作的程序。压栈函数在栈中存放一项，并累加项的计数。

```
inline void stack_push(struct stack &the_stack, const int item)
{
    the_stack.data[the_stack.count] = item;
    ++the_stack.count;
}
```

注意：这个程序不检查栈是否上溢，或其它的错误条件。在后面第14章“类的进一步讨论”中，你将看到如何用这个简单的栈去建立一个更安全更复杂的栈。

出栈仅仅是把栈顶元素删去，并使项的计数减1。

```
inline int stack_pop(struct stack &the_stack)
{
    // Stack goes down by one
    --the_stack.count;
```




```

    // Then we return the top value
    return (the_stack.data[the_stack.count]);
};

```

这里忽略了一件事：初始化栈。在使用栈之前，必须先设置它。再多写两行代码，得到`stack_init`函数。

```

inline void stack_init(struct stack &the_stack)
{
    the_stack.count = 0;    // Zero the stack
}

```

注意，你不必对栈中的`data`域清0，因为`data`的元素要由`push`操作再写一遍。

现在任务完成了。要想实际使用它，需要用`struct`语句先说明它。然后再确保对它进行初始化，之后就可以进行压栈出栈操作了。

```

struct stack a_stack;    // Declare the stack

stack_init(a_stack);    // Initialize the stack
// Stack is ready for use

```

示例13.1使用结构的方法，完整地实现了栈的操作，另外，它还带有一个短小的测试程序。

```

示例13.1 stack_s/stack_s.cc
/*****
 * Stack
 *      A set of routines to implement a simple integer
 *      stack.
 *
 * Procedures
 *      stack_init -- initialize the stack
 *      stack_push -- put an item on the stack
 *      stack_pop -- remove an item from the stack
 *****/
#include <stdlib.h>
#include <iostream.h>

const int STACK_SIZE = 100;    // Maximum size of a stack

// The stack itself
struct stack {
    int count;                // Number of items in the stack
    int data[STACK_SIZE];    // The items themselves
};

/*****

```

```

* stack_init -- initialize the stack
*
* Parameters
*   the_stack -- stack to initialize
*****/
inline void stack_init(struct stack &the_stack)
{
    the_stack.count = 0;        // Zero the stack
}
/*****
* stack_push -- push an item on the stack
*
* Warning: We do not check for overflow
*
* Parameters
*   the_stack -- stack to use for storing the item
*   item -- item to put in the stack
*****/
inline void stack_push(struct stack &the_stack,
                       const int item)
{
    the_stack.data[the_stack.count] = item;
    ++the_stack.count;
}
/*****
* stack_pop -- get an item off the stack
*
* Warning: We do not check for stack underflow
*
* Parameters
*   the_stack -- stack to get the item from
*
* Returns
*   the top item from the stack
*****/
inline int stack_pop(struct stack &the_stack)
{
    // Stack goes down by one
    --the_stack.count;

    // Then we return the top value
    return (the_stack.data[the_stack.count]);
}

// A short routine to test the stack

```


访问成员变量。即你不用写`the_stack.count`，只需写`count`即可。

用类似的方法实现函数`push`和`pop`。

```
inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}

inline int stack::pop(void)
{
    // Stack goes down by one
    -- count;

    // Then we return the top value
    return (data[count]);
}
```

栈类的实现已经完成了。你要做的就是使用它。

类的使用

类的使用很像结构的使用。类变量的说明与结构的说明也是相同的，只是用`class`取代`struct`。

```
class stack a_stack;    // Stack we want to use
```

字`class`并不是必需的，它常常被省略。

```
stack a_stack;    // Stack we want to use
```

访问结构的域时使用的是一个点，例如：

```
structure.fileld = 5;
```

类成员的访问也是类似的，只是类的成员有数据成员和成员函数。而且，你只能访问公有成员。

为了访问栈类的`init`成员函数，可以使用下面的语句：

```
a_stack.init();
```

可以用类似的方法访问`push`和`pop`成员函数：

```
a_stack.push(1);
result = a_stack.pop();
```

示例13.2是用类实现的栈。

示例13.2 `stack_c/stack_c.cc`

```

/*****
 * Stack
 *      A file implementing a simple stack class
 *****/
#include <stdlib.h>
#include <iostream.h>

const int STACK_SIZE = 100;      // Maximum size of a stack

/*****
 * Stack class
 *
 * Member functions
 *   init -- initialize the stack
 *   push -- put an item on the stack
 *   pop  -- remove an item from the stack
 *****/
// The stack itself
class stack {
    private:
        int count;           // Number of items in the stack
        int data[STACK_SIZE]; // The items themselves
    public:
        // Initialize the stack
        void init(void);

        // Push an item on the stack
        void push(const int item);

        // Pop an item from the stack
        int pop(void);
};

/*****
 * stack::init -- initialize the stack
 *****/
inline void stack::init(void)
{
    count = 0; // Zero the stack
}

/*****
 * stack::push -- push an item on the stack
 *
 * Warning: We do not check for overflow
 *****/

```

```

*
* Parameters
*      item -- item to put in the stack
*
*****/
inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}
/*****
* stack::pop -- get an item off the stack
*
* Warning: We do not check for stack underflow
*
* Returns
*      the top item from the stack
*
*****/
inline int stack::pop(void)
{
    // Stack goes down by one
    --count;

    // Then we return the top value
    return (data[count]);
}

// A short routine to test the stack
main()
{
    stack a_stack;      // Stack we want to use

    a_stack.init();

    // Push three values on the stack
    a_stack.push(1);
    a_stack.push(2);
    a_stack.push(3);

    // Pop the items from the stack
    cout << "Expect a 3 ->" << a_stack.pop() << '\n';
    cout << "Expect a 2 ->" << a_stack.pop() << '\n';
    cout << "Expect a 1 ->" << a_stack.pop() << '\n';

    return (0);
}

```

介绍构造函数和析构函数

栈类有一个不方便的地方。程序员在使用栈之前必须先调用init成员函数。但是，程序员有可能忘记了，而且总会有人忘记对栈进行初始化。如果C++自动进行栈的初始化不是很好吗？

确实是。实际上，C++将自动调用若干个成员函数。当创建类时，它将调用一个成员函数。这个函数叫构造函数(constructor function)，它与类有相同的名字。例如，stack类的构造函数名为stack（在类外也叫stack::stack）。

当说明一个变量时，这个变量就被创建了。（也可以用new运算符来创建它，new运算符将在第20章“高级指针”中再作介绍。）

你想让这个栈自动进行初始化，所以删去init函数，用构造函数stack::stack来取代它。

```
class stack {
    // ...
public:
    // Initialize the stack
    stack(void);
    // ...
};

inline stack::stack(void)
{
    count = 0;    // Zero the stack
}
```

你可能已经注意到了，在说明构造函数的时候，省略了返回类型void。构造函数永远不返回值，所以void就不需要了。事实上，如果有这个字的话，编译器将显示警告信息。

因为构造函数被自动调用，所以现在的程序更简单了。我们不用再写如下的语句：

```
main()
{
    stack a_stack;    // Stack we want to use

    a_stack.init();
}
```

你只需写：

```
main()
{
    stack a_stack;    // Stack we want to use
}
```



```
}
```

现在可以使用类了。当你说明类的变量时，必须在说明中指定两个参数。例如：

```
main()
{
    person sam("Sam Jones", "555-1234");
```

和其它函数一样，构造函数也可以重载。以`person`为例，你注意到有的人没有电话。所以，需要建立一个只带有一个参数的构造函数：

```
class person {
    // ... rest of the class
public:
    person(const char i_name[]);
};

person::person(const char i_name[])
{
    strcpy(name, i_name);
    strcpy(phone, "No phone");
}
```

这样，你有了两个构造函数：一个带有一个参数，另一个带有两个参数。你没有定义不带参数的构造函数，所以在说明`person`类型的变量时，不能一个参数都没有。即：

```
person unnamed_source;    // Illegal
```

将产生一条错误信息。

带参数的析构函数

没有带参数的析构函数。析构函数永远都不带参数，也不返回值。它们只是删除变量。

拷贝构造函数

拷贝构造函数是一类特殊的构造函数，它用来生成类的一个精确备份。例如，用于`stack`类的一个拷贝构造函数为：

```
stack::stack(const stack &old_stack)
{
    int i;    // Index used to copy the data

    for (i = 0; i < old_stack.count; ++i)
        data[i] = old_stack.data[i];
    count = old_stack.count;
}
```

下面仔细研究这个函数。说明：

```
stack::stack(const stack &old_stack)
```

表明，这是一个拷贝构造函数。单参数(const stack &old_stack)表明，这个特殊的构造函数是一个拷贝构造函数。它把当前类转为参数的精确备份。

代码：

```
for (i = 0; i < old_stack.count; ++i) {  
    data[i] = old_stack.data[i];  
count = old_stack.count;
```

取出原来栈中的所有数据，把它放到新栈中。

拷贝运算符也可以直接调用，见下面的例子：

```
stack a_stack;           // A simple stack  
  
a_stack.push(1);         // Put a couple of elements on the stack  
a_stack.push(2);  
  
stack b_stack(a_stack);   // Create a copy of the stack
```

从表面看，拷贝构造函数好像不太重要。但你是不是还记得，我们在第9章“变量作用域和函数”中讨论过，C++向一个函数传递参数时有不同的方法，按值调用是其中一种方法。也就是，拷贝一个参数并把它传给函数。

当栈或是其它的类作为一个按值传送的参数传送时，就使用拷贝构造函数建立一个类的备份。

在下面的代码中，我们写了一些注解，向读者说明C++将自动调用的函数。

```
void use_stack(stack local_stack)  
{  
    local_stack.push(9);  
    local_stack.push(10);  
    .. Do something with local_stack  
}  
  
main()  
{  
    stack a_stack;           // Generate a default stack  
    a_stack.push(1);  
    a_stack.push(2);  
  
    use_stack(a_stack);  
  
    // Prints "2"  
    cout << a_stack.pop() << '\n';
```

local_stack::~~stack() called

astack.stack called

local_stack.stack(a_stack) called
(This is part of the parameter-passing mechanism)

正如你看到的，C++做了许多工作。从说明a_stack开始，C++还调用缺省的构造函数来创建a_stack。

使用变量a_stack，然后把它传送给函数use_stack。因为a_stack是按值传送的，所以必须用拷贝构造函数local_stack.stack(a_stack)建立一个栈的备份。

然后，函数向local_stack中增加若干项。注意：这是a_stack的一个备份，所以对local_stack的任何操作，都不影响a_stack的状态。在函数结束的时候，local_stack中含有四项内容：1、2、9、10，而a_stack中含有两项内容：1、2。

最后，函数调用结束后，可以输出a_stack的栈顶元素，它是“2”。

自动生成的成员函数

每个类都有一个构造函数和一个析构函数。如果程序员没有写这些成员函数，C++会自动生成这些函数。而且，像拷贝构造函数这样的成员函数可被自动调用。

自动生成与使用的函数

`class::class()` 缺省的构造函数

如果没有定义其它的构造函数，该函数将自动生成。生成的代码将用随机数给数据成员赋值。

当不使用参数来说明这个类的变量时，将自动调用该函数，如：

类类型 变量；

`class::class(const class &old_class)` 拷贝构造函数

除非程序员明确定义一个拷贝构造函数，否则将自动生成该函数。C++生成的这个函数将把原类中所有的数据成员备份到新类中。

当向一个函数传送按值调用的参数时，将自动调用该函数。当创建一个变量的备份时，也可能调用该成员函数：

调用类型 第一个变量；

// Call copy constructor to

// make duplicate of first_var

类类型 第二个变量(第一个变量)；

`class::~class()` 析构函数

除非程序员已定义，否则自动生成该函数。当一个变量消失时，自动调用该函数。当在变量作用域之外时，变量消失。（它还被delete运算符所调用，delete运算符将在第20章“高级指针”中讨论。）

`class class::operator = (const class &old_class) 赋值运算符`

(运算符重载将在第18章“运算符重载”中讨论。)

该函数自动生成，它处理一个类对另一个类的赋值。C++生成的这个函数把原类中所有数据成员的值拷贝到新类中。

类类型 变量1;

类类型 变量2;

变量1 = 变量2; // "operator =" called

快捷方法

到目前为止，你在自己创建的类中只使用了函数原型，你还可以在类的内部定义函数体。所以：

```
class stack {
    public:
        // ... rest of class

        // Push an item on the stack
        void push(const int item);
};
inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}
```

可以写为：

```
class stack {
    public:
        // ... rest of class

        // Push an item on the stack
        void push(const int item) {
            data[count] = item;
            ++count;
        }
};
```

在第二种写法中，并不一定非要写inline命令，因为在类中说明的所有函数都自动被说明为是内联的。

风格

类的编程风格很像结构和函数的编程风格。每个成员变量都应该有注释来解释，每个成员函数也应该有注释，这与函数是一样的。

但是，你给成员函数的原型所加的注释，不同于给普通函数原型所加的注释。对于普通函数，你可以在原型的前面放上一个解释整个函数的注释块。如果你对类的成员函数也这样做的话，注释会使类的结构不清楚。这是由注释引起的众多麻烦中的一种。所以你应该在每个成员函数原型的前面放上一行注释，完整的注释应该放在函数本身的前面。

内联函数的函数体整个都在类中说明，那么，对于内联函数怎么办呢？你如何为它们写注释？如果你加入完整的注释块，就会弄乱整个类的结构。如果你只放一行注释，会忽略掉很多有用的信息。合适的注释是一种折衷艺术。你需要把有用的信息放到类中，而把无用的信息删掉。

解决的办法是，保持内联函数尽量地小。这样做有两个理由，第一，所有的内联函数都应该小。第二，在类的内部说明大函数，会使类过分复杂。好的原则是，如果函数的代码多于5行，就把原型放到类中，把函数体留在外面。

结构非常小的成员函数的意义很明显，所以不需要一个大的注释块。如果函数的含义不明显，需要有注释的话，可以把原型放在类中，而把注释和函数体放在程序中。

在第一个成员变量之前，C++并没有要求必须有存取保护说明（`public`、`private`或是`protected`）。下面的代码完全合法：

```
class example {
    int data;
    // ...
};
```

但是，`data`的存取保护是什么？是`public`、`private`还是`protected`？如果有明确说明的话，你就不用担心这样的问题了。（存取保护的缺省值是`private`。）

最后，C++将自动生成某些成员函数，如缺省的构造函数、拷贝构造函数和赋值运算符。假如你有一个类，它没有指定拷贝构造函数的话，例如：

```
// Comments describing the class
// Note: The style of this class leaves something to be desired
class queue {
    private:
        int data[100];    // Data stored in the queue
        int first;        // First element in the queue
        int last;         // Last element in the queue
    public:
        queue();           // Initialize the queue
```

```

        void put(int item); // Put an item in the queue
        int get(void);      // Get an item from the queue
    };

```

创建这个类的程序员是不是忘记了拷贝构造函数？拷贝构造函数是不是由C++自动生成，或是程序员设计的这个类知道永远也不需要调用拷贝构造函数？这些重要的问题并没有在类中说明。

所有的类都有一个缺省的构造函数、拷贝构造函数、赋值运算符和析构函数。如果你想使用C++自动生成的一个函数的话，就把注释放到类中，以表明要使用缺省的函数。

```

// Comments describing the class
class queue {
    private:
        int data[100];    // Data stored in the queue
        int first;        // First element in the queue
        int last;         // Last element in the queue
    public:
        queue();           // Initialize the queue
        // queue(const queue &old_queue)
        //      Use automatically generated copy constructor

        // queue operator = (const queue &old_queue)
        //      Use automatically generated assignment operator

        // ~queue()
        //      Use automatically generated destructor

        void put(int item); // Put an item in the queue
        int get(void);      // Get an item from the queue
};

```

一眼就能看出，程序员想让C++自动生成的是哪些成员函数，而且，明确写明这些在程序设计时很重要。

C++自动生成的拷贝构造函数相当简单而且有限。它并不一定在任何情况下都能工作，例如后面将看到的构造更复杂的类时的情况。可是，当自动生成的拷贝构造函数不工作时，会怎么样呢？你不会想自己建立一个吧？

总之，你可能决定，一个类永远不被拷贝（或者，如果被拷贝的话，就出错）。

解决办法是创建一个哑拷贝构造函数，它输出一条错误信息，并中断程序的执行：

```

class no_copy {
    // Body of the class
    public:
        // Copy constructor

```

```

        no_copy(const no_copy &old_class) {
            cerr <<
                "Error: Copy constructor for 'no_copy' called. Exiting\n";
            exit(8);
        }
};

```

这个函数有些作用。但问题是，它只在运行时才发现错误，而不是在编译时发现。你想尽可能早地发现错误，所以这个办法没有用处。

但是，你可以不让编译器自动调用拷贝构造函数。办法是，把它说明为私有的。你宣布：“这是一个拷贝构造函数，但是别人不能用它。”

```

class no_copy {
    // Body of the class
private:
    // There is no copy constructor
    no_copy(const no_copy &old_class);
};

```

当编译器试图去使用拷贝构造函数时，你会得到一条错误信息，如“Error: Attempt to access private member function.”

注：因为拷贝构造函数永远也不被调用，所以永远也不用定义它的函数体。

编程练习

练习13.1：编写一个parity类。它对送入的任意个数进行判定，如果项数是偶数，则返回TRUE，如果是奇数，则返回FALSE。

成员函数为：

```

void parity::put(void); // Count another element
int parity::test(void); // Return TRUE(1) if an even number of
                        // puts have been done. Return FALSE(0)
                        // for an odd number.

```

练习13.2：编写“checkbook”类。把一串数放到类中，并得到总和。

成员函数为：

```

void check::add_item(int amount); // Add a new entry to the checkbook
int check::total(void);           // Return the total of all items

```

练习13.3：编写一个类，实现简单的队列。一个队列类似于一个栈，只是数据的移动遵循按先进先出（FIFO）的顺序。

成员函数为:

```
void queue::put(int item); //Insert an item in the queue
int queue::get(void);      // Get the next item from the queue
```

简单使用法:

```
queue a_queue;

a_queue.put(1);    // Queue contains: 1
a_queue.put(2);    // Queue contains: 1 2
a_queue.put(3);    // Queue contains: 1 2 3

cout << a_queue.get() << '\n'; // Prints 1, queue contains 2 3
cout << a_queue.get() << '\n'; // Prints 2, queue contains 3
```

练习13.4: 定义一个类, 它存储一组从0到31的整数。可以用set成员函数来设置元素, 用clear成员函数来清除它。设置一个已经设置的数, 或是清除一个已经清除的数都不算错误。函数test用来指明一个元素是否被设置了。

成员函数为:

```
void small_set::set(int item); // Set an element in the set
void small_set::clear(int item); // Clear an element in the set
int small_set::test(void);      // See whether an element is set
```

简单使用法:

```
small_set a_set;

a_set.set(3);      // Set contains [3]
a_set.set(5);      // Set contains [3,5]
a_set.set(5);      // Legal (set contains [3,5])

cout << a_set.test(3) << '\n'; // Prints "1"
cout << a_set.test(0) << '\n'; // Prints "0"
```

练习13.5: 我有一个学习外语单词的简单方法。我在一串一闪而过的卡片上写上单词。然后从头到尾过一遍这些卡片。如果我认对了, 则卡片消失。如果错了, 则这张卡片翻到栈底。

写一个类来实现这个机制。成员函数为:

```
struct single_card {
    char *question; // English version of the word
    char *answer;   // Other language version of the word
};
```

```
// Constructor -- takes a list of cards to initialize the flash card stack
void flash_card::flash_card(single_card list[]);

// Get the next card
void flash_card::next(single_card &next_card);

void flash_card::right(void);    // The student got the current card
                                right
void flash_card::wrong(void);    // The student got the current card
                                wrong
```

本章内容:

- 友元
- 常量函数
- 常量成员
- 静态成员变量
- 静态成员函数
- static的含义
- 编程练习

第14章

类的进一步讨论

友元

在第13章“类的初步讨论”中，已经定义了一个基本的栈类。假定你想写一个函数，来判定两个栈是否相等。乍看上去，这很简单。此函数如示例14.1所示。

示例14.1 `stack_c/s_equal.cc`

```
/* *****  
 * stack_equal -- Test to see whether two stacks are  
 *               equal  
 *  
 *      s1, s2 -- the two stacks  
 *  
 * Returns  
 *      0 -- stacks are not equal  
 *      1 -- stacks are equal  
 * ***** */  
int stack_equal(const stack &s1, const stack &s2)  
{  
    int index; // Index into the items in the array  
  
    // Check number of items first  
    if (s1.count != s2.count)  
        return (0);  
  
    for (index = 0; index < s1.count; ++index) {  
        if (s1.data[index] != s2.data[index])  
            return (0);  
    }  
    return (1);  
}
```


和许多程序一样，这个方法简单、清楚，但却不正确。问题在于，成员变量count和data是私有的。这就是说，你不能访问它们。

那怎么办呢？一个解决办法是，让这些变量变为公有的。这样，函数stack_equal就可以存取count和data了。但是，这样的话，其它的函数也可以访问这两个量了，可这又不是我们的初衷。

所幸的是，C++可以实现这样的操作：“让stack_equal并且只让stack_equal能存取stack类的私有数据data。”通过friend命令就可以实现这一点。类必须说明它们的友元函数。类外的任何函数都不能存取类中的私有数据，除非类允许它们这样做。

```
示例14.2 stack_c/f_stack.cc
// The stack itself
class stack {
    private:
        int count;           // Number of items in the stack
        int data[STACK_SIZE]; // The items themselves
    public:
        // Initialize the stack
        void init(void);

        // Push an item on the stack
        void push(const int item);

        // Pop an item from the stack
        int pop(void);

        friend int stack_equal(const stack &s1, const stack &s2);
};
```

注意

stack_equal不是类stack的成员函数。它是个一般的、简单的函数。这两者的唯一区别是，因为这个函数是友元函数，所以它可以访问把它作为友元函数的类的私有数据。

友元类

友元不仅仅局限于函数。一个类也可以是另一个类的友元。例如：

```
class item {
    private:
        int data;
        friend class set_of_items;
};
```

```
class set_of_items {
    // ...
};
```

这里，因为类set_of_items是item的友元，所以它可以存取item的所有成员。

常量函数

C++允许你定义两类数：常量和非常量。例如：

```
int index;    // Current index into the data array
const int DATA_MAX(100); // Maximum number of items in the array
```

这两个量的处理是不一样的。例如，你可以改变index的值，但不能改变DATA_MAX的值。

现在，让我们考虑实现一组从0到31之间数的一个类。类的定义为：

```
// Warning: The member functions in this class are incomplete
//          See below for a better definition of this class
class int_set {
    private:
        // ... whatever
    public:
        int_set(void);           // Default constructor
        int_set(const int_set &old_set); // Copy constructor
        void set(int value);     // Set a value
        void clear(int value);  // Clear an element
        int test(int value);    // See whether an element is set
};
```

说到数，C++允许你定义两类数：常量和非常量。

```
int_set var_set;    // A variable set (we can change this)
var_set.set(1);     // Set an element in the set
// Define a constant version of the set (we cannot change this)
const int_set const_set(var_set);
```

在int_set类中，成员函数set和clear修改了设置的值。另一个函数test没有修改任何值。很明显，你不想让set和clear对常量进行操作。不过test成员函数却可以这样做。

C++怎样才能知道哪些函数可以用于常量，而哪些不行呢？办法就是在函数头的结尾放一个关键字const。这个关键字告诉C++，这个成员函数可以用于一个常变量。所以，如果你把const放在成员函数test的后面，C++将允许它用于一个常量。成员函数set和clear没有这个关键字，所以不能对常量进行操作。


```

class int_set {
    private:
        // ... whatever
    public:
        int_set(void);           // Default constructor
        int_set(const int_set &old_set); // Copy constructor
        void set(int value);     // Set a value
        void clear(int value);  // Clear an element
        int test(int value) const; // See whether an element is set
};

```

这样说明后，就可以在代码中写下面的语句了：

```

int_set var_set;    // A variable set (we can change this)

var_set.set(1);     // Set an element in the set (legal)

// Define a constant version of the set (we cannot change this)
const int_set const_set(var_set);

// In the next statement we use the member function "test" legally
cout << "Testing element 1. Value=" << const_set.test() << '\n';

const_set.set(5);   // Illegal (set is not allowed on a const)

```

常量成员

类可以包含常量成员，问题是常量在类中和类外的行为略有不同。在类外，一个常变量的说明必须要被初始化。例如：

```

const int data_size = 1024;    // Number of data items in the input
                                stream

```

在类的内部，说明常量时不用对它进行初始化。例如：

```

class data_list {
    public:
        const int data_size;    // Number of items in the list
        // ... rest of the class
};

```

常量成员变量由构造函数进行初始化。这不能简单地写为：

```

class data_list {
    public:

```

```

    const int data_size;    // Number of items in the list

    data_list(void) {
        data_size = 1024;    // This code won't work
    };
    // ... rest of the class
};

```

因为`data_size`是一个常量，所以它的初始化要用特殊的语法：

```

    data_list(void) : data_size(1024) {
    };

```

但是，如果你只想在类中放一个简单的常量怎么办？不幸的是，C++不允许你这样做：

```

class foo {
public:
    const int foo_size = 100;    // Illegal

```

你只能有两种选择：

1. 把常量放到类外：

```

const int foo_size = 100;    // Number of data items in the list
class foo {

```

这会使`foo_size`成为随处可用的常量。

2. 使用语法技巧，欺骗C++定义一个常量：

```

class foo {
public:
    enum {foo_size = 100};    // Number of data items in the list

```

它把`foo_size`定义为一个常量，它的值是100。实际上，C++把`foo_size`说明为`enum`类型的一个元素，并赋以值100。因为C++把`enum`看作是整数，所以这样的定义就等同于定义了整数常量。

这种办法的缺点是：这是一种技巧，它只能对整数有效，而且它利用的是C++语法上的漏洞；当语言改进时，这种漏洞可能不再存在了。这样的代码可能给维护代码且不熟悉这种技巧的其他程序员造成困难。

静态成员变量

假定你在任何时刻都想知道正在使用的栈的数目。一种方法是，建立一个全局变量`stack_count`。当调用一次栈的构造函数时，该量增1；当调用一次析构函数时，该量减1。

```

int stack_count = 0; // Number of stacks currently in use

class stack {
private:
    int count;          // Number of items in the stack
    // ... member variables
public:
    int data_count; // Number of items in the stack
    // ... member variables
    stack() {
        // We just created a stack
        ++stack_count;
        count = 0;
    }
    ~stack() {
        // We now have one less stack
        --stack_count;
    }
    // ... other member functions
};

```

注意，`stack_count`是单一的一个全局变量。不论你创建了多少不同的栈，也只有一个`stack_count`。

虽然这种机制有点作用，但也有缺点。类`stack`的定义中包含了`stack`的所有信息，只是没有`stack_count`变量。把`stack_count`放到类中很不错，但如果你把它定义为成员变量，则每次说明一个`stack`类变量时，就得到`stack_count`的一个新拷贝。

C++有一个用于成员变量的特殊修饰符：`static`。它告诉C++为这个类定义的这个变量有一个，且只有一个。

```

class stack {
private:
    static int stack_count; // Number of stacks currently in use
    int count;             // Number of items in the stack
    // ... member variables
public:
    stack() {
        // We just created a stack
        ++stack_count;
        count = 0;
    }
    ~stack() {
        // We now have one less stack
        --stack_count;
    }
}

```

```

        // ... other member functions
    };

```

后一种写法和使用全局变量时的写法差不多。只是没有对`stack_count`进行初始化。这个工作由下面的语句完成：

```
int stack::stack_count = 0; // No stacks have been defined
```

静态和非静态成员变量的区别是：如果你定义了三个栈，那么你建立了三个不同的`data_count`成员变量，但只有唯一的一个`stack_count`。成员变量属于各个栈。静态变量属于类。所以，如果：

```
stack a_stack;
stack b_stack;
```

则`a_stack.stack_count`等同于`b_stack.stack_count`。`stack`类只有唯一的`stack_count`。C++使用下面的语法访问这个量：

<类>::<变量>

所以，访问`stack_count`时要使用下面的语句：

```
cout << "The number of active stacks is " << stack::stack_count << '\n';
```

（如果`stack_count`不是私有的，至少可以这样使用。）

静态成员函数

成员变量`stack_count`被定义为私有量。这就是说，在类外，任何函数都不能访问它。你想知道定义了多少个栈，所以需要有一个函数来得到`stack_count`的值。最佳办法是：

```
class stack {
    static int stack_count; // Number of stacks currently in use
    // ... member variables
public:
    // Not quite right
    int get_count(void) {
        return (stack_count);
    }
    // ... other member functions
};

```

这段代码可以工作，但是需要一个`stack`类型的变量来访问这个函数。

```
{
    stack temp_stack; // Stack for getting the count
    cout << "Current count " << temp_stack.get_count() << '\n';
}

```

```
}
```

因为`get_count`不使用`stack`中的任何非静态数据，所以可以把它说明为静态成员函数。

```
class stack {
    static int stack_count; // Number of stacks currently in use
    // ... member variables
public:
    // Right
    static int get_count(void) {
        return (stack_count);
    }
    // ... other member functions
};
```

现在你可以访问静态成员函数`get_count`，这很像访问静态成员变量`stack_count`：

```
cout << "The number of active stacks is " << stack::get_count() << '\n';
```

静态成员函数的限制很多。它们不能访问类中的非静态成员变量或函数。它们只能访问静态成员数据、静态成员函数和类外的函数和数据。

static的含义

在C++中，关键字`static`有许多不同的含义。表14.1中列出了`static`的所有使用方法。

表14.1 static的含义

用 法	含 义
函数体之外的变量	变量作用域局限于说明它的文件中。
函数内部的变量说明	变量是永久的。它只被初始化一次，即使递归调用这个函数，该变量也只建立一个。
函数说明	函数作用域局限于说明它的文件中。
成员变量	为每个类建立一个变量拷贝（不是为每个变量）。
成员函数	函数只能访问类的静态成员。

编程练习

练习14.1：两个类共享一个文件。在程序的其它部分需要知道此时文件是否被使用。建立一个函数，当文件被其中的一个类使用时，返回1。

练习14.2：你可能编写过Dr. Able Smith提出的有关PHD（猪、马、狗）的程序。为

每种动物定义一个类。每个类都应该记录相关动物的数量，这由一个私有静态变量来定义。最后定义一个函数，返回所有三种动物的总数。

练习14.3：编写一个类，该类的每个实例可以访问一个栈——并不是每个实例有一个栈，而是只有一个栈。当类的任何实例独占这个栈时，都可以锁定它，过后再开锁。定义一个成员函数，完成加锁和开锁操作。

如果你还有兴趣，可以让开锁函数检查类的当前实例与前一次给栈加锁的实例是否相同。

练习14.4：你需要提供一些I/O程序来处理文件中的行号。行号类的基本定义是：

```
class line_number {
public:
    void goto_line(int line);
    int get_current_line(void);
    long int get_char_pos(void);
}
```

成员函数定义为：

```
void goto_line(int line);
```

对输入文件定位到指定行。

```
int get_current_line(void);
```

返回当前行号（与goto_line所设置的一样）。

```
long int get_char_pos(void);
```

返回当前行的字符位置。（这是个技巧。）

在任何时刻可以使用几个line_number类。类来管理它自己的内部表，所以它知道目前有哪些line_number在用。当调用goto_line时，函数将扫描line_number类的内部表，找出离给定行号最近的一个，并用它去查找给定的行。

例如，假定有4个活动的line_number变量：

变 量	位 置
beginning	第0行
chapter_start	第87行
current_heading	第112行
current_location	第52行

你想把current_location移到第90行。goto_line函数将查找该表，找到最近的一个新位置（此处是chapter_start），并用它跳到87行。然后，一个字符一个字符地读文件，直到它找过三个行尾字符，把位置定到第90行为止。

本章内容:

- 常量指针
- 指针和输出
- 指针和数组
- 分隔串
- 指针和结构
- 命令行自变量
- 编程练习
- 答疑

第15章

简单指针

下图中有一物体，还有一个指向该物体的指针（图15.1）。

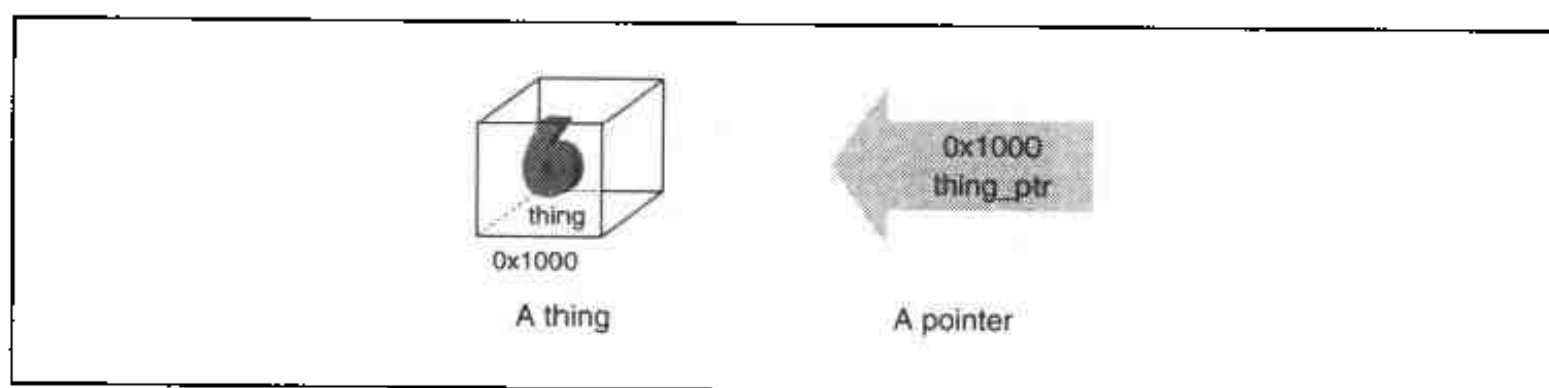


图 15.1 一个物体和一个指向该物体的指针

物体可以是任何尺寸；有的大，有的小。指针只有一种规格（相对地要小一些）*。

本书自始至终用一个盒子来代表物体。盒子可大可小，但总是一个盒子。指针用箭头来表示。

大多数编程新手对指针和它们指向的物体感到迷惑不解。为了简化这个问题，本书中的所有指针变量都使用扩展名_ptr。在自己的程序中，你可能也想沿用这个惯例。虽然它不像应有的那样流行，但也是非常有用的。

图15.1显示了一个物体：一个名为thing的变量。它的名字写在表示它的盒子上，值为6。变量的实际地址是0x1008。C++编译器在编译时自动为每个变量分配一个地址。机器不同，实际的地址也不同。大多数情况下，你不必操心变量的地址，因为编译器会处理这些事情。（毕竟你不了解地址的细节，但也学习了11章的编程技术。）

指针（thing_ptr）指向这个变量thing。指针也被称为地址变量(address variable)，因为它们包含其它变量的地址。此例中，指针包含地址0x1008。因为这是thing的地址，所以说

* 在Turbo-C++中，并不总是这样。因为8086特殊的体系结构，所以强迫Turbo-C++使用了near指针（16位）和far指针（32位）。详见Turbo-C++手册。

thing_ptr指向thing。（你可以把另一个地址放到thing_ptr中，强迫它指向别的地方。）

日常生活中，我们都使用“物体”和“地址”这样的字眼。例如，你生活在一所房子内（一件物体）。街道地址可能是“123 W. Main Street”。在纸上写一个地址是件区区小事。但是把房子放到纸上就困难多了，可能需要一台起重机。因为街道地址都差不多长短：占用一行。但房子却大小不一。所以“1600 Pennsylvania Ave”可能是指一所大房子，而“8347 Skid Row”可能是指一居室的棚屋，虽然两个地址的长短差不多。

许多不同的地址变量可以指向同一个物体，对街道地址来说也一样。表15.1中列出小镇上重要的服务机构的地址。

表15.1 小镇地址名录

服务机构（变量名）	地址（地址值）	建筑物（物体）
Fire department	1 Main Street	City Hall
Police station	1 Main Street	City Hall
Planning office	1 Main Street	City Hall
Gas station	2 Main Street	Ed's Gas Station

这里有一座大的多用途的建筑物，它被几个服务机构所共用。虽然这里有三个地址变量（服务机构），但只有一个地址（1 Main Street）指向该建筑物（City Hall）。

正如你将在本章中看到的，指针可以用来快速简单地访问数组。在后面的章节中，你将发现，如何用指针建立新的变量和复杂的数据结构，如链表和树。当你读完本书的其余章节后，就可以了解这些数据结构，同时自己也可以建立了。

在说明语句中，指针说明是在变量名之前加一个星号（*）：

```
int thing;           // Define "thing" (see Figure 15.2A)
int *thing_ptr;      // Define "pointer to a thing" (see Figure 15.2B)
```

表15.2中列出了与指针相连的运算符。

表15.2 指针运算符

运算符	含 义
*	逆引用（给定一个指针，得到所引用的物体）
&	地址（给定一个物体，得到指向它的地址）

&符号把物体改为指针。*符号把指针改为物体。这些运算符很容易混淆。让我们详细地研究研究这些运算符使用的例子。

thing 一个物体

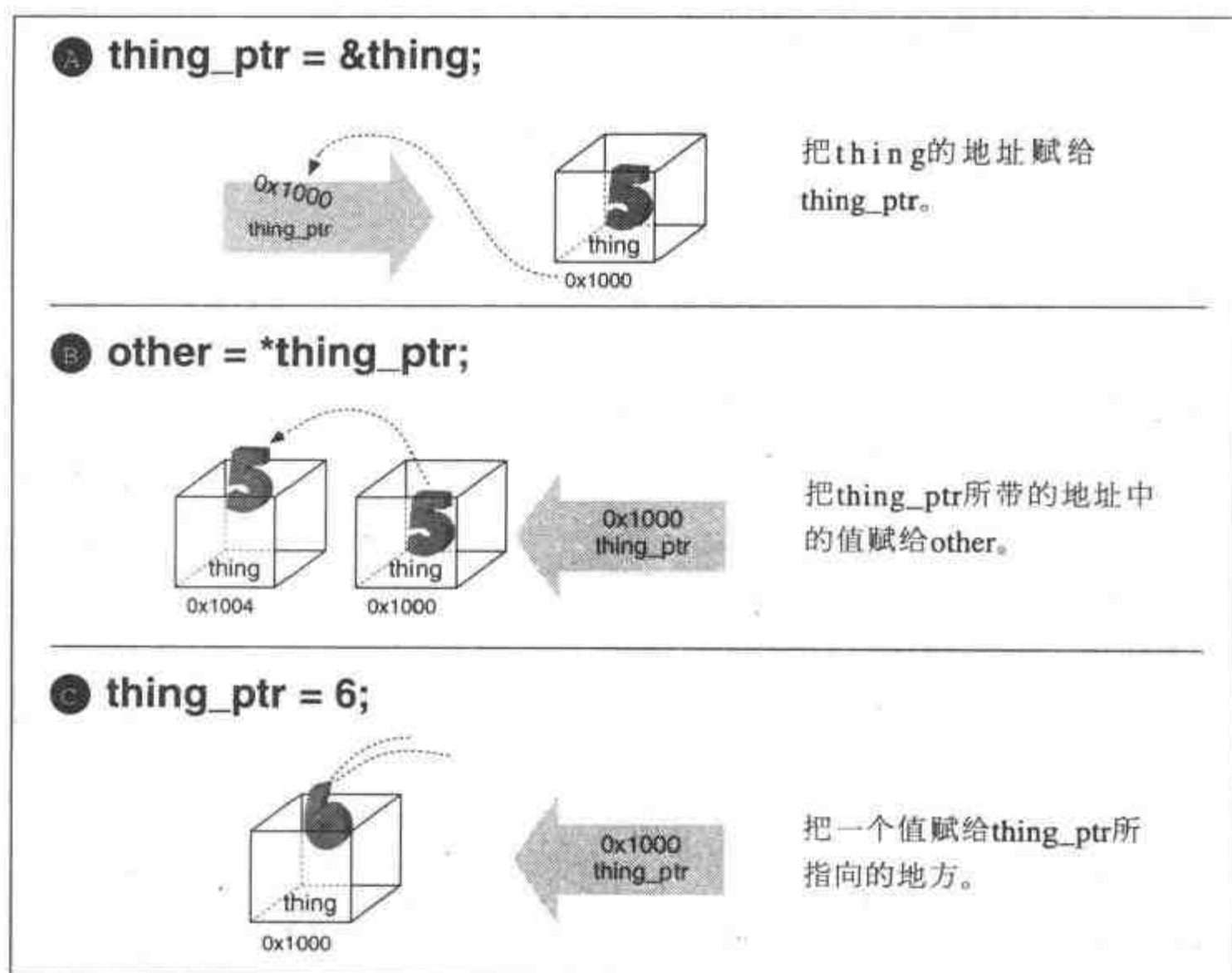


图15.2 指针运算符

说明`int thing`并没有包含一个星号，所以`thing`不是指针。例如：

```
thing = 4; // See Figure 15.2C
```

`&thing` 指向物体的一个指针

`thing`是一个对象。`&`运算符得到这个对象的地址（一个指针），所以`&thing`是一个指针。例如：

```
thing_ptr = &thing; // Point to the thing
                // (See Figure 15.2A)
*thing_ptr = 5;     // Set "thing" to 5
                // (See Figure 15.2B)
```

`thing_ptr` 物体指针

说明中的星号表示这是一个指针。而且你可以把扩展名`_ptr`放到名字中。

`*thing_ptr` 一个物体

变量`thing_ptr`是一个指针。`*`（逆引用运算符）告诉C++去关注指针所指向的数据，而不是指针本身。注：它指向任何一个整数。它可以指向具体的变量`thing`，也可以不指向它。

```
*thing_ptr = 5;    // Assign 5 to an integer
                  // We may or may not be pointing
                  // to the specific integer "thing"
```

下面是错用指针运算符的示例。

***thing** 非法。它告诉C++要得到由变量**thing**所指向的对象。因为**thing**不是一个指针，所以这是个无效的运算。

&thing_ptr 合法，但是很奇怪。**thing_ptr**是一个指针。**&**运算符得到一个指向对象（即**thing_ptr**）的指针。此处得到一个指向指针的指针。（指向指针的指针在更复杂的程序中确实会出现。）

示例15.1说明了指针的简单使用。它说明了一个对象**thing_var**和一个指针**thing_ptr**。**thing_var**的值由下面的语句来设置：

```
thing_var = 2;
```

语句：

```
thing_ptr = &thing_var;
```

让C++把**thing_var**的地址赋给**thing_ptr**。此时，**thing_var**和***thing_ptr**是一样的。

示例15.1 *thing/thing.cc*

```
#include <iostream.h>
main()
{
    int    thing_var; // Define a variable
    int    *thing_ptr; // Define a pointer

    thing_var = 2;      // Assigning a value to "thing"
    cout << "Thing " << thing_var << '\n';

    thing_ptr = &thing_var; // Make the pointer point to "thing"
    *thing_ptr = 3;         // thing_ptr points to thing_var so
                           // thing_var changes to 3
    cout << "Thing " << thing_var << '\n';

    // Another way of printing the data
    cout << "Thing " << *thing_ptr << '\n';
    return (0);
}
```

几个指针可以指向同一个目标：

```
1:      int    something;
2:
```

```

3:      int      *first_ptr;      // One pointer
4:      int      *second_ptr;     // Another pointer
5:
6:      something = 1;            // Give the thing a value
7:
8:      first_ptr = &something;
9:      second_ptr = first_ptr;

```

在第8行，使用&运算符把一个简单变量（something）改为一个指针，然后赋给了first_ptr。因为first_ptr和second_ptr同样是指针，所以可以像第9行那样直接赋值。

运行该代码段后，得到的情形如图15.3所示。

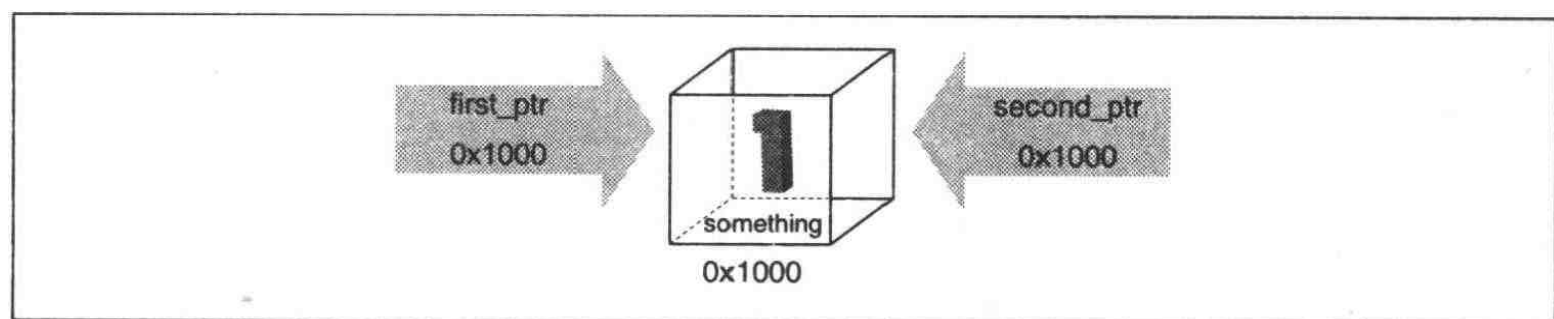


图 15.3 两个指针和一个目标

注意有一点很重要，即你有三个变量，但只有一个整数（thing）。下面的语句是等价的：

```

something = 1;
*first_ptr = 1;
*second_ptr = 1;

```

最后，有一个特殊的指针名叫NULL，它不指向任何对象。（实际值是0。）标准include文件stddef.h中定义了常量NULL。（大多数标准include文件虽然没有对指针做任何处理，但也都包含NULL。）NULL指针如图15.4所示。

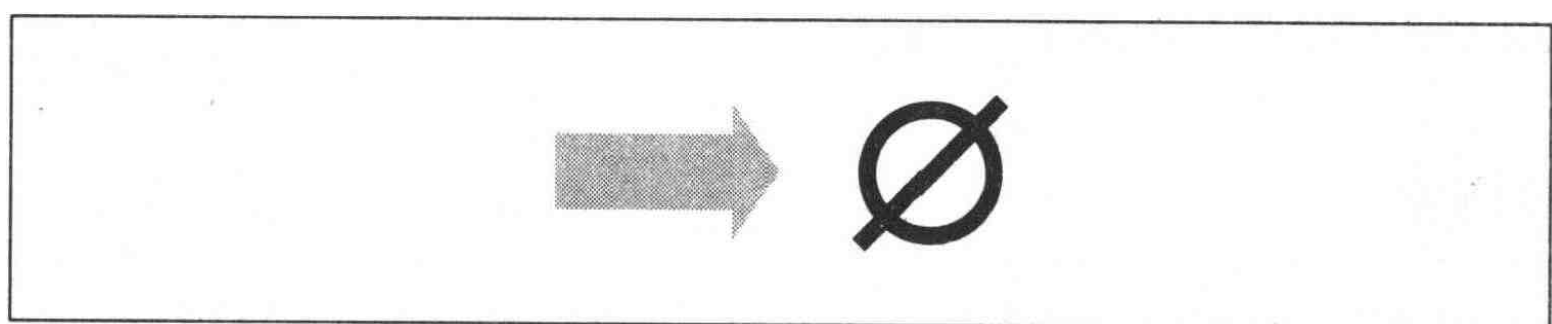


图15.4 NULL

常量指针

说明常量指针是一个小技巧。例如，说明：

```
const int result = 5;
```

告诉C++: `result`是一个常量, 所以

```
result = 10;    // Illegal
```

是非法的。但是说明:

```
const char *answer_ptr = "Forty-Two";
```

并没有告诉C++数据 ("Forty_Two") 是一个常量。相反, 它告诉C++: `answer_ptr`是一个常量。指针不能被改变, 但是指向的数据却可以被改变。你应该了解了目标和指向目标的指针之间的区别了吧。

`answer_ptr`是什么? 是一个指针。它能被改变吗? 不能, 它是一个常量。它指向什么? 一个char数组。由`answer_ptr`指向的数据能改变吗? 可以, 它不是常量。

所以在C++中:

```
answer_ptr = "Fifty-One";    // Illegal (answer_ptr is a constant)
*answer_ptr = 'X';           // Legal (*answer_ptr is a char)
```

如果你把`const`放到*的后面, 那么告诉C++的是: 数据是常量。例如:

```
char *const name_ptr = "Test";
```

`name_ptr` 是什么? 一个指针。它能被改变吗? 是的。它指向什么? 一个常量字符。`name_ptr`指向的数据能改变吗? 不行。

```
name_ptr = "New";           // Legal (name_ptr is not constant)
*name_ptr = 'B';            // Illegal (*name_ptr is a const char)
```

最后, 你可以把`const`放在两个地方, 生成一个不能被改变的指针, 同时指向的数据项也不能被改变。

```
const char *const title_ptr = "Title";
```

指针和输出

在C++中, 你可以输出一个指针的值, 就像你可以输出一个整型或浮点型这样的简单变量的值一样。例如:

```
int an_integer = 5;          // A simple integer
int *int_ptr = &an_integer;  // Pointer to an integer

cout << "Integer pointer " << int_ptr << '\n';
```

输出

Integer pointer 0x58239A

此处，值0x58239A代表一个内存地址。这个地址因程序的不同而不同。

C++在对待字符指针时，与其它的指针略有不同。一个字符指针是一个指向串的指针。

例如：

```
char some_characters[10] = "Hello"; // A simple set of characters
char *char_ptr = &some_characters[0]; // Pointer to a character

cout << "String pointer " << char_ptr << '\n';
```

输出

String pointer Hello

指针和数组

C++允许指针进行运算，可以对指针进行加法和减法操作。假如你有下面的代码：

```
char array[10];
char *array_ptr = &array[0];
```

它表示的含义如图15.5所示。

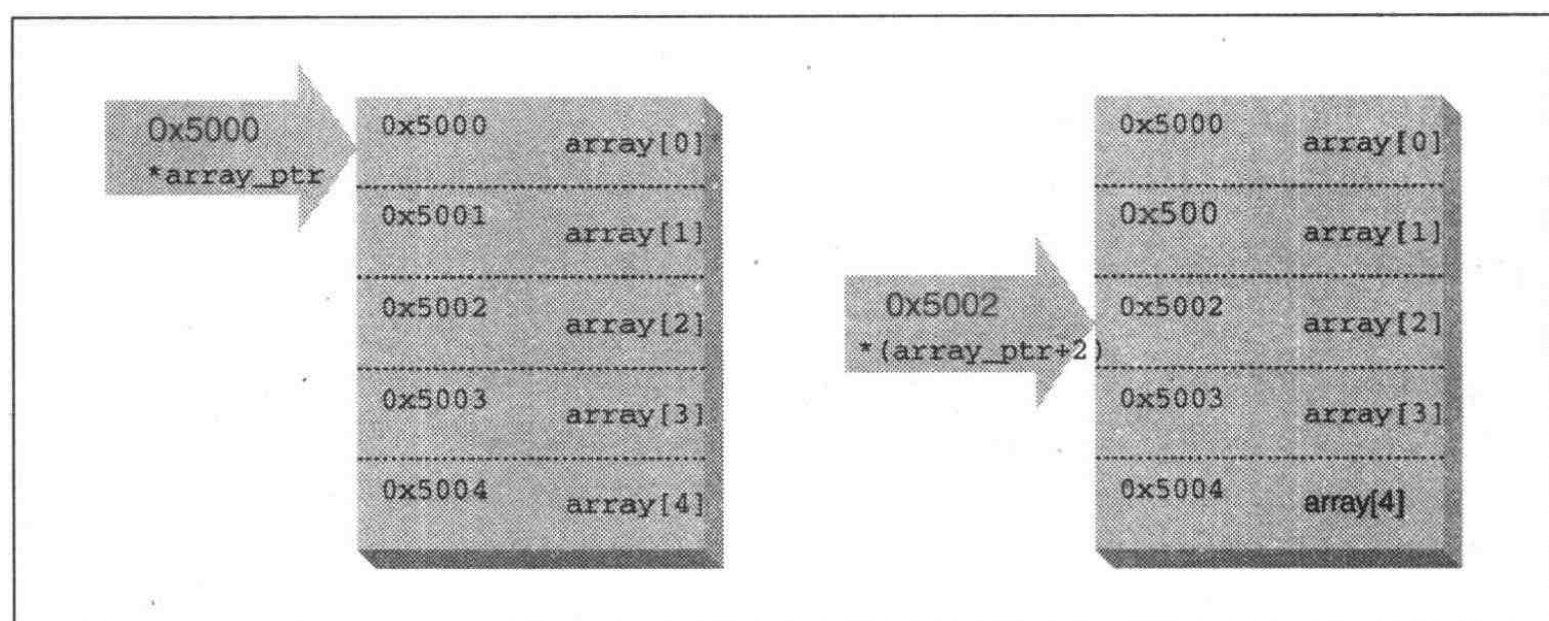


图 15.5 指针和一个数组

在这个例子中，`*array_ptr`就是`array[0]`，`*(array_ptr+1)`就是`array[1]`，依此类推。注意括号的使用。`(*array_ptr)+1`不等于`array[1]`。`+1`在括号外，在逆引用操作完成后进行加法运算。所以`(*array_ptr)+1`就是`array[0]+1`。

乍看上去，这好象是简单数组的复杂表示方法。你从简单指针运算开始，在后面的章节中，你将使用更复杂的指针来高效地处理更困难的函数。

指针占用很少的内存地址。在一个数组中，所有的元素都分配在连续的地址中。例如，`array[0]`可能放在地址`0xff000024`中。则`array[1]`应该放在地址`0xff000025`中，依此类推。示例15.2输出简单字符数组中的元素及其地址。（注：I/O操作符`hex`和`dec`将在第16章“文件输入/输出”中介绍。）

示例15.2 `array-p/array-p.cc`

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

const int ARRAY_SIZE = 10; // Number of characters in array
// Array to print

char array[ARRAY_SIZE] = "012345678";

main()
{
    int index; /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        cout << hex;      // Trick to print hex numbers
        cout <<
            "&array[index]=0x" << int(&array[index]) <<
            " (array+index)=0x" << int(array+index) <<
            " array[index]=0x" << int(array[index]) << '\n',
        cout << dec;
    }
    return (0);
}
```

运行该程序时，将输出：

```
&array[index]=0x20090 (array+index)=0x20090 array[index]=0x30
&array[index]=0x20091 (array+index)=0x20091 array[index]=0x31
&array[index]=0x20092 (array+index)=0x20092 array[index]=0x32
&array[index]=0x20093 (array+index)=0x20093 array[index]=0x33
&array[index]=0x20094 (array+index)=0x20094 array[index]=0x34
&array[index]=0x20095 (array+index)=0x20095 array[index]=0x35
&array[index]=0x20096 (array+index)=0x20096 array[index]=0x36
&array[index]=0x20097 (array+index)=0x20097 array[index]=0x37
&array[index]=0x20098 (array+index)=0x20098 array[index]=0x38
&array[index]=0x20099 (array+index)=0x20099 array[index]=0x0
```

字符通常占用一个字节，所以字符数组中的元素将分配连续的地址。一个短整型数占用两个字节，所以在短整型数组中元素的地址间隔为2。这是不是意味着`short_array+1`只能

用于字符而不能用于其它类型？不。C++自动进行指针的算术运算，所以这样的操作是正确的。本例中，`short_array+1`将指向下标为1的元素。

C++提供了一种处理数组时的简写法。我们不用写：

```
array_ptr = &array[0];
```

只需写：

```
array_ptr = array;
```

C++并不认真区分指针和数组，而把它们认为是同一类事物。这里，你把变量`array`当作指针使用，C++将自动进行必要的转换。

示例15.3统计元素中非零元素的个数，找到0时停止。它不进行界限检查，所以在数组中至少要有一个0。

示例15.3 *ptr2/ptr2a.cc*

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

```
int array[10] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
```

```
int index;
```

```
main()
```

```
{
```

```
    index = 0;
```

```
    while (array[index] != 0)
```

```
        ++index;
```

```
    cout << "Number of elements before zero " << index << '\n';
```

```
    return (0);
```

```
}
```

用指针改写这个程序，得到示例15.4。

示例15.4 *ptr2/ptr2.cc*

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

```
int array[10] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
```

```
int *array_ptr;
```

```
main()
```

```
{
```

```
    array_ptr = array;
```

```
    while ((*array_ptr) != 0)
```

```
        ++array_ptr;
```


示例15.5 init-a/cnit-a.cc

```
cout << "Number of elements before zero " <<
      (array_ptr - array) << '\n';
return (0);
}
```

第一个程序中使用了表达式`(array[index] != 0)`。这要求编译器进行下标操作，它比简单的指针逆引用操作`(* array_ptr) != 0`要花更多的时间。该程序结尾的表达式`array_ptr - array`计算`array_ptr`在数组中的位置。

当把数组传递给一个过程时，C++将自动把这个数组改为一个指针。事实上，如果你在数组前放上`&`号，C++将输出一个警告信息。示例15.5是数组传递的例子。

示例15.5 init-a/init-a.cc

```
const int MAX = 10;
```

```
/* *****
 * init_array_1 -- Zero out an array
 *
 * Parameters
 *   data -- the array to zero
 * *****/
void init_array_1(int data[])
{
    int index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

/* *****
 * init_array_2 -- Zero out an array
 *
 * Parameters
 *   data_ptr -- pointer to array to zero
 * *****/
void init_array_2(int *data_ptr)
{
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}
#
main()
```

```

{
    int array[MAX];

    // One way of initializing the array
    init_array_1(array);

    // Another way of initializing the array
    init_array_1(&array[0]);

    // Similar to the first method but
    // function is different
    init_array_2(array);

    return (0);
}

```

分隔串

假定你有一个这样的串“Last/First”，你想把它分成两个串，一个含有名，另一个含有姓。

示例15.6读入一行，从中删去换行符。然后调用函数strchr，找到斜杠 (/) 的位置。（实际上，函数strchr是一个标准函数。你只是在本例中复制了它，所以你可以看看它是如何工作的。）

这样last_ptr指向姓的开始字符，first_ptr指向斜杠。然后可以用串结束符（NULL或是'\0'）替换斜杠 (/)，把串分出来。现在last_ptr仅指向了名字，而first_ptr指向了一个空串。把first_ptr移向下一个字符，则first_ptr指向姓的开始。

这个过程如图15.6所示。

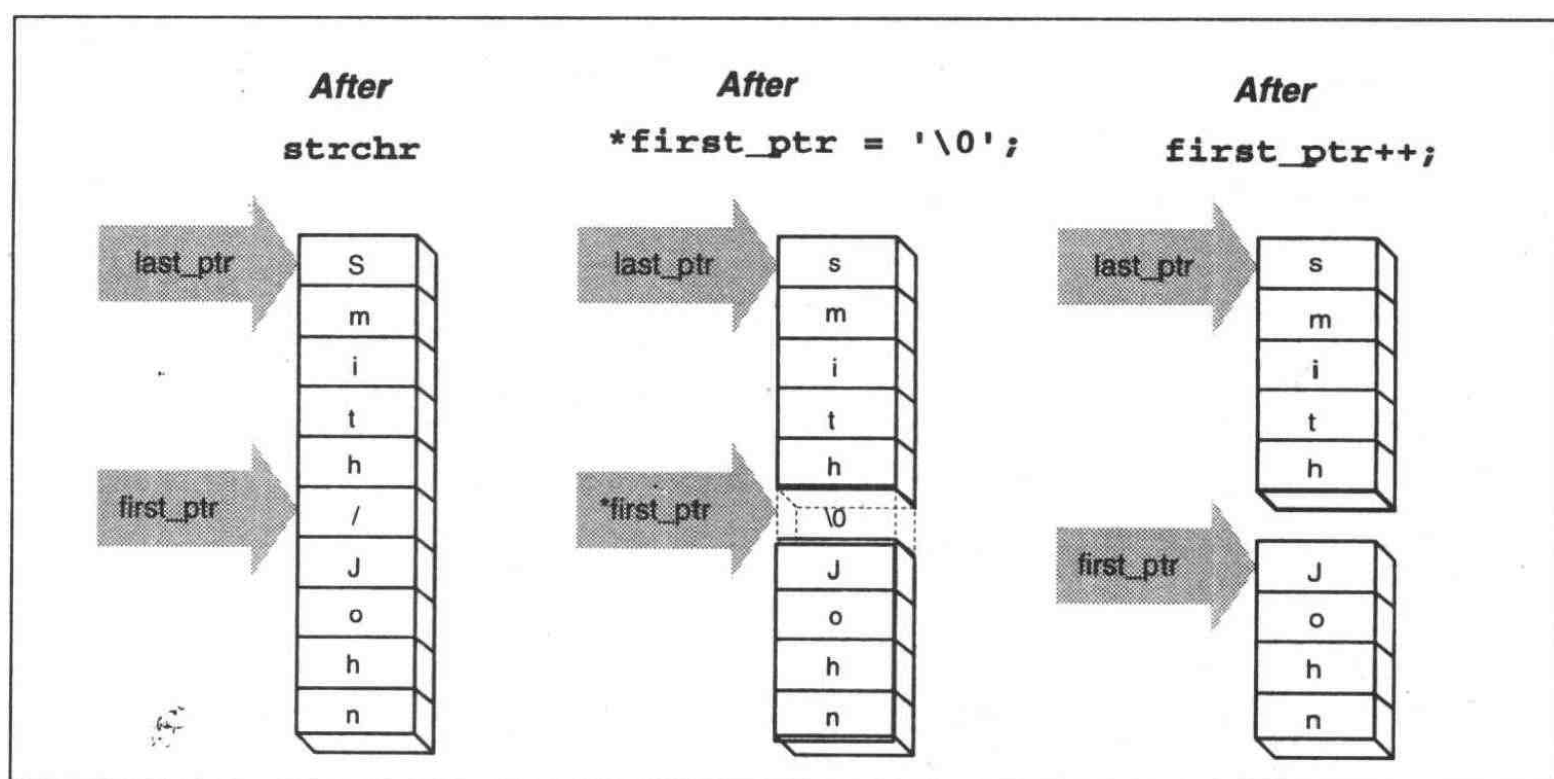


图 15.6 分隔一个串

示例15.6中有程序的全部代码。

示例15.6 *split/split.cc*

```

/*****
 * Split -- split an entry of the form Last/First
 *         into two parts
 *****/
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

main()
{
    char line[80];          // The input line
    char *first_ptr;        // Pointer we set to point to the first name
    char *last_ptr;         // Pointer we set to point to the last name

    cin.getline(line, sizeof(line));

    last_ptr = line;        // Last name is at beginning of line

    first_ptr = strchr(line, '/');    // Find slash

    // Check for an error
    if (first_ptr == NULL) {
        cerr << "Error: Unable to find slash in " << line << '\n';
        exit (8);
    }

    *first_ptr = '\0';      // Zero out the slash

    ++first_ptr;            // Move to first character of name

    cout << "First:" << first_ptr << " Last:" << last_ptr << '\n';
    return (0);
}

/*****
 * strchr -- find a character in a string
 *         Duplicate of a standard library function,
 *         put here for illustrative purposes.
 *
 * Parameters
 *     string_ptr -- string to look through
 *     find -- character to find
 *****/
```

```

    * Returns
    *     pointer to 1st occurrence of character in string*
    *     or NULL for error
    *****/
char *strchr(char * string_ptr, char find)
{
    while (*string_ptr != find) {

        // Check for end

        if (*string_ptr == '\0')
            return (NULL);        // Not found

        ++string_ptr;
    }
    return (string_ptr);        // Found
}

```

这个程序说明了指针和字符数组可用来进行简单的串处理。

问题15.1: 程序15.7应该输出:

Name: tmp1

但是你得到的却是:

Name: !_@#\$ds80

(你的结果可能与此不同。) 为什么会这样呢?

示例15.7 tmp-name/tmp-name.cc

```

#include <iostream.h>
#include <string.h>

/*****
 * tmp_name -- return a temporary filename
 *
 * Each time this function is called, a new name will
 * be returned
 *
 * Returns
 *     pointer to the new filename
 *****/
char *tmp_name(void)
{
    char name[30];                // The name we are generating
    static int sequence = 0;      // Sequence number for last digit

```

```

    ++sequence; // Move to the next filename

    strcpy(name, "tmp");

    // Put in the sequence digit
    name[3] = sequence + '0';

    // End the string
    name[4] = '\0';

    return(name);
}

int main()
{
    cout << "Name: " << tmp_name() << '\n';
    return(0);
}

```

指针和结构

在第12章“高级类型”中，你为邮件表定义了一个结构：

```

struct mailing {
    char name[60];    // Last name, first name
    char address1[60]; // Two lines of street address
    char address2[60];
    char city[40];
    char state[2];    // Two-character abbreviation
    long int zip;     // Numeric zip code
} list[MAX_ENTRIES];

```

邮件表常常按名字和zip码排序。你可以对输入内容进行排序，但是每个输入项有226个字节的内容。所以排序时要把许多的数据移来移去。解决这个问题一个办法是，说明一个指针数组，然后对指针进行排序：

```

// Pointer to the data
struct mailing *list_ptrs[MAX_ENTRIES];

int current;    // Current mailing list entry

// ...

```

```

for (current = 0; current < number_of_entries; ++current)
    list_ptrs = &list[current];

// Sort list_ptrs by zip code

```

现在可以不必移动226个字节的结构，只需移动4个字节的指针。这样，排序的速度更快。假定你有一个装满又重又大的箱子的仓库，你需要快速找到一个箱子的位置。一种办法是，把箱子按字母顺序排列。但是，这要费很大的力气移动箱子。你也可以为每个位置指定一个号码，在索引卡上写下箱子名字和位置号码，然后按名字对卡片进行排序即可。

命令行自变量

实际上，main过程有两个自变量。它们是argc和argv。（它们也不一定非得叫argv和argc，但是99.99%的C++程序都使用这样的名字。）

```

main(int argc, char *argv[])
{

```

当你意识到它们是按字母序排列时，很容易就能记住哪个变量在前面了。

参数argc是命令行中自变量的个数（包括程序名）。数组argv包含实际的自变量。例如，如果用下面的命令行运行程序args：

```
args this is a test
```

那么：

```

argc      = 5
argv[0]   = "args"
argv[1]   = "this"
argv[2]   = "is"
argv[3]   = "a"
argv[4]   = "test"

```

注意

UNIX外壳在把命令行送给程序之前，扩充了通配符，如*、?和[]。详见sh或是csh手册。

如果你的程序连接了WILDARG.OBJ文件的话，Turbo-C++将扩充通配符。详见Turbo-C++手册。

几乎所有的UNIX命令都使用一个标准的命令行格式。这个“标准”也延伸到了其它的操作环境中。一个标准的UNIX命令格式为：

```
命令 选项 文件1 文件2 文件3 ...
```


选项的前面有一个连字符 (-)，而且通常是一个字符。例如，选项-v是打开详细 (verbose) 方式。如果选项带有参数，则参数跟在字母的后面。例如，开关-m1024设置了最大的符号数为1024，-ooutfile设置输出文件名为outfile。

你已经写过规定了文件的格式并输出其内容的程序。这个程序的一部分文档如下：

```
print_file [-v] [-l<length>] [-o<name>] [file1] [file2] ...
```

在这行中，-v设置了verbose选项，它打开程序执行期间的许多信息。选项-l<length>设置每页的大小为<length>行（缺省值为66），而-o<name>设置了输出文件为<name>（缺省值为print.out）。打印的文件列表列在这些选项的后面 ([file1][file2]...)。如果没有指定文件，则输出文件print.in。

while循环将在这些选项之间反复执行。真正的循环是：

```
while ((argc > 1) && (argv[1][0] == '-')) {
```

任何情况下总有一个变量，即程序名。表达式 (argc > 1) 检查额外的自变量。第一个将编号为1。第一个自变量中的第一个字符是argv[1][0]。如果这个字符是一个破折号，则将有一个选项。

循环结尾的代码是：

```
--argc;  
++argv;  
}
```

这就处理完一个自变量。自变量数减1，表示选项少了1个，指向第1个选项的指针前移，选项表向左移1个位置。（注：指针第一次前移后，argv[0]不再指向程序名。）

switch语句用来解释选项的含义。自变量的第0个字符是连字号 (-)。第1个字符是选项字符，所以可以用表达式：

```
switch (argv[1][1]) {
```

对选项进行解码。

选项-v没有参数；它只是设置一个标志。

选项-l有一个整数参数。库函数atoi用来把串转为一个整数。在前一个例子中，你知道argv[1][2]的开头是一个包含了数的串。这个串传给atoi。

选项-o用文件名作为参数。它并没有拷贝整个的串，而是设置字符指针out_file来指向这个串的一部分。此时，有：

```
argv[1][0] = '-'  
argv[1][1] = 'o'  
argv[1][2] = start of the file name
```

用下面的语句把out_file设置为指向串的指针：

```
out_file = &argv[1][2];
```

最后，从语法上分析所有的选项，并转到处理这些选项的循环部分，对每个文件变量执行do_file函数。

示例15.8包含选项解码程序的全部代码。

示例15.8 print/print.cc

```
* Print -- format files for printing *
*****/

#include <iostream.h>
#include <stdlib.h>

int verbose = 0;           // Verbose mode (default = false)
char *out_file = "print.out"; // Output file name
char *program_name;        // Name of the program (for errors)
int line_max = 66;         // Number of lines per page

/*****
 * do_file -- dummy routine to handle a file
 *
 * Parameter
 *   name -- name of the file to print
 *****/
void do_file(char *name)
{
    cout << "Verbose " << verbose << " Lines " << line_max <<
        " Input " << name << " Output " << out_file << '\n';
}

/*****
 * Usage -- tell the user how to use this program and
 *           exit
 *****/
void usage(void)
{
    cerr << "Usage is " << program_name << " [options] [file-list]\n";
    cerr << "Options\n";
    cerr << "  -v          verbose\n";
    cerr << "  -l<number>  Number of lines\n";
    cerr << "  -o<name>    Set output filename\n";
    exit (8);
}

main(int argc, char *argv[])
{
    // Save the program name for future use
```



```

program_name = argv[0];

/*
 * Loop for each option
 * Stop if we run out of arguments
 * or we get an argument without a dash
 */
while ((argc > 1) && (argv[1][0] == '-')) {
    /*
     * argv[1][1] is the actual option character
     */
    switch (argv[1][1]) {
        /*
         * -v verbose
         */
        case 'v':
            verbose = 1;
            break;
        /*
         * -o<name> output file
         * [0] is the dash
         * [1] is the "o"
         * [2] starts the name
         */
        case 'o':
            out_file = &argv[1][2];
            break;
        /*
         * -l<number> set max number of lines
         */
        case 'l':
            line_max = atoi(&argv[1][2]);
            break;
        default:
            cerr << "Bad option " << argv[1] << '\n';
            usage();
    }
    /*
     * Move the argument list up one
     * Move the count down one
     */
    ++argv;
    --argc;
}

```

```

/*
 * At this point all the options have been processed.
 * Check to see if we have no files in the list
 * and, if so, we need to list just standard in
 */
if (argc == 1) {
    do_file("print.in");
} else {
    while (argc > 1) {
        do_file(argv[1]);
        ++argv;
        --argc;
    }
}
return (0);
}

```

这是分析自变量列表的一种方法。使用while循环和switch语句简单易懂。但该方法有一定的局限性。自变量必须紧跟在选项的后面。例如，-odata.out是可行的，但-o data.out将不可行。改进的分析方法将使程序更友好，不过这段代码可以用于简单的程序。（详见系统中有关getopt函数的文档信息。）

编程练习

练习15.1: 编写一个程序，用指针把数组中的每个元素设置为0。

练习15.2: 编写一个函数，它仅有一个串自变量，返回指向串中非空字符的指针。

答疑

问题15.1: 问题出在变量name是个临时变量。进入函数时编译器为name分配空间，退出函数时回收空间。函数赋给name正确的值，并返回一个指向它的指针。但是，函数已经执行完了，所以name消失了，你有一个指针，但它的值是非法的。

解决的办法是把name说明为静态的。因而，它是一个永久变量，在函数结束时也不会消失。

问题15.2: 修改函数后，试着使用两个文件名来执行函数。示例15.9应该输出：

Name: tmp1

Name: tmp2

但它没有。它输出的是什么？为什么？

示例15.9 tmp2/tmp2.cc

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

/*****
 * tmp_name -- return a temporary filename
 *
 * Each time this function is called, a new name will
 * be returned
 *
 * Warning: There should be a warning here, but if we
 * put it in we would answer the question.
 *
 * Returns
 * pointer to the new filename
 *****/
char *tmp_name(void)
{
    static char name[30];      // The name we are generating
    static int sequence = 0;   // Sequence number for last digit

    ++sequence; // Move to the next file name

    strcpy(name, "tmp");

    // Put in the sequence digit
    name[3] = sequence + '0';

    // End the string
    name[4] = '\0';

    return(name);
}

int main()
{
    char *name1;               // Name of a temporary file
    char *name2;               // Name of a temporary file

    name1 = tmp_name();
    name2 = tmp_name();

    cout << "Name1: " << name1 << '\n';
    cout << "Name2: " << name2 << '\n';
    return(0);
}
```

}

回答15.2: 第一次调用`tmp_name`时将返回一个指向`name`的指针。此时只有一个`name`。第二次调用`tmp_name`时修改了`name`，并返回一个指向它的指针。所以有了两个指针，它们指向同一个量：`name`。

有几个库函数返回指向静态串的指针。第二次调用这样的程序时将改变第一次的值。解决的办法是对值进行拷贝：

```
char name1[100];  
char name2[100];  
  
strcpy(name1, tmp_name());  
strcpy(name2, tmp_name());
```

第四部分

高级编程概念

本章内容:

- C++文件I/O
- 转换程序
- 二进制和ASCII码文件
- 行尾难题
- 二进制I/O
- 缓冲问题
- 非缓冲的I/O
- 设计文件格式
- C风格的I/O程序
- C风格的转换程序

第16章

文件输入/输出

一个文件是相关数据的集合。C++把文件看作是一串字节。文件大多在磁盘上；不过，像终端、打印机和磁带这样的设备也被认为是文件。

《Annotated C++ Reference Manual》(Ellis和Stroustrup所著)是目前使用最广泛的C++标准。该书并没有包含I/O系统的规格说明。基于AT&T的cfront编译器所提供的库给出了一个事实标准。但是，具体问题可能会因编译器的不同而有所不同。例如，Turbo-C++在处理cout时把每行的内容都输出来，而SunPro UNIX C++却不这样处理。

目前的ANSI C++标准草案(1994年9月)包含了对I/O调用的详细说明。但是，目前还没有编译器能提供这一特征，而且标准仍在完善之中。

本章讨论三个不同的I/O包。第一个是C++ I/O流类。这是最常用的I/O系统，到目前为止我们一直在用这个类。下一个，我们考查单纯的I/O程序，使用这些程序可以直接进行低级I/O访问。最后，看一看C的I/O系统。虽然这有些过时了，但是C的I/O调用仍然出现在原来的代码中。而且，在某种情况下，C风格的I/O程序要优于C++所提供的。

C++文件I/O

C++文件I/O基于三个类：用于输入的istream类、用于输出的ostream类和用于输入/输出的iostream类。因为C++把文件看作是字节流，所以C++把文件称作是流(stream)。当你启动一个程序时，自动创建四个类变量。这四个类变量列在表16.1中。

表16.1 预定义的I/O类变量

变 量	使 用
cin	控制台输入(标准输入)
cout	控制台输出(标准输出)
cerr	控制台错误(标准错误)
clog	控制台记录

这些变量定义在标准包含文件<iostream.h>中。一般地, cin分配给键盘, 而cout、cerr和clog分配给屏幕。大多数操作系统允许你通过I/O改向而修改它们的分配关系(详见操作系统手册)。

例如, 命令

```
my_prog <file.in
```

执行程序my_prog, 并把cin分配给file.in。

当对磁盘文件进行I/O操作时(除去重定向), 你必须使用流类中用于文件的函数。这些函数是ifstream、ofstream和fstream, 它们定义在包含文件<fstream.h>中。

注意

实际上, ifstream类派生于istream类。同样, ofstream类派生于ostream类, 而fstream类派生于iostream类。我们将在第21章“高级类”中讨论派生类。

假如你想从文件numbers.dat中读100个数。开始先说明输入文件变量:

```
ifstream data_file;    // File we are reading the data from
```

下面, 告诉C++要使用哪个磁盘文件。这通过open成员函数来完成:

```
data_file.open("numbers.dat");
```

现在, 使用读cin时用过的相同的语句来读文件:

```
for (i = 0; i < 100; ++i)
    data_file >> data_array[i];
```

最后, 告诉I/O系统你对文件的操作已经结束了:

```
data_file.close();
```

关闭文件, 释放资源; 过后, 该资源可以再被程序使用。

C++允许在构造函数中使用open调用。例如, 不必写下面的语句:

```
ifstream data_file;    // File we are reading the data from
data_file.open("numbers.dat");
```

可以这样写:

```
ifstream data_file("numbers.dat"); // File we are reading the data from
```

另外, 析构函数自动调用close。

但是, 如果文件numbers.dat不存在怎么办呢? 如果真的没有, 你如何表示出来? 如果这个文件不存在, 成员函数bad返回“true”, 否则返回“false”。所以只需用下面的语句来测试这个文件是否存在:

```

if (data.file.bad()) {
    cerr << "Unable to open numbers.dat\n";
    exit (8);
}

```

改进的读数程序列在示例16.1中。

示例16.1 read/read.cc

```

/*****
 * Read -- read in 100 numbers and sum them
 *
 * Usage:
 *     read
 *
 * Numbers are in the file "numbers.dat"
 *
 * Warning: No check is made for a file with fewer than
 * 100 numbers in it
 *****/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main()
{
    const int DATA_SIZE = 100; // Number of items in the data
    int data_array[DATA_SIZE]; // The data
    ifstream data_file("numbers.dat"); // The input file
    int i; // Loop counter

    if (data_file.bad()) {
        cerr << "Error: Could not open numbers.dat\n";
        exit (8);
    }

    for (i = 0; i < DATA_SIZE; ++i)
        data_file >> data_array[i];

    int total; // Total of the numbers

    total = 0;
    for (i = 0; i < DATA_SIZE; ++i)
        total += data_array[i];

    cout << "Total of all the numbers is " << total << '\n';
}

```

```
    return (0) ;  
}
```

最终，有了getline成员函数。它从输入文件中读入一整行数据。这个函数的定义是：

```
istream &getline(char *buffer, int len, char delim = '\n')
```

这个函数的参数是：

buffer 存储已读入数据的缓冲区。

len 以字节计的缓冲区长度。此函数最多可以把len - 1个字节的数据放入缓冲区中。（一个字节留作结束符\0。）这个参数通常是sizeof(buffer)。

delim 用来表示行尾的字符。

该函数返回输入文件的引用。它读入的数据中包含行尾符号。行尾符号不存储在缓冲区中。

如果指定的数太大了，会出现问题。C++提供一种常用的方法，即使用sizeof运算符，以确保指定参数的大小合适。

sizeof运算符返回其自变量的大小，单位是字节。例如：

```
long int array[10];    // Each element contains 4 bytes  
char string[30];
```

sizeof(string)的结果为30，而sizeof(array)的结果是40（每个长整数的4个字节乘以数组中的10个长整数）。

注意

sizeof不等同于长度。sizeof运算符返回string中的字节数（使用的或是没有使用的）。

输出文件

用于输出文件的函数类似于用于输入文件的。例如，说明：

```
ofstream out_file("out.dat");
```

建立一个名为out.dat的文件，允许你使用文件变量out_file向文件中写入数据。

实际上，构造函数可以带有两个另外的参数。输出文件构造函数的完整定义是：

```
ofstream::ofstream(const char *name, int mode=ios::out,  
                  int prot = filebuf::openprot);
```

这个函数的参数是：

- name** 文件名
- mode** 用或连接在一起的一组标志，可以判定打开方式。标志`ios::out`用于输出文件。其它的标志列在表16.2中。（`ios::`前缀用来表明常量的作用域。该运算符将在第21章“高级类”中详细讨论。）
- prot** 文件保护。这是一个依赖于操作系统的值，它判定文件的保护方式。在UNIX系统中，缺省值为0664（读/写所有者、组读、其它读）。对于MS-DOS/Windows系统，这个值的缺省值为0（一般文件）。

表16.2 打开标志

标 志	含 义
<code>ios::app</code>	在输出文件的最后追加内容。
<code>ios::ate</code>	打开时跳到文件的最后。
<code>ios::in</code>	当作输入文件打开（只能用于 <code>opens</code> for <code>ifstream</code> 变量）。
<code>ios::out</code>	当作输出文件打开（只能用于 <code>ofstream</code> <code>opens</code> ）。
<code>ios::binary</code>	二进制文件（如果不是这种形式的，就作为ASCII文件打开）。有关二进制文件的介绍详见“二进制I/O”一节。
<code>ios::trunc</code>	当打开用于写入的文件时删去文件中的所有内容。
<code>ios::nocreate</code>	如果文件不存在，则失败。（仅用于输出文件。如果文件不存在，输入文件永远失败。）
<code>ios::noreplace</code>	不覆写已存在的文件。如果文件存在，则打开失败。

例如，语句：

```
ofstream out_file("data.new",
                  ios::out|ios::binary|ios::nocreate|ios::app);
```

向一个名为`data.new`的已存在文件（`ios::nocreate`）中添加（`ios::app`）二进制数据（`ios::binary`）。

转换程序

到目前为止，我们仅考虑字符和串的写入。在本节，我们将考虑一些更高级的I/O运算符：转换。

为了向打印机或终端输出一个数，你必须把数转换为字符。打印机只知道字符，不知道数字。例如，数567必须被转换为三个字符“5”、“6”“7”才能打印。

<<运算符把数据转换为字符，然后把它们放到一个文件中。这个函数非常灵活。它可以把一个简单的整数转换为定长的或不定长的串，作为左或右对齐的十六进制、八进制或

十进制的数。到目前为止，你已经在输出中用到了缺省的转换函数。它表现得非常好，但是如果你想精确地控制你的输出，你还需进一步了解转换标志。

成员函数setf和unsetf用于设置和清除控制转换过程的标志。该函数的一般形式是：

文件变量.setf(标志); // Set flags

文件变量.unsetf(标志); // Clear flags

表16.3中列出了所有的标志及其它们的含义。

表16.3 I/O转换标志

标 志	含 义
ios::skipws	跳过输入中前面的空格。
ios::left	输出时左对齐。
ios::right	输出时右对齐。
ios::internal	在符号或基数字符和数本身之间插入一个填充字符，拉长输出的数。
ios::dec	数以基数为10的十进制格式输出。
ios::oct	数以基数为8的八进制格式输出。
ios::hex	数以基数为16的十六进制格式输出。
ios::showbase	在每个数的前面打印基数标志符。例如：在十六进制数前附加“0x”。
ios::showpoint	不论是否需要，在所有浮点数中显示小数点。
ios::uppercase	当转换十六进制数时，数字A~F显示为大写。
ios::showpos	在所有正数前输出+号。
ios::scientific	输出时所有浮点数以科学计数法表示。
ios::fixed	输出时把所有浮点数转换为定点数。
ios::unitbuf	缓冲区输出。（后面详细介绍。）
ios::stdio	每一次输出后刷新流。

如果你想以十六进制形式输出一个数，只需写：

```
number = 0x3FF;
cout << "Dec: " << number << '\n';
cout.setf(ios::hex);
cout << "Hex: " << number << '\n';
cout.setf(ios::dec);
```

执行时，该程序段的输出如下：

```
Dex 1023
Hex 3ff
```

注意

人们通常希望以十进制形式输出，所以在每次输出之后重新设置输出方式，以避免混淆确实是一个好主意。

当把数转换为字符时，成员函数：

```
int 文件变量.width(int size);
```

决定需占用的最少字符数。例如，数3将转换为字符串“3”（注意没有空格）。如果宽度设置为4，则结果为“ 3”，此处有3个空格。

成员函数：

```
int 文件变量.precision(int digits);
```

控制在小数点后打印多少位数字。

最后，函数：

```
char 文件变量.fill(char pad);
```

判定文件字符。当数的位数小于指定的宽度时，用该字符来填充。

注意

在这些标志和参数中，有些要在每次输出调用后重新设置，而有些则不需要。哪些是永久设置，哪些是临时设置，这要依编译器而定。总的来说，不要假定所有的设置都将被保留。（只因为编译器不以你的意志为转移。）

可以直接调用这些函数，或者可以使用I/O操纵函数。I/O操纵函数是一个特殊的函数，它可以用在I/O语句中来改变格式。你可以把操纵函数看作是一粒魔术弹，当把它射入输入或输出文件中时，可以改变文件的状态。操纵函数不会导致任何的输出；它仅仅是改变了文件的状态。例如，操纵函数hex把输出改为十六进制形式。

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

```
#include <iomanip.h>
```

```
number = 0x3FF;
```

```
cout << "Number is " << hex << number << dec << '\n';
```

在包含文件<iomanip.h>中定义了I/O操纵函数。表16.4列出了所有的I/O操纵函数。

表16.4 I/O操纵函数

操纵函数	描 述
<code>setiosflags(long flags)</code>	设置所选的转换标志。
<code>resetiosflags(long flags)</code>	重新设置所选的标志。
<code>dec</code>	数以十进制格式输出。
<code>hex</code>	数以十六进制格式输出。
<code>oct</code>	数以八进制格式输出。
<code>setbase(int base)</code>	把转换基数设置为8、10或16，有点类似于广义的 <code>dec</code> 、 <code>hex</code> 、 <code>oct</code> 。
<code>setw(int width)</code>	设置输出的宽度。
<code>setprecision(int precision)</code>	设置浮点数输出时的精度。
<code>setfill(char ch)</code>	设置填充字符。
<code>ws</code>	跳过输入中的空格。
<code>endl</code>	输出行结束符。
<code>ends</code>	输出串结束符（'\0'）。
<code>flush</code>	强制输出缓冲区中的所有内容。（该函数的使用方法见第17章“调试和优化”）。

示例16.2显示一些I/O操纵函数的使用方法。

示例16.2 io/io.cc

```
#include <iostream.h>
#include <iomanip.h>
main()
#
{
    int number = 12;    // A number to output
    float real = 12.34; // A real number

    cout << "123456789012345678901234567890\n"; // Output ruler
    cout << number << "<-\n";
    cout << setw(5) << number << "<-\n";
    cout << setw(5) << setfill('*') << number << "<-\n";
    cout << setiosflags(ios::showpos|ios::left) << setw(5) <<
        number << "<-\n";

    cout << real << "<-\n";
    cout << setprecision(1) << setiosflags(ios::fixed) << real << "<-\n";
    cout << setiosflags(ios::scientific) << real << "<-\n";
    return (0);
}
```



```
}
```

该程序的输出为:

```
123456789012345678901234567890
12<-
    12<-
***12<-
+12**<-
12.34<-
12.3<-
1e+01<-
```

二进制和ASCII码文件

到目前为止, 我们只接触了ASCII码文件。“ASCII”代表American Standard Code for Information Interchange(美国信息交换标准代码)。它是一个集合, 有95个可打印字符和33个控制码。(完整的ASCII码表列在附录A中。)我们能够读懂ASCII码文件。当你写程序时, prog.cc文件就是ASCII码文件。

终端、键盘和打印机处理字符数据。当你在屏幕上写一个像1234这样的数时, 它必须被转换为4个字符(1、2、3、4)来写。

同样地, 当你从键盘上读一个数时, 数据必须从字符转换为整数。这由>>运算符来完成。

ASCII字符“0”的值是48, “1”的值是49, 依此类推。当你想把一个数字从ASCII码转换为整数时, 必须减去48这个值:

```
int integer;
char ch;

ch = '5';
integer = ch - 48;
cout << "integer " << integer < '\n';
```

不用记着“0”的值是48, 你可以让它减“0”:

```
integer = ch - '0';
```

计算机使用的是二进制数。当从ASCII码文件中读数时, 程序必须通过一个转换程序进行字符数据的处理, 这些转换程序很像我们刚刚定义的整数转换程序。这需要额外做一些工作。二进制文件不需要转换。而且一般情况下, 它们还比ASCII码文件占用更少的空间。缺点是, 它们不能直接输出到终端或是打印机上。(如果你看见打印机输出一段很长的内容, 内容的开始很像是“!E#(@@^^JHC%^X”, 那么你应该意识到你打印了一个二进制文

件。)

ASCII码文件是可以移植的(对大多数情况而言)。它们可以不费什么周折地从一台机器移到另一台机器。几乎所有的二进制文件肯定不能移植。除非你是一位编程专家,否则没有办法移植一个二进制文件。(详见第25章“移植问题”。)

你应该使用哪种文件类型呢?大多数情况下,ASCII码文件最好。如果你有少量的数据,转换时间不足以影响到程序的性能。(谁会在乎是花费了0.5秒还是0.3秒呢?)使用ASCII码文件时,数据的修改也简单。

只有当你的数据量非常之大,足以影响到所占空间及程序性能的时候,才使用二进制格式文件。

行尾难题

在没有计算机时,有一种名叫Teletype Model 33的魔术般的设备。这个令人吃惊的机器包含一个由马达和转子制成的移位寄存器,还有一个仅由杆杠和弹簧组成的键盘ROM。

电传打字电报机有一个键盘、一台打印机和一台纸带阅读/打孔机。它可以在电话线上通过调制解调器传输信息,速度是每秒钟10个字符。

但是电传打字电报机有一个问题。它要用2/10秒的时间把打印头从右侧移到左侧。2/10秒的时间可以传输2个字符。如果在打印头移动的过程中,又来了一个字符,则这个字符将丢失。

电传打字电报机的研制人员通过在行尾加两个符号解决了这个问题。这两个符号是:<回车>把打印头定位在左边界,<换行>把纸向上移一行。在打印头回到左边界的时候,“打印”出<换行>。

计算机发展的早期,一些设计人员认为在每一行的结尾使用两个字符太浪费存储空间(那个时候,存储器很贵)。有的只在行尾放一个<换行>,有的选择了<回车>。有些固守陈规的仍使用两个连续符号。

UNIX在行尾使用<换行>。新行符号\n的代码是0xA(LF或是<换行>)。

MS-DOS/Windows使用了两个符号<换行><回车>。编译器设计人员在处理原有的C程序时,会考虑新行是否就是<换行>。解决的办法是给I/O库增加代码,从ASCII输入文件中删去<回车>符,并在输出时把<换行>改为<换行><回车>。

在MS-DOS/Windows中,要特别注意文件是以ASCII码格式打开,还是以二进制格式打开。标志ios::binary用来标识一个二进制文件:

```
// Open ASCII file for reading
ascii_file.open("name", ios::in);

// Open binary file for reading
```

```
binary_file.open("name", ios::in|ios::binary);
```

问题16.1: 成员函数put可以往二进制文件中写入一个字节。下面的程序把数0到127写入文件test.out中。它在UNIX下工作得很好，建立了一个128个字节的文件；但在MS-DOS/Windows下，文件含有129个字节。为什么？

示例16.3 wbin/wbin.cc

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

main()
{
    int cur_char;    // Current character to write
    ofstream out_file; // Output file

    out_file.open("test.out", ios::out);
    if (out_file.bad()) {
        (cerr << "Cannot open output file\n");
        exit (8);
    }

    for (cur_char = 0; cur_char < 128; ++cur_char) {

        out_file << cur_char;
    }
    return (0);
}
```

提示: 下面是MS-DOS/Windows文件按十六进制格式的输出内容:

```
000:0001 0203 0405 0607 0809 0d0a 0b0c 0d0e
010:0f10 1112 1314 1516 1718 191a 1b1c 1d1e
020:1f20 2122 2324 2526 2728 292a 2b2c 2d2e
030:2f30 3132 3334 3536 3738 393a 3b3c 3d3e
040:3f40 4142 4344 4546 4748 494a 4b4c 4d4e
050:4f50 5152 5354 5556 5758 595a 5b5c 5d5e
060:5f60 6162 6364 6566 6768 696a 6b6c 6d6e
070:6f70 7172 7374 7576 7778 797a 7b7c 7d7e
080:7f
```

C++库函数自动整理ASCII码文件，对这一点UNIX程序员不必过分担心。在UNIX中，一个文件就是一个文件，ASCII码文件与二进制文件没有区别。事实上，如果你愿意的话，你可以写一个半是ASCII码半是二进制码的文件。

二进制I/O

有两个成员函数：`read`和`write`来完成二进制I/O。`read`的语法为：

```
in_file.read(data_ptr, size);
```

其中，

`data_ptr`指向放数据地方的指针。

`size`是要读入的字节数。

成员函数`gcount`返回最近一次`read`函数得到的字节数。这个数可能比所需要的字节数少。

例如，`read`可能会遇到一个文件尾符或是错误：

```
struct {
    int    width;
    int    height;
} rectangle;

in_file.read((char *)(&rectangle), sizeof(rectangle));
if (in_file.bad()) {
    cerr << "Unable to read rectangle\n";
    exit (8);
}
if (in_file.gcount() != sizeof(rectangle)) {
    cerr << "Error: Unable to read full rectangle\n";
    cerr <  "I/O error of EOF encountered\n";
}
```

在这个例子中，你读入结构`rectangle`。`&`运算符把`rectangle`变为一个指针。“`(char *)`”转换是必需的，因为`read`需要一个字符数组。`sizeof`运算符用来判定要读入多少个字节，并检查`read`是否成功地读完。

成员函数`write`的调用类似于`read`函数的调用：

```
out_file.write(data_ptr, size);
```

缓冲问题

缓冲I/O不是立即把数据写入文件。相反，数据保留在一个缓冲区中，直到缓冲区中的数据量足够一次写入，或是遇到一次清缓冲操作为止。下面的程序设计为在每完成一个阶段后就输出一条信息。

```
cout << "Starting";
do_step_1();
```

```
cout << "Step 1 complete";
do_step_2();
cout << "Step 2 complete";
do_step_3();
cout << "Step 3 complete\n";
```

cout并没在每步执行完立即输出一条信息，而是把它们输入到缓冲区中。只当程序执行完毕才能清缓冲区，同时所有的信息立即全部输出。

I/O操作符flush强制输出缓冲区的内容。上面的例子正确的写法是：

```
cout << "Starting" << flush;
do_step_1();
cout << "Step 1 complete" << flush;
do_step_2();
cout << "Step 2 complete" << flush;
do_step_3();
cout << "Step 3 complete" << flush;
```

非缓冲I/O

在缓冲I/O中，数据先缓存，然后再送到文件中；在非缓冲I/O中，数据立即送到文件中。

如果你把很多的纸夹掉到了地板上，你可以以缓冲或非缓冲的方式拾起它们。在缓冲方式下，你用右手拾起一个夹子，然后放到左手上。重复这样做，直到左手满了，然后把手上的纸夹子放到桌上的盒子里。在非缓冲方式下，你每拾一个夹子，就把它放到盒子里。不需要先放到左手中。

大多数情况下，缓冲I/O比非缓冲I/O更有用。在非缓冲I/O中，每一次的读写操作需要一次系统调用。对操作系统的任何调用都有消耗。缓冲I/O使得这些调用次数最少。

仅当读写大量的二进制数据，或是需要直接控制一个设备或文件时，才应该使用非缓冲I/O。再回到纸夹例子中——如果你拾的是小的物体，就像纸夹这样的，你可能用左手当缓冲区。但是如果你拾的是炮弹（它很大），就不需要用缓冲区。

系统调用open用来打开一个非缓冲文件。该调用的宏定义因系统不同而有所不同。因为你用的是UNIX和MS-DOS/Windows，所以要用条件编译（#ifdef/#endif）才能得到正确的文件。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#ifdef __MSDOS__          // If we are MS-DOS
```

```
#include <io.h>           // Get the MS-DOS include file for raw I/O
#else /* __MSDOS__ */
#include <unistd.h>       // Get the UNIX include file for raw I/O
#endif /* __MSDOS__*/
```

```
int    file_descriptor;
```

```
file_descriptor = open (名字, 标志);      // Existing file
file_descriptor = open (名字, 标志, 方式); //New file
```

file_descriptor 标识用于读、写和关闭所调用文件的一个整数。如果**file_descriptor**小于0，则发生错误。

名字 文件的名字。

标志 定义在头文件fcntl.h中。打开标志列在表16.5中。

表 16.5 打开标志

标 志	意 义
O_RDONLY	打开只读文件
O_WRONLY	打开只写文件
O_RDWR	打开读写文件
O_APPEND	在文件尾添加新数据
O_CREAT	创建文件（当指定了该标志后，还要指定方式）
O_TRUNC	如果文件已存在，则把它的长度截为0
O_EXCL	如果文件已存在，则失败
O_BINARY	以二进制方式打开文件（原来的UNIX系统可能没有此标志）

方式 该文件的保护方式。一般地，对大多数文件来讲，该值是0666。

例如，下面的语句以文本方式打开一个用于输入的已存在文件data.txt:

```
data_fd = open("data.txt", O_RDONLY);
```

下面的例子显示如何创建一个用于输出的文件output.dat:

```
out_fd = open("output.dat", O_CREAT|O_WRONLY, 0666);
```

注意，你可以用OR (|) 运算符把标志连接起来。这是合并多个标志的快速简单的方法。

当程序第一次运行时，系统已经打开了三个文件。这三个文件列在表16.6中。



表 16.6 标准非缓冲文件

文件号	描 述
0	标准输入
1	标准输出
2	标准错误

`read` 调用的格式为:

读字节数 = `read` (文件描述符, 缓冲区, 大小);

读字节数 读取的真正字节数。0表明文件尾, 负数表明一个错误。

文件描述符 一个打开文件的文件描述符。

缓冲区 指向要读数据的指针。

大小 要读入的数据大小。这是申请的大小。读取的真正字节数可能比它小。
(例如, 你的数据可能读完了。)

`write`调用的格式为:

写字节数 = `write`(文件描述符, 缓冲区, 大小);

写字节数 写入的真正字节数。负数表明一个错误。

文件描述符 一个打开文件的文件描述符。

缓冲区 指向数据要写入地方的指针。

大小 要写入的数据大小。系统将按此字节数写入数据, 但是如果设备满了或是有其它的问题, 写入的数据字节数要小于该数。

最后, 关闭文件的`close`调用为:

标志 = `close`(文件描述符);

标志 0表示成功, 负数表示错误。

文件描述符 一个打开文件的文件描述符。

示例16.4拷贝一个文件。因为缓冲区较大, 所以使用了非缓冲I/O。没有理由用缓冲I/O把1K的数据读入一个缓冲区中(使用`ifstream`), 然后再把它送到16K的缓冲区中。

示例16.4 `copy2/copy2.cc`

```

/*****
 * Copy -- copy one file to another
 *
 * Usage
 *      copy <from> <to>
 *
 * <from> -- the file to copy from
 *****/

```



```

* <to>    -- the file to copy into      *
*****/
#include <iostream.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#ifdef __MSDOS__          // If we are MS-DOS
#include <io.h>           // Get the MS-DOS include file for raw I/O
#else /* __MSDOS__ */
#include <unistd.h>       // Get the UNIX include file for raw I/O
#endif /* __MSDOS__ */

const int BUFFER_SIZE = (16 * 1024); // Use 16K buffers

main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE]; // Buffer for data
    int in_file;              // Input file descriptor
    int out_file;             // Output file descriptor
    int read_size;            // Number of bytes on last read

    if (argc != 3) {
        cerr << "Error: Wrong number of arguments\n";
        cerr << "Usage is: copy <from> <to>\n";
        exit(8);
    }

    in_file = open(argv[1], O_RDONLY);
    if (in_file < 0) {
        cerr << "Error: Unable to open " << argv[1] << '\n';
        exit(8);
    }

    out_file = open(argv[2], O_WRONLY | O_TRUNC | O_CREAT, 0666);
    if (out_file < 0) {
        cerr << "Error: Unable to open " << argv[2] << '\n';
        exit(8);
    }

    while (1) {
        read_size = read(in_file, buffer, sizeof(buffer));

        if (read_size == 0)
            break;                // End of file

        if (read_size < 0) {

```

```

        cerr << "Error: Read error\n";
        exit(8);
    }
    write(out_file, buffer, (unsigned int) read_size);
}
close(in_file);
close(out_file);
return (0);
}

```

在这个程序中，要注意几个问题。首先，缓冲区大小定义为一个常量，所以很容易修改。程序员不用死记16K就是16384，只需使用表达式(16 * 1024)即可。这样更明确地表示这是16K。

如果用户使用程序的方法不正确，会得到一条错误信息。为了帮助用户能够正确地使用它，信息中包含了使用程序的正确方法。

最后一次读数可能读不满整个缓冲区。这就是使用read_size的原因，为的是判定要写人的字节数。

设计文件格式

假定你正在设计一个绘图程序。在图形配置文件中定义了高、宽、范围和缩放比例。还要求你写一个用户友好的程序，该程序询问操作人员有关的问题，并据此写一个配置文件。这样，他或是她就不必学习文本编辑器的使用了。你如何设计一个配置文件？

一种方法是：

高度（单位：英寸）

宽度（单位：英寸）

x下限

x上限

y下限

y上限

x缩放比例

y缩放比例

典型的配置文件可能如下所示：

```

10.0
7.0
0
100

```



```
30
300
0.5
2.0
```

该文件包含了所有的数据。但在读它的时候，你不能具体地识别出它们，例如，你不知道哪个值是Y的下限。解决的办法是给文件加注释，这样，配置程序不仅写出数据，而且还写出描述数据的字符串。

```
10.0      height (in inches)
7.0       width (in inches)
0         x lower limit
100       x upper limit
30        y lower limit
300       y upper limit
0.5       x-scale
2.0       y-scale
```

现在我们可以读这个文件了。但是假设一个用户执行这个绘图程序，并键入了错误的文件名，则程序得到的是今天中午的午餐菜谱，而不是绘图配置文件。当程序试图建立一个图形时可能会弄得非常混乱，因为图形的尺寸不是“BLT on white”，而是“Meatloaf and gravy”。

结果会一塌糊涂。应该有办法识别出一个绘图配置文件。一种办法是在文件的第一行写上“Plot Configuration File”。然后，当某人给程序输入错误的文件名时，程序将输出报错信息。

这样做会解决文件错误问题，但是当要求你增强程序的功能，绘制额外的对数图时怎么办？简单的办法就是在配置文件中添加额外的行，但是所有的旧文件怎么办呢？没有理由让每个人都把它们扔掉。最好的办法是（从用户的角度看）认可原有的文件格式。你可以在文件中加入版本号，这样就简单了。

现在，典型的文件如下：

```
Plot Configuration File V1.0
log      Logarithmic or normal plot
10.0     height (in inches)
7.0      width (in inches)
0        x lower limit
100      x upper limit
30       y lower limit
300      y upper limit
0.5      x-scale
2.0      y-scale
```

在二进制文件中，在文件的前4个字节放入一个标识号是常用的办法。这个号称为魔术

号(*magic number*)。对每一类文件, 魔术号应该有所不同。

选择魔术号的一种方法是取程序的前4个字母开头(例如, list), 把它们转换为十六进制数: 0x6c607374。然后再加上0x80808080作为魔术号: 0xECE0F3F4。

这样产生的魔术号可能是唯一的。每个字节的高位设置为1, 以保证这些字节不在ASCII码范围内, 从而避免在ASCII码文件和二进制文件中引起混淆。

当从包含许多不同类型结构的二进制文件中进行读写操作时, 很容易弄错。例如, 当你想要一个尺寸结构的时候, 可能读入了一个名字结构。通常不能马上发现这个错误, 要到程序执行的后面才能发现。为了尽早发现这个错误, 程序员可以在每个结构的开头加入魔术号。这样, 如果程序读入名字结构并且魔术号不对时, 它就知道出错了。结构的魔术号不需要把每个字节的高位置为1。4个ASCII字符做魔术号使得从文件中抽取结构的开始信息更简单了。

C风格的I/O程序

C++允许你在C++程序中使用C的I/O库。这种情况经常出现, 因为有些程序员把C程序转换为C++程序, 但不想转换I/O调用。在有些情况下, 旧的C库比新的C++库更好也更易用。例如, C中像sscanf和sprintf这样的字符串转换程序比C++中类似的函数要简单得多。C中I/O函数所用的结构和函数的说明存储在标准包含文件<stdio.h>中。

文件变量的说明如下:

```
File *文件变量; /* Comment */
```

例如:

```
#include <stdio.h>
File *in_file; /* File containing the input data */
```

在文件使用前, 必须先使用fopen函数打开。fopen返回一个指向该文件对应的文件结构的指针。fopen的格式为:

文件变量 = fopen(名字, 方式);

文件变量 一个文件变量。

名字 文件的真正名字 (data.txt, temp.dat, 等等)。

方式 表明文件是用于读还是用于写。方式为“w”时用于写, 为“r”时用于读。

fclose函数关闭文件。fclose的格式为:

状态 = fclose(文件变量);

如果fclose成功, 则状态变量为0, 否则为非0。

C提供了三种预打开文件。这些文件列在表16.7中。

表16.7 标准文件

文 件	描 述
stdin	标准输入（打开用于读）。等价于C++的cin。
stdout	标准输出（打开用于写）。等价于C++的cout。
stderr	标准错误（打开用于写）。等价于C++的cerr。
	C中没有文件等价于C++的clog。

函数fgetc从文件中读入单个字符。如果文件中没有数据，函数将返回常量EOF（EOF定义在stdio.h中）。注意，fgetc返回一个整数，而不是一个字符。这是必需的，因为EOF标志必须是一个非字符的值。

示例16.5统计文件input.txt中的字符数。

```

示例16.5 copy/copy.cc
#include <stdio.h>
#include <stdlib.h>      /* ANSI Standard C file */
#include <iostream.h>

const char FILE_NAME[] = "input.txt";    // Name of the input file

main()
{
    int count = 0;    // Number of characters seen
    FILE *in_file;    // Input file

    int ch;           // Character or EOF flag from input

    in_file = fopen(FILE_NAME, "rb");
    if (in_file == NULL) {
        cerr << "Cannot open " << FILE_NAME << '\n';
        exit(8);
    }

    while (1) {
        ch = fgetc(in_file);
        if (ch == EOF)
            break;
        ++count;
    }

    cout << "Number of characters in " << FILE_NAME << " is " << count << '\n';
}

```

```

    fclose(in_file);
    return (0);
}

```

类似的函数fputc用于写一个字符。它的格式为:

fputc(字符, 文件);

函数fgets和fputs一次操作一行。fgets调用的格式为:

串指针 = fgets(串, 长度, 文件);

串指针 如果读操作正确, 它等于串; 如果读入的是EOF或是遇到一个错误, 则它为NULL。

串 函数存放字符串的字符数组。

长度 字符数组的大小。fgets读取一行(行尾是\n), 或是读取size - 1个字符。然后在字符串的末尾加上结束符(\0)。

例如:

```

char string[100];
...
fgets(string, sizeof(string), in_file);

```

fputs类似于fgets, 只是它写入一个串, 而不是读取它。fputs函数的格式为:

串指针 = fputs(串, 文件);

fputs的参数类似于fgets的参数。fputs不需要长度参数, 因为它从串的长度可得知要写出的行的大小。(它一直写, 直到遇到一个空字符\0时为止)。

C风格的转换程序

C++的输出使用<<运算符。C使用printf系列函数。printf调用包含两部分: 描述如何输出数据的格式和要输出的一系列数据。

printf调用的一般格式为:

printf(格式, 参数-1, 参数-2,...);

格式串要原样照印。例如:

printf("Hello World\n");

将输出:

Hello World

要打印一个数, 必须在格式串中放一个%转换。例如, 当C在格式串中见到%d时, 它从参数表中取出下一个参数(它必须是个整数), 并输出它。图16.1表示printf语句的元素

如何产生最后结果，

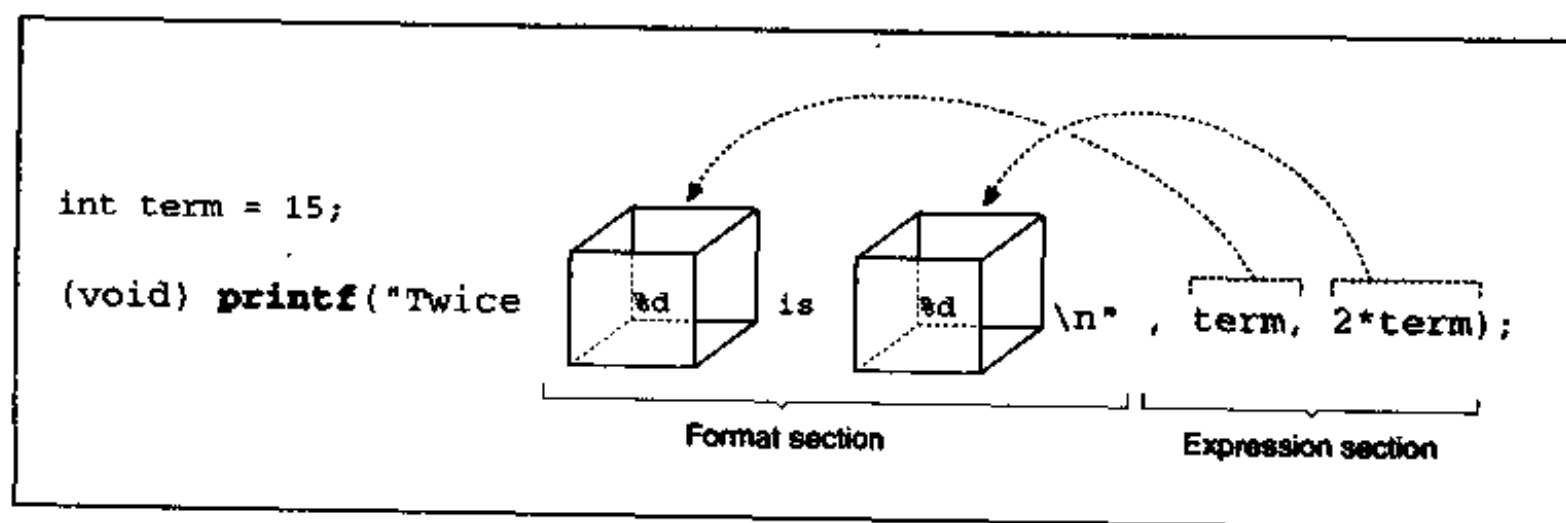


图 16.1 printf结构

%d转换用于整数。其它类型的参数使用不同的转换。例如，如果你想输出一个浮点数，你要用%f转换。表16.8中列出了这些转换。

表16.8 C风格的转换

转 换	变量类型
%d	整型
%ld	长整型
%d	短整型
%f	浮点
%lf	双精度
%u	无符号整型
%lu	无符号长整型
%u	无符号短整型
%s	字符串
%c	字符
%o	整型（输出八进制数）
%x	整型（输出十六进制数）
%e	浮点（以d.dddE+dd格式输出）

在printf语句中还使用了许多其它的转换，详见参考手册。

printf函数不检查每一行参数的个数是否正确。如果你的参数太多了，多余的参数将被忽略。如果你的参数太少了，C将补足缺少的参数。而且C也不检查参数的类型，所以如果你把%d用于浮点数，会得到奇怪的结果。

问题16.2：为什么 $2 + 2 = 5986$ ？（你的答案可能多种多样。）

```

[File: two/two.c]
#include <stdio.h>
main()
{
    int answer;

    answer = 2 + 2;
    printf("The answer is %d\n");
    return (0);
}

```

问题16.3: 为什么 $21 / 7 = 0$? (你的答案可能多种多样。)

```

[File: float3/float3.c]
#include <stdio.h>

main()
{
    float result;

    result = 21.0 / 7.0;
    printf("The result is %d\n", result);
    return (0);
}

```

函数 `fprintf` 类似于 `printf`, 只是它多带有一个参数, 即要输入到的文件。

`fprintf(文件, 格式, 参数-1, 参数-2,...);`

`printf` 系列函数中还有 `sprintf` 调用。`sprintf` 的第一个参数是一个串。该函数按格式输出, 并把结果写入给定的串中。

`sprintf(串, 格式, 参数-1, 参数-2, ...);`

例如:

```

char string[40];    /* The filename */

/* Current file number for this segment */
int file_number = 0;

sprintf(string, "file.%d", file_number);
++file_number;
out_file = fopen(string, "w");

```


`sprintf`的返回值因系统不同而有所不同。ANSI标准把它定义为存储在串中的字符数；但是有些UNIX系统把它定义为指向串的指针。

通过调用`scanf`系列函数可以完成读操作。`scanf`函数类似于其姊妹函数：`fscanf`和`sscanf`。`fscanf`函数的格式为：

数目 = `fscanf`(文件, 格式, &参数-1,...);

数目 转换成功的参数个数

文件 打开的用于读入的文件

格式 描述要读入的数据格式

参数-1 要读入的第一个参数

如果你忘记在`scanf`的每个变量前放`&`号，可能会发生“Segmentation violation core dumped”或是“Illegal memory access”错误。有时，还会改变一个变量或是指令。这个错误在UNIX系统中不多见，但在MS-DOS/Windows系统中，因为它缺少内存保护机制，所以不容易发现这个错误。在MS-DOS/Windows系统中，缺少`&`会导致系统瘫痪。

使用`scanf`时也有问题：它几乎不可能正确地处理行尾符。但是，有一个简单的办法可以避开`scanf`的这个局限——不用它。我们使用`fgets`函数，再使用`scanf`用于串的变体函数`sscanf`：

```
char line[100];    // Line for data

fgets(line, sizeof(line), stdin);    // Read numbers
sscanf(line, "%d %d", &number1, &number2);
```

最后，讨论`scanf`用于文件的变体`fscanf`。这个函数的使用非常困难，不要用它。使用`fgets`和`sscanf`来代替它。

C风格的二进制I/O

通过两个函数：`fread`和`fwrite`可以完成二进制的I/O操作。`fread`的语法是：

读入长度 = `fread`(数据指针, 1, 长度, 文件);

读入长度 读入数据的长度。如果该数小于长度，则是遇到了文件尾或是发生了错误。

数据指针 指向要读入数据的指针。

长度 要读入的数据长度。

文件 输入文件。

例:

```
struct {
    int width;
    int height;
} rectangle;
if (fread((char *)&rectangle, 1, sizeof(rectangle), in_file) !=
    sizeof(rectangle)) {
    fprintf(stderr, "Unable to read rectangle\n");
    exit (8);
}
```

在这个例子中, 读入一个结构rectangle。&运算符把结构转为一个指针。转换“(char *)”把&rectangle转换为合适的参数类型, sizeof运算符用来判定要读入多少个字节, 并检查读操作是否正确。

fwrite的调用序列与fread类似:

写大小 = fwrite(数据指针, 1, 长度, 文件);

注意

为了保证程序设计得简单容易, 我总是用1来作fread和fwrite函数中的第二个参数。这些函数的详细说明请参见C语言参考手册。

问题16.4: 不管你在下面的程序中输入什么文件名, 程序都找不到它。为什么?

```
[File: fun-file/fun-file.c]
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char        name[100]; /* Name of the file to use */
    FILE        *in_file;  /* File for input */

    printf("Name? ");
    fgets(name, sizeof(name), stdin);

    in_file = fopen(name, "r");
    if (in_file == NULL) {
        (void) fprintf(stderr, "Could not open file\n");
    }
}
```



```

        exit(8);
    }
    printf("File found\n");
    fclose(in_file);
    return (0);
}

```

编程练习

练习16.1: 编写一个程序，它读入一个文件，并统计文件中的行数。

练习16.2: 编写一个程序，它拷贝一个文件，并将其中的水平制表符换为多个空格。

练习16.3: 编写一个程序，它读入一个含有一串数的文件，把其中能被3整除的数写入一个文件中，其余的数写入另一个文件中。

练习16.4: 编写一个程序，它读入一个含有一串数的ASCII码文件，并把同样的数写入一个二进制文件中。编写一个程序，它执行与上述相反的操作，以检验前一个程序的正确性。

练习16.5: 编写一个程序，它拷贝一个文件，并删除其中高位为1的所有字符（`((ch & 0x80) != 0)`）。

练习16.6: 设计一种文件格式，它存储人名、地址和其它信息。编写一个程序读入这个文件，并产生一组邮件表。

答疑

问题16.1: 问题出在，你在写一个ASCII码文件，但你想要二进制文件。在UNIX系统中，ASCII码文件等同于二进制文件，所以程序运行良好。在MS-DOS/Windows系统中，行尾的出现会导致错误。当你向文件中写一个换行符号（0x0a）时，同时要向文件中加入一个回车符（0x0D）。（记住，在MS-DOS/Windows中行尾是<回车><换行>，或0x0d、0x0a。）因为这种编辑方式，所以在输出文件中得到了一个多余的回车符（0x0d）。

为了写入二进制数据（不进行输出编辑），必须用二进制选项打开文件：

```
out_file = fopen("test.out", "wb");
```

问题16.2: printf调用不检查参数的个数。语句：

```
printf("The answer is %d\n");
```

告诉printf打印出串“The answer is”，并且接着输出结果。问题是，含有结果的参数被省略了。这时，printf从一个随机的地址取出数，并输出出来。

printf语句的正确写法是：

```
printf("The answer is %d\n", answer);
```

问题16.3: printf调用不检查参数的类型。你告诉printf调用输出一个整数（%d），并提供了一个浮点参数（result）。这种不匹配会导致意料之外的结果，如打印出错误的答案。

当输出浮点数时，需要使用%f转换。printf语句的正确写法是：

```
printf("The answer is %f\n", result);
```

问题16.4: 问题出在fgets得到了整个的行，其中包括换行符（\n）。如果你有一个名为sam的文件，程序读入sam\n，并查找这个名字的文件。因为没有这样的文件，所以程序报告出错。

要做的修改就是去掉名字中的换行符：

```
name[strlen(name) - 1] = '\0'; /* Get rid of last character */
```

这样的出错信息设计得不好。事实上，你没有打开文件，但是程序员可以为用户提供更多的信息。你想打开文件用来读或写吗？你想打开的文件名是什么？你甚至不知道你得到的信息是一条错误信息、还是一条警告信息或仅仅是正常操作的一部分。好的报错信息是：

```
fprintf(stderr, "Error: Unable to open %s for input\n", name);
```

注意，这条信息还可以帮助我们检查编程错误。当你键入“sam”时，应该有报错信息：

```
Error: Unable to open sam
for input
```

它清楚地告诉我们，你试图打开一个其名字中含有换行符的文件。

本章内容:

- 调试
- 分而治之
- 只用于调试的代码
- 调试命令行开关
- 交互调试器
- 调试一个二分查找程序
- 运行时错误
- 公开声明调试方法
- 如何优化
- 实例研究
- 编程练习
- 答疑

第17章

调试和 优化

调 试

程序最难的部分不是设计和书写，而是调试阶段。在这个阶段中，你要查明程序为何能运转起来（而不是你想当然的运转方式）。

为了根除错误，你需要两件事：重现错误的方法和让你定位错误并修正程序的信息。

有时，发现一个错误很容易。你自己就可以发现错误，测试部门产生一个清楚简单的测试，它可以显示程序中有错误，或者程序的输出总是不对。

有时，特别是在交互程序中，重现错误时90%的情况下有问题。当处理由领域用户送交的错误报告中的问题时，更是如此。用户打来的电话可能是这样的：

用户：你给我的数据库程序坏了。

程序员：出了什么错？

用户：我正排序，有时它排的顺序是错误的。

程序员：你用的是什么命令？

用户：排序命令。

程序员：告诉我你具体键入的是什么，一个按键一个按键地说，直到出错为止。

用户：我记不清具体的了。我做了很多次排序。

程序员：如果我来的话，你能不能再现这样的错误？

用户：当然。

5分钟后，程序员来到了用户的办公室，说出了关键的话：“做给我看”。用户键入命

令后程序运转正常，不管用户让它做什么。程序员没有办法，只得回到了自己的办公室，但是发现了来自用户的一条信息：“你走后5分钟程序出错了”。

示例17.1是一个简短的数据库查询程序。它要求用户输入一个名字，并在一个名字编码列表中查找该名字。虽然这个程序很简单，但它的结构与很大的更复杂的交互程序毫无两样。

示例17.1 base/base.cc

```

/*****
 * Database -- a very simple database program to
 *             look up names in a hard-coded list
 *
 * Usage:
 *   database
 *
 *   Program will ask you for a name.
 *   Enter the name? it will tell you whether
 *   the name is in the list.
 *
 *   A blank name terminates the program.
 *****/
const int STRING_LENGTH = 80;  /* Length of typical string */
#include <iostream.h>
#include <string.h>

main()
{
    char name[STRING_LENGTH];  // A name to look up

    int lookup(char *);        // Look up a name

    while (1) {
        cout << "Enter name: ";
        cin.getline(name, sizeof(name));

        // Check for blank name
        if (strlen(name) <= 0)
            break;

        if (lookup(name))
            cout << name << " is in the list\n";
        else
            cout << name << " is not in the list\n";
    }
    return (0);
}
/*****

```

```

* Lookup -- look up a name in a list
*
* Parameters
*     name -- name to look up
*
* Returns
*     1 -- name in the list
*     0 -- name not in the list
*****/
int lookup(char *name)
{
    // List of people in the database
    // Note: Last name is a NULL for end-of-list
    static char *list[] = {
        "John",
        "Jim",
        "Jane",
        "Clyde",
        NULL
    };

    int index;          // index into list

    for (index = 0; list[index] != NULL; ++index) {
        if (strcmp(list[index], name) == 0)
            return (1);
    }
    return (0);
}

```

A typical execution of this program is:

```

Enter name: Sam
Sam is not in the list
Enter name: John
John is in the list
Enter name:

```

当我们交付这个程序时，用户立即开始抱怨那个奇怪的问题。但当程序员在现场时，它就不出现。如果有一个小精灵站在用户的肩膀上，记下用户键入的所有内容不是很好吗？不幸的是，这个小精灵是不存在的；但是，我们可以修改这个程序，让它产生一个保存文件，文件中含有用户键入的所有按键。

程序中使用下面的语句读入用户键入的数据：

```
cin.getline(name, sizeof(name));
```


现在写一个新的程序：`extended_getline`，用它来代替`getline`函数。这个新函数不仅得到一行数据，还把用户的响应保存到一个保存文件中。

示例17.2 `xgets/xgets.cc`

```
#include <iostream.h>
#include <fstream.h>
/*
 * The main program will open this file if -S is on
 * the command line.
 */
ofstream save_file;          // File to use for saving input
int save_file_open = 0;      // Save file defaults to not open
/*****
 * extended_getline -- get a line from the input file  *
 *                    and record it in a save file if needed *
 *
 * Parameters                                           *
 *   line -- the line to read                          *
 *   size -- sizeof(line) -- maximum number of        *
 *           characters to read                        *
 *   file -- file to read data from                    *
 *           (normally stdin)                          *
 *
 * Returns                                              *
 *   NULL -- error or end-of-file in read              *
 *   otherwise line (just like getline)                *
 *****/
istream &extended_getline(char *line, int size, ifstream &file)
{
    istream *result;          /* Result of gets */

    result = &file.getline(line, size);

    // Did someone ask for a save file?
    if (save_file_open) {
        save_file << line << '\n';
    }

    return (*result);
}
```

我们还修改了主程序，使它能处理新的选项：“`-Sfile`”，这个选项用来指定一个保存文件。（一般地，大写字母用于调试和其它不常用的选项。）新的主程序如示例17.3所示。

示例17.3 `base/base2.cc`

```

/*****
 * Data Base -- A very simple database program to
 *              look up names in a hard-coded list
 *
 * Usage:
 *      database [-S<file>]
 *
 *      -S<file>      Specify save file for
 *                      debugging purposes
 *
 *                      Program will ask you for a name.
 *                      Enter the name; it will tell you whether
 *                      the name is in the list.
 *
 *                      A blank name terminates the program.
 *****/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

ofstream save_file;      // Save file if any
int save_file_open = 0; // Save file open flag

char *extended_getline(char *line, int size, istream &file);

main(int argc, char *argv[])
{
    char name[80];        // A name to look up
    char *save_file_name; // Name of the save file

    int lookup(char *name); // look up a name

    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            case 'S':
                save_file_name = &argv[1][2];
                save_file.open(save_file_name, ios::out);
                if (save_file.bad())
                    cerr << "Warning: Unable to open save file " <<
                        save_file_name << '\n';
                else
                    save_file_open = 1;
                break;
            default:
                cerr << "Bad option: " << argv[1] << '\n';
        }
    }
}

```



```

        exit (8);
    }
    --argc;
    ++argv;
}

while (1) {
    cout << "Enter name: ";
    extended_getline(name, sizeof(name), cin);

    /* ... rest of program ... */
}
return (0);
}

```

现在我们有用户键入的完整记录。看看他的输入，我们发现他键入了如下的内容：

```

Sam
 John

```

第二个名字的开头是一个空格，虽然“John”在列表中，但是“John”并没在列表中。这样我们发现是因为输入而导致出错；但是，更复杂的程序可能有更复杂的输入。调试时，我们可以键入所有的信息；或者，我们可以给`extended_getline`添加其它的功能，给它增加一个回放文件。当使用回放文件时，就不需要在键盘上键入所有的输入，而是从文件中读出相应的内容。

示例17.4 xgets/xgets2.cc

```

#include <iostream.h>
#include <fstream.h>
#
ofstream save_file;      // Save input in this file
int save_file_open = 0; // Save file has been opened
ifstream playback_file; // Playback data from this file
int playback_file_open = 0; // Playback file open flag
/*****
 * extended_getline -- get a line from the input file *
 *                    and record it in a save file if needed *
 *                    *
 * Parameters *
 *   line -- the line to read *
 *   size -- sizeof(line) -- maximum number of *
 *           characters to read *
 *   file -- file to read data from *
 *           (normally stdin) *
 *****/

```

```

*
* Returns
*     NULL -- error or end-of-file in read
*     otherwise line (just like getline)
*
*****/
istream &extended_getline(char *line, int size, istream &file)
{
    istream *result;          // Result of getline

    if (playback_file_open) {
        result = &playback_file.getline(line, size);
        if (file == cin)
            // Echo the input to the standard out
            // so the user sees it
            cout << line << '\n';
    } else
        result = &file.getline(line, size);

    // Did someone ask for a save file?
    if (save_file_open)
        save_file << line << '\n';

    return (*result);
}

```

我们给命令行增加一个新的回放选项: **-Pfile**。有了这个选项, 可以自动键入引起错误的命令。现在的主函数如示例17.5所示。

示例17.5 base/base3.cc

```

/*****
* Database -- A very simple database program to
*             look up names in a hard-coded list
*
* Usage:
*     database [-S<file>]
*
*     -S<file>      Specify save file for
*                   debugging purposes
*
*     -P<file>      Specify playback file for
*                   debugging or demonstration
*
*     Program will ask you for a name.
*     Enter the name; it will tell you whether
*     the name is in the list.
*****/

```

```

*
*          A blank name terminates the program.
*
*****/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

ifstream save_file;    // Save file if any
int save_file_open = 0; // Save file open flag
ofstream playback_file; // Playback data
int playback_file_open = 0; // True if playback in progress

char *extended_getline(char *line, int size, istream &file);

main(int argc, char *argv[])
{
    char name[80];    // A name to look up
    char *save_file_name; // Name of the save file
    char *playback_file_name; // Name of the playback file

    int lookup(char *name); // lookup a name

    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            case 'S':
                save_file_name = &argv[1][2];
                save_file.open(save_file_name, ios::out);
                if (save_file.bad())
                    cerr << "Warning: Unable to open save file " <<
                        save_file_name << '\n';
                else
                    save_file_open = 1;
                break;
            case 'P':
                playback_file_name = &argv[1][2];
                playback_file.open(playback_file_name, ios::in);
                if (playback_file.bad()) {
                    cerr <<
                        "Error: Unable to open playback file " <<
                        playback_file_name << '\n';
                    exit (8);
                }
                playback_file_open = 1;
                break;
            default:

```

```

        cerr << "Bad option: " << argv[1] << '\n';
        exit (8);
    }
    --argc;
    ++argv;
}

while (1) {
    cout << "Enter name: ";
    extended_getline(name, sizeof(name), cin);

    /* ... rest of program ... */
}
return (0);
}

```

现在当用户打电话报告出错时，我们可以告诉他：“使用保存文件功能，再试一次程序，然后把文件的备份给我。”用户可以运行程序，并将输入保存到文件save.txt中。

% database -Ssave.txt

```

Enter name: Sam
Sam is not in the list
Enter name: John
John is in the list
Enter name:

```

他送来了文件save.txt，我们用回放选项运行这个程序。

% database -Psave.txt

```

Enter name: Sam
Sam is not in the list
Enter name: John
John is in the list
Enter name:

```

现在我们有了重现程序的可靠办法。在许多情况下，这就成功了一半。一旦你重现了问题，就可以进入到下一步：找到问题并解决它。

有一次，程序员要求用户送来软盘备份。第二天一份特快专递就到了，里边是软盘的照片。但是这位用户不是个完完全全的计算机盲：他知道这是一张双面软盘，所以他把软盘的两面都照了下来。

连续调试

在开始调试之前，把旧的能运行的程序保存到安全的地方。（如果你使用了源控制系统，例如SCCS、RCS或是PSVS，那么你最后的一个版本就被登记。）许多情况下，当你查找问题时，可能会发现需要尝试几种不同的解决办法，或是添加临时调试代码。有时，你发现你已找到了问题的根源，并需要从头做起。这时最后的一个能运行的版本就变得异常珍贵了。

一旦你重现了问题，就必须判定是什么原因造成的。有几种方法可以做到。

分而治之

分而治之法已在第6章“判定和控制语句”中做了简要讨论。这种方法就是，在你知道数据还是正确的地方（一定要保证数据是正确的）及在数据不正确的地方之间的几个点加入cout语句。这种办法可以从包含错误的整段代码着手。更多的cout语句可以进一步缩小出错代码的范围，直至最终找到错误。

只用于调试的代码

分而治之法使用临时的cout语句。它们放在需要的地方，用过再删除掉。预处理器条件编译命令用于加入和删除调试代码。例如：

```
#ifdef DEBUG
    cout << "Width " << width << " Height " << height << '\n';
#endif /* DEBUG */
```

该段程序在没有定义DEBUG时被编译，所以当需要调试时，你可以定义它。

调试命令行开关

你可以不用编译时开关创建一个程序的特殊版本，而是在程序中永久地包含调试代码，并添加一个特殊的程序开关，这个开关打开调试信息的输出。例如：

```
if (debug)
    cout << "Width " << width << " Height " << height << '\n';
```

这里，**debug**是一个变量。当在命令行中出现-D选项时，该变量被设置。

这个方法有其先进性，即程序只有一个版本。含有“只用于调试的”代码的程序有如下的问题：除非代码常用，否则代码很快会过时。常常是，程序员试图找到一个错误，但其中的调试代码已过时了，需要再做修改。

调试命令行开关的另一个好处是，领域用户可以打开这个开关，保存输出信息，并把它送给你做进一步的分析。运行时开关应该用在所有的情况下，而不要用条件编译。除非你有某种理由不想让顾客得到调试信息。

有些程序用到了调试级别这个概念。0级仅输出最少的调试信息，1级输出多一点的信息，最高到9级，它输出所有的调试信息。

在Ghostscript*程序中用到了另一项调试技术。这个程序实现了字母调试的算法。命令选项-Zxxx为每种类型的诊断输出设置了调试标志。例如，f是用于填充算法的编码，p是用于路径跟踪器的编码。如果我想跟踪所有这些代码段，就需要指定-Zfp。

下面的代码实现这些选项：

```
/*
 * Even though we only put 1 zero, C++ will fill in the
 * rest of the arrays with zeros
 */
char debug[128] = {0};    // The debugging flags

main(argc, argv)
int argc;
char *argv[];
{
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /* .... normal switch .... */

            // Debug switch
            case 'Z':
                debug_ptr = &argv[1][2]
                // Loop for each letter
                while (*debug_ptr != '\0') {
                    debug[*debug_ptr] = 1;
                    ++debug_ptr;
                }
                break;
        }
        --argc;
        ++argv;
    }
}
```

* Ghostscript是像Postscript那样的解释器，它是Free Software Foundation的产品。Free Software Foundation公司位于675 Mass Ave., Cambridge, MA 02139。


```

    }

    /* Rest of program */
}

```

程序中可以这样使用:

```

if (debug['p'])
    cout << "Starting new path\n";

```

Ghostscript是一个大程序(大约25000行代码), 并且很不容易调试。这种调试方法可以让用户简单地就得到大量的信息。

通读输出

打开调试输出是得到信息的好办法。但在很多情况下, 数据太多了, 你想要的信息很容易就漏掉了。

C++通过使用“>file”选项, 可以让你改变输出方向。这样, 一般地输出到屏幕的信息可以输出到文件中。例如:

```
buggy -D9 >tmp.out
```

将运行程序buggy, 其调试级别为9, 并把输出送到文件tmp.out中。

系统中的文本编辑器还可以作为一个好的文件浏览器来使用。你可以用它的查找功能找到你想找的信息。

交互调试器

大多数编译器提供商提供了一个交互调试器。这些调试器具有这样的能力: 在程序的任一点停止执行、检查并改变变量的值和单步执行程序。因为每个调试器都不相同, 所以不可能详细讨论它们。

下面, 我们将讨论一个调试器gdb。许多UNIX机器都使用这个来自于Free Software Foundation公司的产品。Turbo-C++有它内置的调试器。虽然调试器使用的具体语法可能有所不同, 但是这里介绍的原理适用于所有的调试器。

基本的GDB命令是:

run 开始运行程序。

break 行号 在给定行插入一个断点。当程序运行到断点处时, 停止运行, 控制返回到调试器中。

break 函数名 在给定的函数的第一行插入一个断点。break main命令用于在程序的开

	头停止运行。
cont	在断点后继续执行。
print 表达式	显示表达式的值。
step	执行程序中的一个单行。如果当前语句调用一个函数，则单步执行函数。
next	执行程序中的一个单行，但把函数调用当作一个单行来对待。该命令用于跳过函数调用。
list	列出源程序。
where	打印当前活动函数的列表。
status	打印断点列表。
delete	删除断点。

有一个程序统计一串数中出现的3和7的个数。该程序在统计7的个数时出错。程序列在示例17.6中。

示例17.6 seven/count.cc

```

1 #include <iostream.h>
2 int seven_count;    /* Number of seven's in the data */
3 int data[5];        /* The data to count 3 and 7 in */
4 int three_count;    /* Number of threes in the data */
5
6 main() {
7     int index; /* Index into the data */
8     void get_data(int data[]);
9
10    seven_count = 0;
11    three_count = 0;
12    get_data(data);
13
14    for (index = 1; index <= 5; ++index) {
15        if (data[index] == 3)
16            ++three_count;
17        if (data[index] == 7)
18            ++seven_count;
19    }
20    cout << "Threes " << three_count <<
21         " Sevens " << seven_count << '\n';
22    return (0);
23 }
24 /*****
25  * get_data -- get 5 numbers from the command line  *

```



```

26  *****/
27 void get_data(int data[])
28 {
29     cout << "Enter 5 numbers\n";
30     cin >> data[1] >> data[2] >> data[3] >> data[4] >> data[5];
31 }

```

运行该程序时，输入数据3 7 3 0 2，得到结果：

Threes 3 Sevens 3

我们用要调试的程序名（count）来调用调试器（GDB）。调试器初始化，输出提示符（gdb），并等待命令。

% **gdb count**

```

GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (m68k-sun-sunos4.0.3),
Copyright 1994 Free Software Foundation, Inc...
(gdb)

```

我们不知道变量在哪被改变了，所以从程序的开头着手，直到发现错误为止。每一步我们都显示变量seven_count的值，并确保它是正确的。

我们必须在程序的开头停止，这样就可以单步执行它。命令**break main**告诉GDB在函数main 的第一条指令处设置一个断点。命令**run**告诉GDB启动程序，然后运行程序，直到它遇到第一个断点。

```

(gdb) break main
Breakpoint 1 at 0x22c2: file count.cc, line 10.
(gdb)

```

GDB用数1来标识断点。现在我们需要启动程序：

```

(gdb) run
Starting program: /usr/sdo/count/count

Breakpoint 1, main () at count.cc:10
10         seven_count = 0;
(gdb)

```

信息Breakpoint 1, main...表明程序遇到了一个断点，它现在已把控制权交给了调试器。

我们已执行到了初始化seven_count的地方。命令**next**将执行一个单语句，函数调用也作为一个语句来对待。（你所用的调试器中的命令名可能有所不同。）我们执行初始化部分，看看它是否起作用了：

```
(gdb) next
11         three_count = 0;
(gdb) print seven_count
$1 = 0
(gdb)
```

它确实执行了。我们继续执行下面的几行程序，每次都检查结果：

```
(gdb) next
12         get_data(data);
(gdb) print seven_count
$2 = 0
(gdb) next
Enter 5 numbers
3 7 3 0 2
14         for (index = 1; index <= 5; ++index) {
(gdb) print seven_count
$3 = 2
(gdb)
```

不知为何，seven_count变为2。我们执行的最后一个语句是get_data(data)，所以在这个函数中出了问题。我们在get_data的开始添加一个断点，移去main开头的那个断点，用run命令启动程序：

```
(gdb) break get_data
Breakpoint 2 at 0x23b2: file count.cc, line 29.
(gdb) info breakpoints
Num Type             Disp Enb Address          What
1  breakpoint         keep y  0x000022c2 in main at count.cc:10
2  breakpoint         keep y  0x000023b2 in get_data(int *) at count.cc:29
(gdb) delete 1
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) Y
Starting program: /usr/sdo/count/count
Breakpoint 2, get_data (data=0x208f8) at count.cc:29
(gdb)
```

现在再次进入单步执行方式，直至找到错误：

```
Breakpoint 2, get_data (data=0x208f8) at count.cc:29
29         cout << "Enter 5 numbers\n";
(gdb) print seven_count
$5 = 0
(gdb) next
30         cin >> data[1] >> data[2] >> data[3] >> data[4] >> data[5];
```

```

(gdb) print seven_count
$6 = 0
(gdb) next
Enter 5 numbers
3 7 3 0 2
31      }
(gdb) print seven_count
$7 = 2
(gdb) list 22
22      return (0);
23      }
24      /*****
25      * get_data -- get 5 numbers from the command line      *
26      *****/
27      void get_data(int data[])
28      {
29          cout << "Enter 5 numbers\n";
30          cin >> data[1] >> data[2] >> data[3] >> data[4] >> data[5];
31      }

```

在第30行数据是正确的，但到了31行时，数据不正确了，所以问题出在程序的第30行，即cin语句。我们已把范围缩小到一行内。通过检查，可以看到我们使用了data[5]，这在数据data中是不存在的。

但是为什么seven_count的结果不正确呢？因为data只有5个元素，没有data[5]。但是cin >> data[5]又要把data[5]放到某个地方，所以它决定放到内存中随机的一个地址中。在这个例子中，它选择了seven_count的位置。

调试一个二分查找程序

二分查找算法非常简单。你想知道所给的数是否在一个有序列表中，先把这个数与列表的中间元素相比较。如果它等于这个数，你很幸运——可以停止了。如果你的数较大，则可能在列表的上半部分找到该数。在列表的上半部分再继续找。如果你的数较小，则试着在列表的下半部分去找。一直继续这个过程，每次把列表缩小一半，直到找到这个数，或是列表变为一个数为止。

示例17.7使用二分查找法看看一个数是否存在于文件numbers.dat中。

示例17.7 search/searcho.cc

```

/*****
 * Search -- search a set of numbers      *
 *                                          *
 * Usage:      .                          *
 *      search                                *
 */

```

```

*           You will be asked numbers to look up           *
*                                                                 *
* Files:                                                                 *
*   numbers.dat -- numbers 1 per line to search                *
*   (numbers must be ordered)                                     *
*****/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>

const int MAX_NUMBERS = 1000;    // Max numbers in file
const char *const DATA_FILE = "numbers.dat"; // File with numbers

int data[MAX_NUMBERS]; // Array of numbers to search

int max_count;          // Number of valid elements in data
#
main()
{
    ifstream in_file;    // Input file
    int middle;          // Middle of our search range
    int low, high;       // Upper/lower bound
    int search;          // Number to search for

    in_file.open(DATA_FILE, ios::in);
    if (in_file.bad()) {
        cerr << "Error: Unable to open " << DATA_FILE << '\n';
        exit (8);
    }

    /*
     * Read in data
     */

    max_count = 0;
    while (1) {
        char line[30]; // Line from the input file

        if (in_file.eof())
            break;

        in_file.getline(line, sizeof(line));

        sscanf(line, "%d", data[max_count]);
    }
}

```

```

        if (data[max_count] == -1)
            break;

        ++max_count;
    }

    while (1) {
        cout << "Enter number to search for or -1 to quit:" ;
        cin >> search;

        if (search == -1)
            break;

        low = 0;
        high = max_count;

        while (1) {
            middle = (low + high) / 2;

            if (data[middle] == search) {
                cout << "Found at index " << middle << '\n';
            }

            if (low == high) {
                cout << "Not found\n";
                break;
            }

            if (data[middle] < search)
                low = middle;
            else
                high = middle;
        }
    }
    return (0);
}

```

我们的数据文件是:

numbers.dat

```

4
6
14
16
17

```



当我们在UNIX系统下执行这个程序时，结果为：

```
% search
```

```
Segmentation fault (core dumped)
```

当在MS-DOS下执行该程序时，系统锁死了，我们不得不按下reset开关。如果程序运行于Windows下，我们会得到一条应用程序错误的信息（如果幸运的话）。

哪种方法都不好。它意味着在程序中出错了，程序试图去读并不存在的内存。调试器GDB可以读这个文件，并帮助我们检查错在哪。

```
% gdb search
```

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (m68k-sun-sunos4.0.3),
Copyright 1994 Free Software Foundation, Inc...
(gdb) run
Starting program: /usr/sdo/search/search
```

```
Program received signal SIGSEGV, Segmentation fault.
0xec46320 in number ()
(gdb)
```

调试器告诉我们，程序是因过程产生的段错误而被中断的。但是我们并没有number过程！这个程序一定是属于C++库的。

现在使用where命令找出究竟是哪个函数调用了哪个函数（也称为栈轨迹(stack trace)）：

```
(gdb) where
#0  0xec46320 in number ()
#1  0xec45cc2 in _doscan ()
#2  0xec45b34 in sscanf ()
#3  0x2400 in main () at search.cc:48
(gdb)
```

最先打印的是当前的函数，然后是调用它的函数，依此类推，直到最外层的函数main。从这里，我们看到number被_doscan所调用，而_doscan又被sscanf调用。我们知道sscanf是一个库函数。其它的函数一定是sscanf调用的子函数。最后一个有控制权的函数是对sscanf的调用，它出现在main的第48行。

现在用list命令看一看这一行的源代码：

```
(gdb) list 48
```



```

43             if (in_file.eof())

44                 break;

45

46             in_file.getline(line, sizeof(line));

47

48             sscanf(line, "%d", data[max_count]);

49             if (data[max_count] == -1)

50                 break;

51

52             ++max_count;

(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) Y

```

这就是出错的那一行代码。

找错误的另一个办法是单步执行程序，直到发现错误为止。在最先列出的程序中的一个适当位置插入一个断点，然后开始单步执行过程。

```

Script started on Mon Oct 31 10:07:19 1994
% gdb search
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (m68k-sun-sunos4.0.3),
Copyright 1994 Free Software Foundation, Inc...
(gdb) list main
18     const char *const DATA_FILE = "numbers.dat";
        // File with numbers

19

20     int data[MAX_NUMBERS]; // Array of numbers to search
21     int max_count;         // Number of valid elements in data
22     main()
23     {
24         ifstream in_file; // Input file
25         int middle;        // Middle of our search range
26         int low, high;     // Upper/lower bound
27         int search;        // Number to search for
(gdb) break main
Breakpoint 1 at 0x2318: file search.cc, line 24.
(gdb) run
Starting program: /usr/sdo/search/search .

Breakpoint 1, main () at search.cc:24
24         ifstream in_file; // Input file
(gdb) step

```

```

29         in_file.open(DATA_FILE, ios::in);
(gdb) step
30         if (in_file.bad()) {
(gdb) step
39             max_count = 0;
(gdb) step
43                 if (in_file.eof())
(gdb) step
46                     in_file.getline(line, sizeof(line));
(gdb) step
48                         sscanf(line, "%d", data[max_count]);
(gdb) step

Program received signal SIGSEGV, Segmentation fault.
0xec46320 in number ()
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y

```

这种方法也指出第48行是出错的地方。检查后，我们注意到，我们忘记在sscanf函数的变量前放&号了。所以我们把第48行的代码：

```
sscanf(line, "%d", data[max_count]);
```

改为：

```
sscanf(line, "%d", &data[max_count]);
```

并再次执行程序。

注意

你可能会奇怪，我们为什么使用sscanf函数，因为

```
cin >> data[max_data];
```

具有同样的功能。

答案很简单。我们用sscanf函数来引发错误。没有指针错误的话，我们就没有什么要调试的了。cin语句更可靠，可靠的代码在讨论调试的章节中是没有用处的。

列表中的第一个数是4，所以我们用它试验程序。这一次的输出是：

```

Enter number to search for or -1 to quit: 4
Found at index 0
Found at index 0
Not found
Enter number to search for or -1 to quit: ^C

```


程序应该找到这个数，让我们知道它的下标是0，然后询问下一个数。但是我们得到的是两个found信息和一个not found信息。我们知道这一次程序顺利地执行到输出第一条found信息之处。但在这之后，又出错了。

再回到调试器中，使用list命令定位到found信息，并插入一个断点。

% gdb search

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (m68k-sun-sunos4.0.3),
Copyright 1994 Free Software Foundation, Inc...
```

(gdb) **list 66,77**

```
66             while (1) {
67                 middle = (low + high) / 2;
68
69                 if (data[middle] == search) {
70                     cout << "Found at index " << middle << '\n';
71                 }
72
73                 if (low == high) {
74                     cout << "Not found\n";
75                     break;
76                 }
77
```

(gdb) **break 70**

Breakpoint 1 at 0x249e: file search.cc, line 70.

(gdb) **run**

```
Starting program: /usr/sdo/search/search
Enter number to search for or -1 to quit: 4
```

Breakpoint 1, main () at search.cc:70

```
70             cout << "Found at index " << middle << '\n';
```

(gdb) **step**

Found at index 0

```
73             if (low == high) {
```

(gdb) **step**

```
78             if (data[middle] < search)
```

(gdb) **step**

```
81                 high = middle;
```

(gdb) **step**

```
67             middle = (low + high) / 2;
```

(gdb) **step**

```
69             if (data[middle] == search) {
```

(gdb) **step**

```
70             cout << "Found at index " << middle << '\n';
```

```

(gdb) step
Found at index 0
73             if (low == high) {
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y

```

程序并没有退出循环，而是用search继续查找。因为数已经找到，这个search就导致了奇怪的结果。我们在cout语句之后忘记写break语句了。`

我们要把：

```

if (data[middle] == search) {
    cout << "Found at index " << middle << '\n';
}

```

改为：

```

if (data[middle] == search) {
    cout << "Found at index " << middle << '\n';
    break;
}

```

做了这些修改后，再试一次程序：

```

% search
Enter number to search for or -1 to quit: 4
Found at index 0
Enter number to search for or -1 to quit: 6
Found at index 1
Enter number to search for or -1 to quit: 3
Not found
Enter number to search for or -1 to quit: 5
program runs forever (or until we abort it)

```

这一次程序失去了控制。我们并没有设置断点，只是重新运行程序。几秒钟过去了，我们相信已经陷入了死循环，只得用^C终止程序。一般情况下，这会使程序异常中断，并返回到外壳提示符下。因为我们运行的是调试程序，所以控制返回到GDB下。

```

% gdb search
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (m68k-sun-sunos4.0.3),
Copyright 1994 Free Software Foundation, Inc...
(gdb) run
Starting program: /usr/sdo/search/search

```

Enter number to search for or -1 to quit: 5

^C

Program received signal SIGINT, Interrupt.

0x2500 in main () at search.cc:79

```
79             if (data[middle] < search)
```

现在可以使用单步执行命令一步一步地执行无限循环，看看关键值的变化情况。

```
79             if (data[middle] < search)
(gdb) print middle
$1 = 0
(gdb) print data[middle]
$2 = 4
(gdb) print search
$3 = 5
(gdb) step
80             low = middle;
(gdb) step
67             middle = (low + high) / 2;
(gdb) step
69             if (data[middle] == search) {
(gdb) step
74             if (low == high) {
(gdb) step
79             if (data[middle] < search)
(gdb) step
80             low = middle;
(gdb) step
67             middle = (low + high) / 2;
(gdb) step
69             if (data[middle] == search) {
(gdb) step
74             if (low == high) {
(gdb) step
79             if (data[middle] < search)
(gdb) step
80             low = middle;
(gdb) step
67             middle = (low + high) / 2;
(gdb) step
69             if (data[middle] == search) {
(gdb) step
74             if (low == high) {
(gdb) step
79             if (data[middle] < search)
```

```

(gdb) step
80             low = middle;
(gdb) step
67             middle = (low + high) / 2;
(gdb) step
69             if (data[middle] == search) {
(gdb) print low
$5 = 0
(gdb) print middle
$6 = 0
(gdb) print high
$7 = 1
(gdb) print search
$8 = 5
(gdb) print data[0]
$9 = 4
(gdb) print data[1]
$10 = 6
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y

```

我们执行到

```
low = 0  middle = 0  high = 1
```

的时候有点问题。我们要找的项落在元素0和元素1之间。我们的算法有一个错误。很明显，中间的元素并不匹配。如果它匹配了，我们将退出，并得到found at信息。所以在新的查找区域中没有任何的点包括中间元素。调整间隔的代码是：

```

if (data[middle] < search)
    low = middle;
else
    high = middle;

```

它应该改为：

```

if (data[middle] < search)
    low = middle + 1;
else
    high = middle - 1;

```

程序全部修改后列在示例17.8中。

示例17.8 search/search4.cc

```

/*****
 * Search -- search a set of numbers
 *
 *****/

```

```

* Usage:                                     *
*      search                               *
*      You will be asked numbers to look up *
*                                           *
* Files:                                     *
*      numbers.dat -- numbers 1 per line to search *
*      (numbers must be ordered)             *
* *****/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>

const int MAX_NUMBERS = 1000; // Max numbers in file
const char *const DATA_FILE = "numbers.dat"; // File with numbers

int data[MAX_NUMBERS]; // Array of numbers to search
int max_count;         // Number of valid elements in data
main()
{
    ifstream in_file; // Input file
    int middle;       // Middle of our search range
    int low, high;    // Upper/lower bound
    int search;       // Number to search for

    in_file.open(DATA_FILE, ios::in);
    if (in_file.bad()) {
        cerr << "Error: Unable to open " << DATA_FILE << '\n';
        exit (8);
    }

    /*
     * Read in data
     */

    max_count = 0;

    while (1) {
        char line[30]; // Line from the input file

        if (in_file.eof())
            break;

        in_file.getline(line, sizeof(line));
    }
}

```

```

        sscanf(line, "%d", &data[max_count]);
        if (data[max_count] == -1)
            break;

        ++max_count;
    }

    while (1) {
        cout << "Enter number to search for or -1 to quit:" ;
        cin >> search;

        if (search == -1)
            break;

        low = 0;
        high = max_count;

        while (1) {
            if (low >= high) {
                cout << "Not found\n";
                break;
            }
            middle = (low + high) / 2;

            if (data[middle] == search) {
                cout << "Found at index " << middle << '\n';
                break;
            }

            if (data[middle] < search)
                low = middle + 1;
            else
                high = middle - 1;
        }
    }
    return (0);
}

```

对大多数程序来说，交互式的调试器运行良好。有时它们稍稍需要一点帮助，请考虑示例17.9。我们调试它时，发现当point_number为735时程序失败了。我们想在计算之前放一个断点。当调试器往程序中插入断点时，程序将正常执行到断点处，然后控制返回到调试器。这就允许用户检查并修改变量的值，也可以执行其它的调式命令。当键入cont命令时，程序继续执行，就好象什么都没有发生一样。问题是，在我们所要的那一点之前，要有734个断点，而我们并不想每次都停下来。

示例17.9 debug/cstop.cc

```
float point_color(int point_number)
{
    float correction;           // color correction factor
    extern float red, green, blue; // Current colors

    // Lookup color correction
    extern lookup(int point_number);

    correction = lookup(point_number);
    return (red*correction * 100.0 +
            blue*correction * 10.0 +
            green*correction);
}
```

如何能让调试器仅在`point_number == 735`的时候才停下来？我们可以加入下面的临时代码：

```
48:    if (point_number == 735) /* ### Temp code ### */
49:        point_number = point_number; /* ### Line to stop on ### */
```

第49行什么也不做，只是为了让调试器停在这里。我们可以用命令**break 49**在这一行放一个断点。程序执行734次第一条语句，然后，执行第49行，遇到断点。（有些调试器有一个条件断点。高级GDB命令**break 49 if part_number == 735**可以完成这样的功能，但是，你的调试器可能没有这样的高级功能。）

运行时错误

运行时错误通常是最容易修改的。有些类型的运行时错误是段违例、栈上溢和被0除。

段违例(segmentation violation)

这个错误表明，程序试图去逆引用一个含不正确值的指针。

栈上溢(stack overflow)

程序试图使用太多的临时变量。有时，这意味着程序太大，或是使用了很多的大临时数组。但大多数情况下，它是由于无限递归问题所致。几乎所有的UNIX系统自动检查这个错误。Turbo-C++系统仅当使用了编译选项-N时才检查栈上溢。

被0除(divide by 0)

被0除是一个明显的错误。UNIX使用错误信息Floating exception (core dumped)来标记这个错误，报告一个整数被0除。

在所有的这几种情况下，程序都将停止运行。在UNIX系统中，将输出运行程序的映像，

它称为内核文件。这个文件可以用调试器来分析，从而判定程序为什么死机。第一次运行示例17.7时，导致内核崩溃。（内核崩溃的一个原因是内核文件太大了，很快就把磁盘装满了。）

伴随运行错误的一个问题是，当错误发生时，程序立即终止。用于缓冲文件的缓冲区内容也没有全部输出。这可能产生令人吃惊的结果，请考虑示例17.10。

示例17.10 *debug/flush.cc*

```
#include <iostream.h>
main()
{
    int i, j;    /* Two random integers */

    i = 1;
    j = 0;
    cout << "Starting\n";
    cout << "Before divide...";
    i = i / j; // Divide-by-zero error
    cout << "After\n";
    return(0);
}
```

运行该程序时输出：

```
Starting
Floating exception (core dumped)
```

这可能会让你认为还没有执行到除法，而实际上，它已经执行了。那么信息“Before divide...”到哪去了呢？cout语句已经执行了，并把信息放到缓冲区里了，然后程序死机。缓冲区没有机会清空。

在代码中加一条强制清缓冲区的命令，可以反映出程序执行的真实情况，如示例17.11所示。

示例17.11 *debug/flush2.cc*

```
#include <iostream.h>
main()
{
    int i, j;    /* Two random integers */

    i = 1;
    j = 0;
    cout << "Starting\n";
    cout.flush();
    cout << "Before divide...";
    cout.flush();
    i = i / j; // Divide-by-zero error
}
```



```

        i = i / j; // Divide-by-zero error
        cout << "After\n";
        cout.flush();
        return(0);
    }

```

flush语句使得I/O的效率降低了，但时效性更强了。

公开声明调试方法

公开声明调试方法是这样一种方法：借助于它，程序员可以向别人解释他的程序，向感兴趣的人、向不感兴趣的人、向墙壁——不论他向谁解释多长时间都可以。

典型的声明如下：

```

"Hey, Bill, could you take a look at this? My program has a bug in
it. The output should be 8.0 and I'm getting -8.0. The output is
computed using this formula - and I've checked out the payment
value and rate and the date must be correct, unless there is
something wrong with the leap-year code, which-Thank you Bill,
you've found my problem."

```

当然，Bill不会回答一个字。

这种类型的调试也称做龙套角色。让其他人参与进来会带来全新的视角，因为其他人常常会发现你忽略的问题。

优 化

现在用一个词来说明优化：不需要。大多数程序不需要进行优化。它们运行得足够快。谁会关心一个交互程序是用了0.5秒还是用了0.2秒？

公平地讲，有许多慢程序是可以被加速的。本章介绍的简单优化步骤通常并不能做到这一点，一般要用更高效的算法取代设计得不好的算法。

对于一个书写良好的程序来说，让程序运行得更快的最简单方法是使用一个更快的计算机。很多情况下，买一台功能更强的计算机比优化一个程序来得更便宜，因为你可能会在代码中加入了新的错误。不要期望优化会产生奇迹。通常大多数程序只会提高10%到20%的速度。

示例17.12对一个矩阵（二维数组）进行初始化。

```

示例17.12 matrix/matrix1.cc
const int X_SIZE = 60;

```

```

const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    int x, y;    // Current element to initialize

    for (x = 0; x < X_SIZE; ++x) {
        for (y = 0; y < Y_SIZE; ++y) {
            matrix[x][y] = -1;
        }
    }
}

```

如何对这个程序进行优化？首先我们注意到，我们使用了两个局部变量。通过用修饰符`register`描述这些变量，告诉编译器这些变量使用频繁，要把它们放到快速的寄存器中，而不要放到相对慢一些的主内存中。寄存器的数目因机器的不同而不同。像PC这样的慢机器有2个，大多数UNIX系统大约有11个，超级计算机可以有多达128个的寄存器。说明的寄存器变量可能比你拥有的寄存器还要多。C++将把多余的变量放在主内存中。

修改后的程序列在示例17.13中。

示例17.13 *matrix/matrix2.cc*

```

const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    register int x, y;    // Current element to initialize

    for (x = 0; x < X_SIZE; ++x) {
        for (y = 0; y < Y_SIZE; ++y) {
            matrix[x][y] = -1;
        }
    }
}

```

外层循环执行60次。这就是说，执行内层循环所附带的开销要执行60次。如果我们把循环的次序颠倒一下，则只需处理30次内层循环。

一般地，循环要有次序。使最内层的循环最复杂，最外层的循环最简单。示例17.14中，`init_matrix`函数重排了循环次序。

示例17.14 *matrix/matrix3.cc*

```
const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    register int x, y;    // Current element to initialize

    for (y = 0; y < Y_SIZE; ++y) {
        for (x = 0; x < X_SIZE; ++x) {
            matrix[x][y] = -1;
        }
    }
}
```

2的幂的效能

找一个数组的下标要执行乘法。例如，为执行下面这一行语句：

```
matrix[x][y] = -1;
```

程序必须计算放-1的位置。程序将执行下列各步：

- 1.取matrix的地址。
- 2.计算 $x * Y\_SIZE$ 。
- 3.计算y。
- 4.把上述三部分加在一起，形成地址。在C++中相应的代码为：

```
*(matrix + (x * Y_SIZE) + y) = -1;
```

但是，你不必这样访问数组元素，因为C++会处理细节问题。但是了解它是如何处理的，对编写更高效的代码是有帮助的。

几乎所有的C++编译器都把乘以2的幂（2、4、8、...）的计算转化为移位操作，所以一个昂贵的操作（乘法），就转为了不昂贵的操作（移位）。

例如：

```
i = 32 * j;
```

被编译为：

```
i = j << 5; /* 2**5 == 32 */
```

Y_SIZE是30，它不是2的幂。把Y_SIZE增大到32会浪费一些内存，但程序更快了。

示例17.15表明如何利用2的幂的优势条件。

示例17.15 *matrix/matrix4.cc*

```
const int X_SIZE = 60;
const int Y_SIZE = 32;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    register int x, y;    // Current element to initialize

    for (y = 0; y < Y_SIZE; ++y) {
        for (x = 0; x < X_SIZE; ++x) {
            matrix[x][y] = -1;
        }
    }
}
```

因为我们要初始化一段连续的内存位置，所以可以从第一个位置开始，在随后的X_SIZE * Y_SIZE个元素中放入数-1来初始化矩阵。用这个方法，我们可以把循环的层数减小到1层。矩阵的下标从标准下标（matrix[x][y]）变为指针逆引用（*matrix_ptr），前者要执行移位和加法操作，后者仅执行增量操作（++matrix_ptr）。在示例17.16中，我们把数组转为了指针。但是为什么既有循环计数器又有matrix_ptr呢？我们能不能把两者合并？事实上是可以的。

示例17.16 *matrix/matrix5.cc*

```
const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    register int index;        // Element counter
    register int *matrix_ptr;  // Current element

    matrix_ptr = &matrix[0][0];
    for (index = 0; index < X_SIZE * Y_SIZE; ++index) {
        *matrix_ptr = -1;
        ++matrix_ptr;
    }
}
```

在示例17.17中，我们去掉了循环计数器，把它与数组指针合并到一起。

示例17.17 *matrix/matrix6.cc*

```
const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    register int *matrix_ptr;    // Current element

    for (matrix_ptr = &matrix[0][0];
         matrix_ptr <= &matrix[X_SIZE - 1][Y_SIZE - 1];
         ++matrix_ptr) {

        *matrix_ptr = -1;
    }
}
```

函数已经优化好了。使它更好的唯一办法是我们自己把它变为汇编语言。这样它的执行可能更快，但是汇编语言是根本不能移植的，并且极易出错。

库程序**memset**可以用一个单字符值来填充矩阵或是数组。我们可以用它来初始化这个程序中的矩阵。像**memset**这样的常用库函数常常是汇编语言的，而且利用独特的依赖于处理器的技巧来完成一些操作，可能比在C++中的实现要快一些。在示例17.18中，我们用**memset**函数进行初始化。

示例17.18 *matrix/matrix7.cc*

```
#include <string.h>

const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

void init_matrix(void)
{
    memset(matrix, -1, sizeof(matrix));
}
```

现在函数中仅有一条函数调用语句。好像不应该调用一个只含有调用另一个函数的函数，我们不得不承担两次函数调用的开销。如果我们从main函数中调用**memset**会更好。为什么我们不重写这段代码，用**memset**代替**init_matrix**呢？因为要调用几百次**init_matrix**，我们不想做这些编辑工作。

我们如何能减少函数调用的开销呢？让函数成为内联的即可。这个函数的最终版本使



用了内联函数来减少所有的调用开销，见示例17.19。

示例17.19 *matrix/matrix8.cc*

```
#include <string.h>

const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

inline void init_matrix(void)
{
    memset(matrix, -1, sizeof(matrix));
}
```

问题17.1：为什么memset能成功地把矩阵元素初始化为-1，但当我们试图用它把每个元素设置为1时，执行却失败了？

```
#include <string.h>

const int X_SIZE = 60;
const int Y_SIZE = 30;

int matrix[X_SIZE][Y_SIZE];

inline void init_matrix(void) {
    memset(matrix, 1, sizeof(matrix));
}
```

如何优化

矩阵初始化函数说明了几个优化策略。这就是：

删去不变代码

没必要放到循环内的代码就放到循环的外面。例如：

```
for (i = 0; i < 10; ++i)
    matrix[i] = i + j * 10;
```

可以写为：

```
j_times_10 = j * 10;
```



```
for (i = 0; i < 10; ++i)
    matrix[i] = i + j_times_10;
```

如果可能的话，大多数好的优化编译器会自动为你完成这一步。

循环次序

嵌套的循环应该有次序，最简单的循环放在最外层，最复杂的循环放在最内层。

少花力气

用便宜的操作取代昂贵的操作。表17.1列出了常见操作的相对花费。

表17.1 操作的相对花费

操 作	相对花费
文件输入和输出 (<<和>>)， 包括C函数printf和scanf	1000
new和delete	800
三角函数 (sin, cos, ...)	500
浮点运算 (任何操作)	100
整数除法	30
整数乘法	20
函数调用	10
简单数组下标	6
移位	5
加/减	5
指针逆引用	2
按位与、或、非	1
逻辑与、或、非	1

注 意

使用C格式的函数scanf、printf和sscanf时，其调用开销非常昂贵。因为它们必须一个字符一个字符地读入格式串，寻找转换字符(%)。然后它们必须在字符串和数之间进行昂贵的转换。在强调时间的代码中应避免使用这些函数。

引用参数

使用常量引用参数 (const 类型 &) 取代用于结构、联合和类的常量参数。

2的幂

当进行整数乘法和除法时，使用2的幂。大多数编译器将用移位来代替操作。

指针

对数组来说，指针比下标更快。并有一些技巧可用。

内联函数

使用内联函数可以减小函数调用的开销。但是这会使代码加长，增加调试难度。（见下面的实例。）

实例研究：内联函数对正常函数

有一次，我为一家大的计算机制造厂商编制一个字处理程序。其中一个函数`next_char`，它从当前文件中读取下一个字符。它在程序的各个地方使用上千次。当第一次我们把`next_char`做为一个函数来测试这个程序时，程序慢得难以忍受。分析程序时我们发现，有90%的时间是花在`next_char`上。所以我们把它改为一个内联函数。速度提高了一倍；但是，我们的代码增加了40%，并需要一块内存扩充卡才能工作。虽然速度令人满意了，但代码的长度又不可忍受了。最后，不得不写一个自己进行优化的汇编语言程序，使程序的大小和速度都可以接受。

实例研究：优化一个涂色算法

另外一次我被指派优化一个程序，该程序为一大幅画涂色。问题是，程序用了8个小时才处理完一幅画。这样，一天只能处理一幅。

我做的第一件事是在一台带有浮点加速器的机器上运行该程序。这样，时间缩减到大约6小时。下一步，我被允许使用另一个项目组暂时闲置的高速RISC计算机。这样，时间缩减到2小时。

仅仅使用更快的机器就节省了6小时，虽然我还没修改代码呢。

在最内层循环中，有两个相对简单的函数只调用了一次。用宏重写这些函数，又节省了15分钟。

下一步，我把所有能改的浮点操作改为整数操作。这又节省了1小时45分钟运行时间内的30分钟。然后，我注意到，程序花5分钟去读一个ASCII码文件，该文件中含有一长串用于转换过程的浮点数。我们知道`scanf`是一个极昂贵的函数，所以我把文件改为二进制后，初始化过程几乎不占用什么时间。这时，总的运行时间降到了1小时10分钟。

仔细检查代码，使用我了解的每一个技巧，又节省了5分钟。离我的一小时运行一次的目标只剩下5分钟了。此时，我的项目重新进行调整，程序束之高阁，留待以后再用了。

编程练习

练习17.1： 找一个以前的程序，用交互调试器运行它，并检查几个变量的中间值。

练习17.2: 编写一个矩阵乘法函数。建立一个测试程序，它不仅测试函数，还测试执行的时间。用指针优化这个程序，看看节省了多少时间。

练习17.3: 编写一个程序，把数组中的元素加起来。然后优化它。

练习17.4: 编写一个程序，统计一个字符数组中的位数。通过使用寄存器整数变量来优化它。统计该程序作用于几个不同大小的数组时所花费的时间，优化后比优化前节省了多少时间。

练习17.5: 编写自己的库函数`memcpy`，并优化它。`memcpy`的实现是用汇编语言写的，这能充分利用处理器奇特的地方及它的优势。你的`memcpy`与之相比怎么样？

答 疑

问题17.1: 问题出在`memset`是一个字符填充函数。一个整数由2或4个字节（字符）组成。每个字节赋以值1。所以2个字节的整数将接收下面的值：

```
integer = 0x0101;
```

1个字节大小的十六进制值-1是0xFF。2个字节大小的十六进制值-1是0xFFFF。所以我们可以把两个单字节的-1放到一起，变为-1。对0也可以这样处理。换作其它的数时将出错。例如，1是0x01。两个字节的1是0x0101，或是257。



本章内容

- 运算符函数
- 运算符成员函数
- 复数类的完整定义
- 编程练习
- 答疑

第18章

运算符重载

我们都知道两个整数相加是怎么回事。但是C++没有内置的复数类型，所以它不知道如何相加两个复数。但是通过称为**运算符重载(operator overloading)**的这一C++特性，你可以“教”C++如何处理复数。运算符重载用来定义一组函数，它使用正常的运算符+、-、*和/对复数进行加法、减法、乘法和除法运算。

本节我们定义一个复数类。我们先定义基本的C++复数类。一个复数(*complex number*)包含两部分，实部和虚部：

```
class complex
{
    protected:
        //
        // Complex numbers consist of two parts
        //
        double real_part;           // The real part
        double imaginary_part;      // The imaginary part
        // ...
}
```

下一步，我们定义几个成员函数。这些函数包括常见的构造函数和析构函数，还有取复数实部和虚部的函数。

```
public:
    // Default constructor initializes the number to (0 + 0i)
    complex(void)
    {
        real_part = 0;
        imaginary_part = 0;
    }
    // Copy constructor - initialize one complex from another
    complex(const complex& other_complex)
    {
        real_part = other_complex.real_part;
```

```

        imaginary_part = other_complex.imaginary_part;
    }
    // Construct a complex from two reals
    // If only real supplied assume the imaginary part is .0
    complex(double init_real, double init_imaginary = 0.0)
    {
        real_part = init_real;
        imaginary_part = init_imaginary;
    }
    // Destructor does nothing
    ~complex() {}

    //
    // Functions to return the parts of the number
    //
    double real(void) const    // "const" is discussed later
    {
        return (real_part);
    }
    double imaginary(void) const
    {
        return (imaginary_part);
    }

    // Define functions to set parts of a number
    void set_real(double real) {
        real_part = real;
    }
    void set_imaginary(double imaginary) {
        imaginary_part = imaginary;
    }
};

```

注意

你可能会记起来，函数后出现的const已在第14章“类的进一步讨论”中介绍过。

现在我们想使用复数。变量说明是简单的，甚至把它们初始化为 $(3 + 2i)$ 也很容易。

```

complex start;    // Starting point for the graph
complex end(3.0, 2.0);    // Ending point

```

可是，当我们想把两个复数相加时怎么办？我们需要定义一个函数来做这件事：

```
// Version 1 of the complex add function
inline complex add(const complex &oper1, const complex &oper2)
{
    complex result(oper1.real() + oper2.real(),
                  oper1.imaginary() + oper2.imaginary());
    return (result);
}
```

这个函数中有这么几点要注意。首先，该函数定义为，它接受两个复数，并返回一个复数。我们可以把加法组合在一起：

```
// Add three complex numbers
answer = add(first, add(second, third));
```

两个自变量都使用的是常量引用参数（`const complex &`），这是把结构传递到函数中最高效的方法。最后，因为它是这样小的一个函数，所以我们把它定义为一个内联函数，这样效率更高。

在这个函数中，我们明确定义了一个`result`，并返回它。我们可以把这两步合成一步：

```
// Version 2 of the complex add function
inline complex add(const complex &oper1, const complex &oper2)
{
    return (complex(oper1.real() + oper2.real(),
                  oper1.imaginary() + oper2.imaginary()));
}
```

虽然它不太易懂，但效率更高。

明白C++私下为我们做了哪些很重要。对于这样一条简单语句：

```
answer = add(first, second);
```

它调用了构造函数、赋值运算符函数和析构函数，所有这些都在一小段代码中完成。

在`add`函数的第一种实现中，我们为结果明确分配了一个变量。在第二种实现中，C++自动为结果创建一个临时变量。这个数没有名字，在`return`语句之外，它并不真正存在。

创建临时变量导致调用构造函数。然后临时变量赋给`answer`，所以我们要调用赋值函数。赋值之后，C++不再使用这个临时变量，所以调用析构函数扔掉它。

运算符函数

把`add`函数作用于复数有些不便。最好是当我们用`+`运算符相加两个复数时，能让C++自动调用这个函数。这就是提出运算符重载之原因。我们所要做的就是编写如下的`add`函数：

```
inline complex operator +(const complex &oper1, const complex &oper2)
{
    return (complex(oper1.real() + oper2.real(),
                    oper1.imaginary() + oper2.imaginary()));
}
```

余下的交给C++去处理。

注：运算符重载函数应该小心使用。你应该试图让它们遵从常用的规则。所以，+应该做加法；-应该做减法；等等。C++ I/O流因把移位运算符（<<和>>）定义为输入和输出运算符，而打破了这个规则。有时这会引起混淆，例如：

```
cout << 8 << 2;
```

是在“8”的后面输出“2”，还是输出表达式的值（8 << 2）？除非你是专家，否则你说不清楚。此处，将输出数“8”和数“2”。

你已经见到如何重载+运算符。现在，让我们研究研究可以使用的其它运算符。

二元算术运算符

二元运算符需要两个自变量，运算符的每一边各有一个。例如，乘法和除法运算符是二元运算符：

```
x * y;
a / b;
```

一元运算符只需要一个自变量。一元运算符包括一元-和地址运算符（&）：

```
-x
&y
```

二元算术运算符函数带两个常量参数，并产生一个结果。参数之一必须是一个类或是一个结构。结果可以是任一个。例如，对于二元加法来说，下列函数是合法的：

```
complex operator +(complex v1, complex v2);
complex operator +(complex v1, real v2);
complex operator +(real v1, complex v2);
complex operator +(real v1, real v2);
```

我们已为复数类的加法定义了几个不同的函数。当重载运算符时，这样一堆定义是常见的。

表18.1列出可以重载的二元运算符。

表18.1 可以重载的二元运算符

运算符	含 义
+	加法
-	减法
*	乘法
/	除法
%	取模
^	按位异或
&	按位与
	按位或
<<	左移
>>	右移

关系运算符

关系运算符包括等于(==)和不等于(!=)。一般地,它们带两个常量类,并返回0或1。(实际上,它们可以返回任何值,但是这样会违反关系运算符的惯例。)

用于复数类的等于运算符是:

```
inline int operator == (const complex &oper1, const complex &oper2)
{
    return ((oper1.real() == oper2.real(),
             oper1.imaginary() == oper2.imaginary()));
}
```

表18.2列出了关系运算符。

表18.2 关系运算符

运算符	含 义
==	等于
!=	不等于
<	小于
>	大于
<=	小于或等于
>=	大于或等于

一元运算符

一元运算符带有一个参数，如取负（-）。用于复数类型的取负运算为：

```
inline complex operator - (const complex &oper)
{
    return (complex(-oper.real(), -oper.imaginary()));
}
```

表18.3列出了一元运算符。

表18.3 一元运算符

运算符	含 义
+	正
-	负
*	逆引用
&	地址
~	补

快捷运算符

像+=和-=这样的运算符是用于更复杂运算的快捷运算符。可是+=和-=的返回值是什么呢？仔细研究C++标准后发现，这些运算符返回变量增量或减量后的值。例如：

```
i = 5;
j = i += 2;    // Don't code like this
```

给j赋值7。用于复数类的+=函数为：

```
inline complex &operator +=(complex &oper1, const complex &oper2)
{
    oper1.set_real(oper1.real() + oper2.real());
    oper1.set_imaginary(oper1.imaginary() + oper2.imaginary());
    return (oper1);
}
```

注意，和我们定义的其它运算符函数不一样，第一个参数不是一个常量。而且，我们返回对第一个变量的引用，而不是一个新的变量或是参数的拷贝。

表18.4列出快捷运算符。

表18.4 简单的快捷运算符

运算符	含 义
+=	增量
-=	减量
*=	相乘
/=	相除
%=	余
^=	相异或
&=	相与
=	相或
<<=	左移
>>=	右移

增量和减量运算符

增量和减量运算符有两种形式：前缀和后缀。例如：

```
i = 5;
j = i++;    // j = 5
i = 5;
j = ++i;    // j = 6
```

这两个运算符都使用函数`operator++`。我们怎么区分它们呢？C++语言有一个程序来处理这种情况。运算符的前缀形式有一个自变量，即要累加的项。后缀形式有两个自变量，要累加的项和一个整数。使用的这个实际整数是没有什么意义的，它仅仅是一个位置，表明运算符的两种不同形式。

如果我们为复数类型定义的++是指把实部增1，则处理两种++形式的函数为：

```
// Prefix    x = ++c
inline complex &operator ++(complex &oper)
{
    oper.set_real(oper.real() + 1.0);
    return (oper);
}

// Suffix    x = c++
inline complex operator ++(complex oper, int)
{
    complex result(oper);    // Result before we incremented
    oper.set_real(oper.real() + 1.0);
}
```



```

    return (result);
}

```

这很混乱。C++不让我们过多地使用这些技巧：没有使用的整数参数。实际中，我从来不使用增量的后缀形式，而且总是把前缀形式单独放在一行中。这样做可以避免这些问题。

看看前缀和后缀形式的代码可以决定是使用前缀还是使用后缀形式。正如你已看到的，前缀形式比后缀形式简单得多。所以使用前缀形式不仅简化你的代码，还使编译器的操作更容易。

表18.5列出增量和减量运算符。

表18.5 增量和减量运算符

运算符	含 义
++	增量
--	减量

逻辑运算符

逻辑运算符包括与（&&）、或（||）和非（!）。它们可以被重载，但是你可以这样做并不意味着你应当这样做。在理论上，逻辑运算符只用于布尔值。在实际中，因为C++没有布尔类型，所以它们操作于整数。不要因重载而混淆这些问题。

表18.6列出了逻辑运算符。

表18.6 逻辑运算符

运算符	含 义
	逻辑或
&&	逻辑与
!	逻辑非

I/O运算符

你已把<<和>>运算符用于输入和输出。实际上，这些运算符是移位运算符的重载。这样做的好处是I/O操作相当简单，其代价是容易搞混。

我们应该像其它数据类型一样输出复数，为此我们需要定义一个<<运算符。

我们把数据送到输出流类ostream中，数据本身是复数类型的，输出函数为：



```

inline ostream &operator << (ostream &out_file, const complex &number)
{
    out_file << '(' << number.real() << ',' <<
        number.imaginary() << ')';
    return (out_file);
}

```

函数返回对输出文件的一个引用。这样调用者可以把一串<<运算符放在一起，如：

```

complex a_complex(1.2, 3.4);
cout << "The answer is " << a_complex << '\n';

```

这段代码的结果为：

```
The answer is (1.2, 3.4)
```

一般地，<<运算符带两个常量自变量。本例中，第一个参数是一个非常量ostream。这是因为<<运算符用作输出时有副作用，主要操作是把数据送到输出流中。一般地，把运算符所没有的副作用加在运算符上不是个好想法。

输入应该和输出一样简单。你可以认为我们要做的就是读数（和有关的额外字符）：

```

// Simple-minded input operation
inline istream &operator >> (istream &in_file, complex &number) {
    double real, imaginary; // Parts of the number
    char l_paren, comma, r_paren; // Extra chracters output as part of
    in_file >> l_paren >> real >> comma >> imaginary >> r_paren;
    number.set(real, imaginary);
    return (in_file);
}

```

实际中，它并不简单。首先，我们必须调用一个特殊的成员函数ipfx，来告诉I/O系统我们想使用格式读函数。

```

inline istream &operator >> (istream &in_file, complex &number)
{
    // ...
    in_file.ipfx(1); // Tell the I/O system we are reading formatted

```

下一步我们跳过前导空格。

```
in_file >> ws; // Skip white space
```

现在应该指向数开头的“(”了。但在读这个字符之前，如果可能的话，必须先检查是否有异常情况。

```
if (in_file.bad()) return (in_file);
```

现在，读“(”。当然，只是要确证一下，我们要查看是不是读到了“(”，如果不是的话，程序会异常中断。

```
in_file >> ch;      // Get character after white space
if (ch != '(') {
    in_file.set(ios::failbit);    // We have an error
    return (in_file);
}
```

用set函数来设置一个标志，表明输入操作中遇到了问题。这允许调用程序通过调用bad函数来检测输入是否正确。（该函数还可能导致一个例外，详细内容请见第22章“例外”。）

我们已读到了“(”。下面读的是复数实部：

```
in_file >> real;
```

注意“,”和与之相关的空格。每一步我们都检查有没有错误：

```
if (in_file.bad()) return (in_file);
in_file >> ws >> ch;      // Get first character after number
if (in_file.bad()) return (in_file);

if (ch != ',') {
    in_file.set(ios::failbit);
    return (in_file);
}
```

下面读虚部：

```
in_file >> imaginary;
```

最后，数的后面应该跟着“)”。

```
in_file >> ws >> ch;
if (in_file.bad()) return (in_file);

if (ch != ')') {
    in_file.set(ios::failbit);
    return (in_file);
}
```

工作完成，存储并输出结果。

```
number.set(read, imaginary);
return (in_file);
}
```

读复数的完整代码列在示例18.1中。

示例18.1

```
inline istream &operator >> (istream &in_file, complex &number)
{
    double real, imaginary; // Real and imaginary part
    char ch;                // Random character used to verify input

    number.set(0.0, 0.0);    // Initialize the number (just in case)

    in_file.ipfx(1);         // Tell the I/O system we are reading formatted
    in_file >> ws;           // Skip white space

    if (in_file.bad()) return (in_file);

    in_file >> ch;           // Get character after white space
    if (ch != '(') {
        in_file.set(ios::failbit); // We have an error
        return (in_file);
    }

    in_file >> real;

    if (in_file.bad()) return (in_file);

    in_file >> ws >> ch;      // Get first character after number

    if (in_file.bad()) return (in_file);

    if (ch != ',') {
        in_file.set(ios::failbit);
        return (in_file);
    }
    in_file >> imaginary;

    in_file >> ws >> ch;
    if (in_file.bad()) return (in_file);

    if (ch != ')') {
        in_file.set(ios::failbit);
        return (in_file);
    }
    number.set(real, imaginary);
    return (in_file);
}
```

下标运算符“[]”

[]是C++用来定数组下标的运算符。在第20章“高级指针”中，我们将看到，当定义一个模拟数组的类时，这是一个非常有用的运算符。一般地，该函数带有两个自变量：模拟数组的类和下标，它返回对数组中一个项的引用。

```
double &operator[](array_class &array, int index)
```

我们将在第23章“模块化程序设计”中详细讨论[]运算符。

new和delete

这里我们并不多讲对全局运算符new和delete的重载。第一，第20章“高级指针”中才介绍这两个运算符，所以现在你并不了解它们的意义。第二，当你了解了这两个运算符时，你就不想重载它们了。

我只见过一个重载了new和delete运算符的程序（或者至少是等价的C程序）。那个程序是由一位非常聪明的程序员编写的，他喜欢做点古怪的事情。结果，程序的调试异常困难。

所以除非你是一位非常聪明的程序员，否则不要重载new和delete。而且，如果你是一位聪明的程序员，也请不要重载new和delete。说不定哪天我会调试你编写的代码呢。

不常用的运算符

C++中有一组丰富的运算符。但是，其中有一些很少使用。这些运算符是：

() 允许你为一个类定义缺省函数。

, 逗号运算符。它把两个表达式连接在一起。它很少使用，可能永远也不会被重载。

->* 指向成员的指针。很少使用。

-> 类成员。

所有这些运算符都将在第28章“C++内的‘角落’”中讨论。

运算符成员函数

到目前为止，我们一直把运算符重载函数用作普通的函数。它们还可以定义它为成员函数。唯一的区别是：作为成员函数时，第一个自变量即类本身是隐含的。例如，你可以把运算符+=定义为一个普通的函数，也可以定义为一个成员函数。下面再次列出作为普通函数时的代码。

```
inline complex &operator +=(complex &oper1, const complex &oper2)
{
    oper1.set_real(oper1.real() + oper2.real());
```

```

    oper1.set_imaginary(oper1.imaginary() + oper2.imaginary());
    return (oper1);
}

```

用作成员函数时的代码为:

```

class complex {
    // ...
public:
    inline complex &operator +=(const complex &oper2)
    {
        real_part += oper2.real();
        imaginary_part += oper2.imaginary();
        return (*this);
    }
}

```

函数中用到一个关键字**this**。**this**是预定义的变量，它指向当前的对象。例如，你可以用下面的语句访问数据成员**real_part**:

```
real_part += oper2.real();
```

这条语句还可以这样写:

```
this->real_part += oper2.real();
```

在大多数情况下，你无需使用**this**。但是，在少数情况下，如使用**+=**运算符时，用它还是方便的。

你应该使用哪种运算符重载呢？当然是能使程序最清晰易读的那种。一般地，我们对简单运算符使用标准函数，如**+**、**-**、*****和**/**；而对于快捷和一元运算符，使用成员函数，如**+=**、**-=**、**++**和一元运算符**-**。

有些重载函数只能作为成员函数来使用。这些函数包括**casting**运算符及用于类的**new**和**delete**。

转 换

最后，我们讨论转换运算符。转换是把一种类型变为另一种类型的方法。例如，当我们想**complex**类型转换为**double**类型时，我们只要了实部。我们可以为这个函数定义如下的转换运算符:

```

class complex:
public:
    // (We didn't really put this in our complex class)
    double operator double() {return (real_part);}
}

```

当C++想把**complex**转换为**double**时自动调用该函数。

麻烦的是，定义了一个转换运算符后，你就给了C++调用的权力。就我个人而言，我想知道C++在何时调用它，所以我避免建立转换运算符。除非你有非常好的理由定义它，否则不要建立转换运算符函数。

复数类的完整定义

示例18.2列出复数类的完整代码。头文件的开始摘要说明了定义的所有函数。在创建这个类时，我发现有许多函数（具体地说是29个）只有一两行的代码。在这些函数中写全函数注释块会隐没代码。换句话说，这是添加注释会使程序混乱的少数（极少数）情况之一。所以，大部分的小函数没有注释。

创建这个类时，我注意到，许多函数有类似的结构。例如，+=很像-=，等等。事实上，我写-=运算符的代码时，是先拷贝+=函数，然后再做少量的修改。当你想为类定义完整的运算符集时，因为C++包含了一组丰富的运算符，所以可能会有重复。

最后，简单运算符定义在文件`complex.h`中，而较长的函数写在文件`complex.cc`中。

示例18.2 `complex/complex.h,complex/complex.cc`

File: `complex.h`

```
#ifndef __complex_h__ // Avoid double includes
#define __complex_h__ // Prevent double include

#include <iostream.h>
#include <math.h>

/*****
 * Complex class
 *
 * Members defined
 *      complex() // Default constructor
 *      complex(real, imaginary) // Specify two parts
 *                               // for construction
 *      complex(complex) // Copy constructor
 *
 *      real() // Get real part
 *      imaginary() // Get imaginary part
 *
 *      set(real, imaginary) // Set both parts of #
 *      set_real(real) // Set real part of #
 *      set_imaginary(imaginary) // Set imaginary part
 *
 * Operator member functions
 *
 *      c -- a complex number
 *      s -- a scalar (double)
 *****/
```

```

*      C = C
*      C += C;
*      C += S;
*      C -= C;
*      C -= S;
*      C /= C;
*      C /= S;
*      C *= C;
*      C *= S;
*
* The following functions don't really make a lot of
* sense for complex numbers, but they are defined
* for the purpose of illustration
*      C++
*      ++C
*      C--
*      --C
*
* Arithmetic operators defined
*      C = C + C;
*      C = S + C;
*      C = C + S;
*      C = C - C;
*      C = S - C;
*      C = C - S;
*      C = C * C;
*      C = S * C;
*      C = C * S;
*      C = C / C;
*      C = S / C;
*      C = C / S;
*      -C
*      +C
*      ostream << C    // Output function
*      istream >> C    // Input function
*****/
class complex
{
private:
    //
    // Complex numbers consist of two parts
    //
    double real_part;    // The real part
    double imaginary_part; // The imaginary part

```



```

public:
    // Default constructor, zero everything
    complex(void)
    {
        real_part = 0.0;
        imaginary_part = 0.0;
    }

    // Copy constructor
    complex(const complex& other_complex)
    {
        real_part = other_complex.real_part;
        imaginary_part = other_complex.imaginary_part;
    }

    // Construct a complex out of two real numbers
    complex(double init_real, double init_imaginary = 0.0)
    {
        real_part = init_real;
        imaginary_part = init_imaginary;
    }

    // Destructor does nothing
    ~complex() {}

    //
    // Function to return the parts of the number
    //
    double real(void) const
    {
        return (real_part);
    }
    double imaginary(void) const
    {
        return (imaginary_part);
    }

    // Functions to set parts of a number
    void set(double real, double imaginary) {
        real_part = real;
        imaginary_part = imaginary;
    }

    void set_real(double real) {
        real_part = real;
    }

```

```

    }

    void set_imaginary(double imaginary) {
        imaginary_part = imaginary;
    }

    complex operator = (const complex& oper2) {
        complex result (oper2);
        return (result);
    }

    complex& operator += (const complex& oper2) {
        real_part += oper2.real();
        imaginary_part += oper2.imaginary();
        return (*this);
    }

    complex& operator += (double oper2) {
        real_part += oper2;
        return (*this);
    }

    complex& operator -= (const complex& oper2) {
        real_part -= oper2.real();
        imaginary_part -= oper2.imaginary();
        return (*this);
    }

    complex& operator -= (double oper2) {
        real_part -= oper2;
        return (*this);
    }

    complex& operator *= (const complex& oper2) {
        // Place to hold the real part of the result
        // while we compute the imaginary part
        double real_result = real_part * oper2.real() -
                               imaginary_part * oper2.imaginary();

        imaginary_part = real_part * oper2.imaginary() +
                          imaginary_part * oper2.real();

        real_part = real_result;
        return *this;
    }

```

```

    complex& operator *= (double oper2) {
        real_part *= oper2;
        imaginary_part *= oper2;
        return (*this);
    }

    complex& operator /= (const complex& oper2);

    complex& operator /= (double oper2) {
        real_part /= oper2;
        imaginary_part /= oper2;
        return (*this);
    }

// c++
    complex operator ++(int) {
        complex result(*this);
        real_part += 1.0;
        return (result);
    }

// ++c
    complex &operator ++(void) {
        real_part += 1.0;
        return (*this);
    }

// c--
    complex operator --(int) {
        complex result(*this);
        real_part -= 1.0;
        return (result);
    }

// --c
    complex &operator --(void) {
        real_part -= 1.0;
        return (*this);
    }
};

inline complex operator + (const complex& oper1, const complex& oper2)
(
    return complex(oper1.real()      + oper2.real(),

```

```

        oper1.imaginary() + oper2.imaginary());
    }

inline complex operator + (const complex& oper1, double oper2)
{
    return complex(oper1.real() + oper2,
        oper1.imaginary());
}

inline complex operator + (double oper1, const complex& oper2)
{
    return complex(oper1 + oper2.real(),
        oper2.imaginary());
}

inline complex operator - (const complex& oper1, const complex& oper2)
{
    return complex(oper1.real() - oper2.real(),
        oper1.imaginary() - oper2.imaginary());
}

inline complex operator - (const complex& oper1, double oper2)
{
    return complex(oper1.real() - oper2,
        oper1.imaginary());
}

inline complex operator - (double oper1, const complex& oper2)
{
    return complex(oper1 - oper2.real(),
        -oper2.imaginary());
}

inline complex operator * (const complex& oper1, const complex& oper2)
{
    return complex(
        oper1.real() * oper2.real() - oper1.imaginary() * oper2.imaginary(),
        oper1.real() * oper2.imaginary() + oper1.imaginary() * oper2.real());
}

inline complex operator * (const complex& oper1, const double oper2)
{
    return complex(oper1.real() * oper2,
        oper1.imaginary() * oper2);
}

```

```

inline complex operator * (const double oper1, const complex& oper2)
{
    return complex(oper1 * oper2.real(),
                   oper1 * oper2.imaginary());
}

extern complex operator / (const complex& oper1, const complex& oper2);

inline complex operator / (const double& oper1, const complex& oper2) {
    return (complex(oper1, 0.0) / oper2);
}

inline complex operator / (const complex& oper1, const double& oper2) {
    return (oper1 / complex(oper2, 0.0));
}

inline int operator == (const complex& oper1, const complex& oper2)
{
    return ((oper1.real() == oper2.real()) &&
           (oper1.imaginary() == oper2.imaginary()));
}

inline int operator != (const complex& oper1, const complex& oper2)
{
    return (!(oper1 == oper2));
}

inline complex operator - (const complex& oper1)
{
    return complex(-oper1.real(), -oper1.imaginary());
}

inline complex operator + (const complex& oper1)
{
    return complex(+oper1.real(), +oper1.imaginary());
}

inline ostream& operator << (ostream& out_file, const complex& number)
{
    out_file << '(' << number.real() << ',' << number.imaginary() << ')';
    return (out_file);
}

extern istream& operator >> (istream& in_file, complex& number);

```

```

#endif /* __complex_h__ */      // Avoid double includes

File: complex.cc

#include "complex.h"

/*****
 * c = c / c -- complex division
 *
 * Parameters
 *      oper1, oper2 -- two operands of the divide
 *
 * Returns
 *      result of the divide
 *****/
complex operator / (const complex& oper1, const complex& oper2)
{
    // Denominator of the result
    double den = fabs(oper2.real()) + fabs(oper2.imaginary());

    // Real part of the oper1 factor
    double oper1_real_den = oper1.real() / den;
    // Imaginary part of the oper1 factor
    double oper1_imag_den = oper1.imaginary() / den;

    // Real part of the oper2 factor
    double oper2_real_den = oper2.real() / den;
    // Imaginary part of the oper2 factor
    double oper2_imag_den = oper2.imaginary() / den;

    // Normalization factor
    double normalization = oper2_real_den * oper2_real_den +
                           oper2_imag_den * oper2_imag_den;

    return complex((oper1_real_den * oper2_real_den +
                    oper1_imag_den * oper2_imag_den) / normalization,
                  (oper1_imag_den * oper2_real_den -
                    oper1_real_den * oper2_imag_den) / normalization);
}

/*****
 * c /= c -- complex divide by
 *
 * Parameters
 *****/

```



```

*      oper2 -- operator to divide by
*
* Returns
*      reference to the result of the divide
*****/
complex& complex::operator /= (const complex& oper2)
{
    // Denominator of the result
    double den = fabs(oper2.real()) + fabs(oper2.imaginary());

    // Denominator -- operator 1 real part
    double oper1_real_den = real_part / den;

    // Denominator -- operator 1 imaginary part
    double oper1_imag_den = imaginary_part / den;

    // Denominator -- operator 2 real part
    double oper2_real_den = oper2.real() / den;

    // Denominator -- operator 2 imaginary part
    double oper2_imag_den = oper2.imaginary() / den;

    // Normalization factor
    double normalization = oper2_real_den * oper2_real_den +
                           oper2_imag_den * oper2_imag_den;
    real_part      = (oper1_real_den * oper2_real_den +
                     oper1_imag_den * oper2_imag_den) / normalization;
    imaginary_part = (oper1_imag_den * oper2_real_den -
                     oper1_real_den * oper2_imag_den) / normalization;
    return (*this);
}

/*****
* istream >> complex -- read a complex number
*
* Parameters
*      in_file -- file to read
*      number -- place to put the number
*
* Returns
*      reference to the input file
*****/
istream &operator >> (istream &in_file, complex &number)
{
    double real, imaginary; // Real and imaginary part

```

```

char ch;                // Random character used to verify input

number.set(0.0, 0.0);    // Initialize the number (just in case)

in_file.ipfx(1);         // Tell the I/O system we are reading formatted
in_file >> ws;           // Skip white space

if (in_file.bad()) return (in_file);

in_file >> ch;           // Get character after white space
if (ch != '(') {
    in_file.setf(ios::failbit);    // We have an error
    return (in_file);
}

in_file >> real;

if (in_file.bad()) return (in_file);

in_file >> ws >> ch;      // Get first character after number

if (in_file.bad()) return (in_file);

if (ch != ',') {
    in_file.setf(ios::failbit);
    return (in_file);
}

in_file >> imaginary;

in_file >> ws >> ch;
if (in_file.bad()) return (in_file);

if (ch != ')') {
    in_file.setf(ios::failbit);
    return (in_file);
}
number.set(real, imaginary);
return (in_file);
}

```

问题18.1: 为什么示例18.3失败了? 运行时, 它一遍一遍地输出:

```

Copy constructor called
Copy constructor called

```


提示：回忆第13章中“拷贝构造函数”一节。

示例18.3 *equal/equal.cc*

```
1 #include <iostream.h>
2
3 class trouble {
4     public:
5         int data;
6
7         trouble(void);
8         trouble(const trouble &old);
9         trouble operator = (trouble old_trouble);
10 };
11
12 trouble::trouble(void) {
13     data = 0;
14 }
15
16 trouble::trouble(const trouble &old) {
17     cout << "Copy constructor called\n";
18     *this = old;
19 }
20
21 trouble trouble::operator = (trouble old_trouble) {
22     cout << "Operator = called\n";
23     data = old_trouble.data;
24     return (*this);
25 }
26
27 int main()
28 {
29     trouble trouble1;
30     trouble trouble2(trouble1);
31
32     return (0);
33 }
```

编程练习

练习18.1：编写一个类，处理带分数，如“ $1\frac{1}{3}$ ”。为这些分数定义加法、减法、乘法和除法运算符。例如： $1\frac{1}{3} + 2\frac{1}{2} = 3\frac{5}{6}$ 。

练习18.2：编写一个定点数类来处理数。所有的数都是DDDDDD.D形式的。即，所有的数都只有一位小数。用整数来实现这个类。



练习18.3: 编写一个类, 实现稀疏整数数组。它很像一个简单的整数数组:

```
int simple_array[100];
```

但是和一个简单数组不同的是, 数组的下标为0到1000000。这是个坏消息。好消息是, 任何时候, 最多只有100个元素有非零值。其余元素为0。

练习18.4: 编写一个时间类。实现时间的加、减、读和输出。

练习18.5: 编写一个日期类, 实现日期的加、减、读和输出。日期的格式为MM/DD。假定年份不跨年。

练习18.6: (高级) 编写一个全日期类, 日期格式为MM/DD/YY, 实现日期的加、减、读和输出。

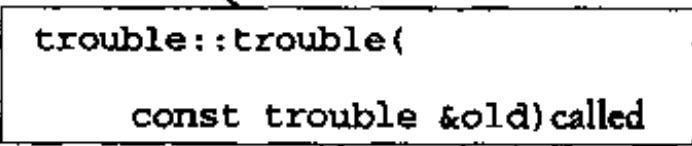
答 疑

解答18.1: 拷贝构造函数调用operator =函数。该函数的参数表是:

```
trouble trouble::operator = (trouble old_trouble) {
```

这个函数的参数是按值传送的参数。见到这类的参数时, C++调用拷贝构造函数, 把参数放到栈中。

```
trouble trouble::operator = (trouble old_trouble) {  
    cout << "Operator = called\n";  
    data = old_trouble.data;  
    return (*this);  
}
```



```
trouble::trouble(  
    const trouble &old) called
```

所以我们陷入了一个无限循环中。拷贝构造函数调用operator =函数。C++见到按值调用参数, 就去调用拷贝构造函数, 而构造函数又调用operator =函数, 这样又调用到拷贝构造函数。这个过程一直持续下去, 直到系统的栈空间满, 或者用户异常中断程序为止。

解决的办法是, 把传送给operator =的参数变为一个引用。这不仅更有效, 而且程序能运转起来了。

```
trouble trouble::operator = (const trouble &old_trouble) {
```

1

.

.

.

.

.

.

..

.

.



本章内容:

- 浮点数格式
- 浮点数加法/减法
- 乘法
- 除法
- 上溢和下溢
- 舍入误差
- 精度
- 让舍入误差最小
- 判定精度

第19章

浮点数

计算机处理整数时非常好，算法简单、精确并且快速。而处理浮点数时正相反。计算机进行浮点数运算时有很大困难。

本章讨论浮点数运算时遇到的一些问题。另外，为了阐明浮点数运算的原理，我们定义了一个简单的十进制浮点数格式。我们建议你不用计算机，而用笔和纸来做这些题。这样，你可以直接面对这些问题。

计算机使用的格式非常类似于本章定义的格式，只是本章中的基数为10，而计算机中的基数为2、8或16。但是，这里讲述的所有问题都可能在计算机中出现。

浮点数格式

浮点数包含三部分：符号、小数和幂。分数表示为4位十进制数。幂是1位十进制数。所以我们定义的格式为：

$$\pm f.fff \times 10^{\pm e}$$

这里：

$\pm$ 是符号（加号或减号）。

$f.fff$ 是4位十进制小数。

$\pm e$ 是1位数字的幂。

零表示为 $+0.000 \times 10^0$ 。我们用“E”格式来表示这些数： $\pm f.fff \times 10E \pm e$ 。

这种格式类似于许多计算机中使用的浮点数格式。IEEE已经定义了浮点数标准（#742），但是并不是所有的机器都使用这个标准。

表19.1列出一些浮点数。

表19.1 浮点数

Notation	Number
+1.000E+0	1.0
+3.300E+5	330000.0
-8.223E-3	-0.008223
+0.000E+0	0.0

本章定义的浮点数运算严格遵守一组规定。为了使错误减少到最少，我们使用了一位监视位。这是运算期间加在小数最后的一个额外位。许多计算机在它们的浮点运算部件中都使用了监视位。

浮点数加法/减法

计算机把2.0和0.3相加时，可能要执行下列各步：

1. 准备数。

+2.000E+0 数为2.0

+3.000E-1 数为0.3

2. 给两个数加上监视位。

+2.0000E+0 数为2.0

+3.0000E-1 数为0.3

3. 把幂最小的数右移一位，同时幂增1。继续这个过程，直到两个数的幂相等。

+2.0000E+0 数为2.0

+0.3000E-0 数为0.3

4. 把两个小数相加。结果的幂与两个数的幂相等。

+2.0000E+0 数为2.0

+0.3000E-0 数为0.3

+2.3000E+0 结果为2.3

5. 标准化该数，左移或右移它，直到它的小数点左侧仅有一位非零数时为止。相应地调整它的幂。像+0.1234E+0应当标准化为+1.2340E-1。因为+2.3000E+0已经是标准形式，所以我们不用再做什么了。

6. 最后，如果监视位大于或等于5，把下一位四舍五入。否则丢掉。

+2.3000E+0 丢去最后一位



+2.300E+0 结果为2.3

7. 对于浮点数减法, 把第二个操作数的符号变反, 再做加法。

乘 法

当我们想做乘法时, 例如, 0.12×11.0 , 要遵守下面的规则:

1. 给两个数加上监视位。

+1.2000E-1 数为0.12

+1.1000E+1 数为11.0

2. 两个小数进行乘法运算, 幂数相加。 ($1.2 \times 1.1 = 1.32$) ($-1 + 1 = 0$)

+1.2000E-1 数为0.12

+1.1000E+1 数为11.0

+1.3200E+0 结果为1.32

3. 结果标准化。如果监视位小于等于5, 则把下一位四舍五入。否则丢掉。

+1.3200E+0 数为1.32

注意, 在乘法中, 你不必进行移位。乘法的规则比加法要少得多。整数乘法比整数加法要慢得多。浮点数乘法的速度最接近于加法的速度。

除 法

完成100.0除以30.0这样的除法, 要执行下面的步骤。

1. 加上监视位。

+1.0000E+2 数为100.0

+3.0000E+1 数为30.0

2. 小数部分进行除法, 幂次相减。

+1.0000E+2 数为100.0

+3.0000E+1 数为30.0

+0.3333E+1 数为3.333

3. 结果标准化。

+3.3330E+0 数为3.333

4. 如果监视位小于等于5, 则把下一位四舍五入。否则丢掉。

+3.333E+0 数为3.333

上溢和下溢

计算机可以处理的数有一定的限制。下面计算的结果会是什么样的呢？

$9.000\text{E}+9 \times 9.000\text{E}+9$

相乘后，得到：

8.1×10^{19}

但是，我们的幂次是一位数，放不下19。这是上溢（*overflow*）的一个例子（有时叫幂上溢）。当上溢时，有些计算机做了特殊处理，所以程序会中断，并输出一条错误信息。其它一些计算机做得没有这么好，只产生一个错误结果（如8.100E+9）。遵从IEEE浮点标准的计算机会得到一个特殊的值，名为+Infinity。

当数太小，计算机不能处理时，发生下溢（*underflow*）。例如：

$1.000\text{E}-9 \times 1.000\text{E}-9$

结果为：

1.0×10^{-18}

因为-18太小了，不能用一位数字表示，所以发生了下溢。

舍入误差

浮点数不是精确的数。每个人都知道 $1 + 1$ 等于2，但是 $1/3 + 1/3$ 等于不等于 $2/3$ ？看下面浮点数的计算过程：

$2/3$ 表示为浮点数是6.667E-1

$1/3$ 表示为浮点数是3.333E-1

+3.333E-1

+3.333E-1

+6.666E-1 或 0.6666

它不等于+6.667E-1。

每台计算机在进行浮点数运算时都存在类似的问题。例如，用二进制浮点数就不能精确地表示0.2。

永远不要用浮点数来表示金额。因为我们要表示元以下的数值，如1元9角8分表示为：

```
float amount = 1.98;
```



浮点数计算的次数越多，舍入误差就越大。银行、信用卡和IRS都很注意金额的正确性，拿给IRS一张不太精确的支票会使他们不高兴。金额应该以最小单位的整数来存储。

精度

多少位小数是精确的呢？乍看上去，你可能会被误导地说，所有4位数字。读过上一节舍入误差的读者可能会改为3位。

答案是：精度依赖于计算。确定的计算，如两个很接近的数相减，会得到不精确的结果。考虑下面的计算：

$$1 - 1/3 - 1/3 - 1/3$$

1.000E+0

-3.333E-1

-3.333E-1

-3.333E-1

或

1.000E+0

-3.333E-1

-3.333E-1

-3.333E-1

0.0010E+0 或 1.000E-3

正确的答案是0.000E+0，而我们得到的是1.000E-3。小数的第1位是错的。这是一个称为舍入误差问题的例子。当进行浮点运算时，就会产生舍入误差。

让舍入误差最小

有许多方法可以减小舍入误差。一种方法是我们已经讨论过的监视位。另一个方法是使用double类型取代float类型。这样，数的精度差不多是原来的两倍，表示的范围也为原来的两倍。舍入误差问题也少了两倍，但是舍入误差仍然存在。

减少浮点数带来的问题的高级技术，可以在数值分析的书中找到。它们超出了本书讨论的范围。本章的目的是让你对遇到的问题有一个初步了解。

就其本身特性来说，浮点数是不精确的。人们往往认为计算机是一台非常精确的机器。它们可以是精确的，但也可能得出谬之千里的结果。你一定要注意偷偷溜到程序中的错误。

判定精度

有一个简单的判定浮点数精确度的方法（用于简单计算中）。下面的程序累加1.0 + 0.1、1.0 + 0.01、1.0 + 0.001，依此类推，直到第二个数太小，不足以影响结果时为止。

原来的C语言规定，所有的浮点数都处理为双精度。C++删去了这条规定，但是，因为许多C++编译器继承于C编译器，所以常见的C++算术运算都处理为双精度。这就是说，表达式：

```
float number1, number2;
...
while (number1 + number2 != number1)
```

等价于：

```
while (double(number1) + double(number2) != double(number1))
```

如果你使用了1 + 0.001小技巧，而把float自动转换为double的操作，可能会给你机器的精度计算带来麻烦。（有时，对于32位格式却报告有84位精度。）示例19.1计算用在等式和内存中的两个浮点数的精度。

示例19.1 float/float.cc

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

```
main()
{
    // Two numbers to work with
    float number1, number2;
    float result;           // Result of calculation
    int counter;            // Loop counter and accuracy check

    number1 = 1.0;
    number2 = 1.0;
    counter = 0;

    while (number1 + number2 != number1) {
        ++counter;
        number2 = number2 / 10.0;
    }
    cout << setw(2) << counter << " digits accuracy in calculations\n";

    number2 = 1.0;
    counter = 0;
```

```

    while (1) {
        result = number1 + number2;
        if (result == number1)
            break;
        ++counter;
        number2 = number2 / 10.0;
    }
    cout << setw(2) << counter << " digits accuracy in storage\n";
    return (0);
}

```

在带MC68881浮点芯片的Sun-3/50上运行该程序，得到：

```

20 digits accuracy in calculations
8 digits accuracy in storage

```

该程序只给出大致的浮点数精度计算，更准确的定义在包含文件*float.h*中。

精度和速度

*double*类型变量的精度差不多是一般的*float*变量的两倍。大多数人设想双精度计算会比单精度计算占用更多的时间。情况并不总是这样。假定我们有一个老版本的编译器，它进行任何计算都使用*double*。

对于下面的等式：

```

float answer, number1, number2;
answer = number1 + number2;

```

C++必须执行下面各步：

1. 把*number1*从单精度转换为双精度。
2. 把*number2*从单精度转换为双精度。
3. 进行双精度加法。
4. 把结果转换为单精度并存储在*answer*中。

如果变量是*double*类型的，C++将只执行下面各步：

1. 进行双精度加法。
2. 把结果存储在*answer*中。

正如你看到的，第二种形式简单得多，它不需要进行三次类型转换。在有些情况下，程序从单精度转换为双精度会使程序运行得更快。

因为C++规定浮点数有double或是float，所以你控制不了任何事。把所有的float变为double可能会使程序运行得更快、更慢或者相同。使用浮点数时你能肯定的唯一一件事是，结果是不精确的。

包含PC和Sun/3系列机在内的许多计算机都有一个特殊的芯片来处理所有的浮点数计算，名叫浮点处理器。使用Motorola 68881浮点芯片（用在Sun/3中）和PC上的浮点芯片进行的实际测试表明，单精度和双精度的运行速度是一样的。

幂级数

许多三角函数是用幂级数来计算的。例如，正弦函数的级数为：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

问题是，要得到四位精度时需要计算多少项？表19.2列出了计算 $\sin(\pi/2)$ 时的各项。

表19.2 计算 $\sin(\pi/2)$ 时的各项

	Term	Value	Total
1	x	1.571E+0	
2	$\frac{x^3}{3!}$	6.462E-1	9.248E-1
3	$\frac{x^5}{5!}$	7.974E-2	1.005E+0
4	$\frac{x^7}{7!}$	4.686E-3	9.998E-1
5	$\frac{x^9}{9!}$	1.606E-4	1.000E+0
6	$\frac{x^{11}}{11!}$	3.604E-6	1.000E+0

从表中可以看出，我们需要计算5项。如果我们想计算 $\sin(\pi)$ ，则得到表19.3。

π 需要计算9项。所以，不同的角度需要计算的项数也不同。（附录D中列出计算四位精度下正弦函数的源程序。）

编译器设计人员在设计正弦函数时进退两难。如果他们提前知道要使用多少项的话，他们就可以用项数优化其算法了。但是，他们对有些角度的计算是以牺牲精度为代价的。所以要在速度和精度方面进行权衡。

表19.3 计算 $\sin(\pi)$ 时的各项

	Term	Value	Total
1	x	3.142E+0	
2	$\frac{x^3}{3!}$	5.170E+0	-2.028E+0
3	$\frac{x^5}{5!}$	2.552E-0	5.241E-1
4	$\frac{x^7}{7!}$	5.998E-1	-7.570E-2
5	$\frac{x^9}{9!}$	8.224E-2	6.542E-3
6	$\frac{x^{11}}{11!}$	7.381E-3	-8.388E-4
7	$\frac{x^{13}}{13!}$	4.671E-4	-3.717E-4
8	$\frac{x^{15}}{15!}$	2.196E-5	-3.937E-4
9	$\frac{x^{17}}{17!}$	7.970E-7	-3.929E-4
10	$\frac{x^{19}}{19!}$	2.300E-8	-3.929E-4

不要想当然。因为计算机不同，数也不同，这就是精度。库函数也可以得出坏结果——特别是当计算很大或很小的数时。大多数情况下，你不用担心这些函数的计算结果，但要注意它们适用的范围。

最后，有一个问题， $\sin(1,000,000)$ 结果是什么？我们的浮点格式只有四位精度，而正弦函数是个周期函数。即， $\sin(0) = \sin(2\pi) = \sin(4\pi)$ 。所以， $\sin(1,000,000)$ 等于 $\sin(1,000,000 \bmod 2\pi)$ 。

因为浮点数格式只有四位精度， $\sin(1,000,000)$ 实际上是 $\sin(1,000)$ 。因为1000比 2π 要大，所以得出的正弦函数的结果是没有意义的。

编程练习

练习19.1：编写一个类，它用串来表示浮点数，采用本章所用的格式。类应该包含浮点数的读、写、加、减、乘和除等函数。

我在Cal Tech参加过两位教授主持的一个物理班。有一次，一位教授给我们做关于太阳的演讲，当他说到，“...就是说，太阳内部的温度有13,000,000到25,000,000度。”此时，另外一位导师打断了他的话，问到，“是摄氏温标还是绝对温标（绝对零度或是摄氏-273）？”

第一位教师转向黑板，过了一分钟，他说，“这有什么不同吗？”我说这个故事的寓意在于，当你的计算可能有12,000,000这么大的误差时，273的差距又有何妨呢？

练习19.2：创建一个类，它处理定点数。定点数中小数点右边的数位是一个常量（固定的）。

本章内容:

- 指针、结构和类
- delete运算符
- 链表
- 顺序链表
- 双向链表
- 树
- 树的打印
- 程序的其余部分
- 象棋程序中用到的数据结构
- 编程练习
- 答疑

第20章

高级指针

C++非常有用且复杂的特性是其指针的使用。使用指针，你可以创建复杂的数据结构，如链表和树。图20.1展示了这些数据结构中的几个例子。

到目前为止，你接触的所有数据结构都由编译器来分配空间，不管是永久变量还是临时变量。你可以使用指针建立并分配动态数据结构，这样的结构可以按照需要变大变小。在本章，你将学习如何使用这些较常见的动态数据结构。

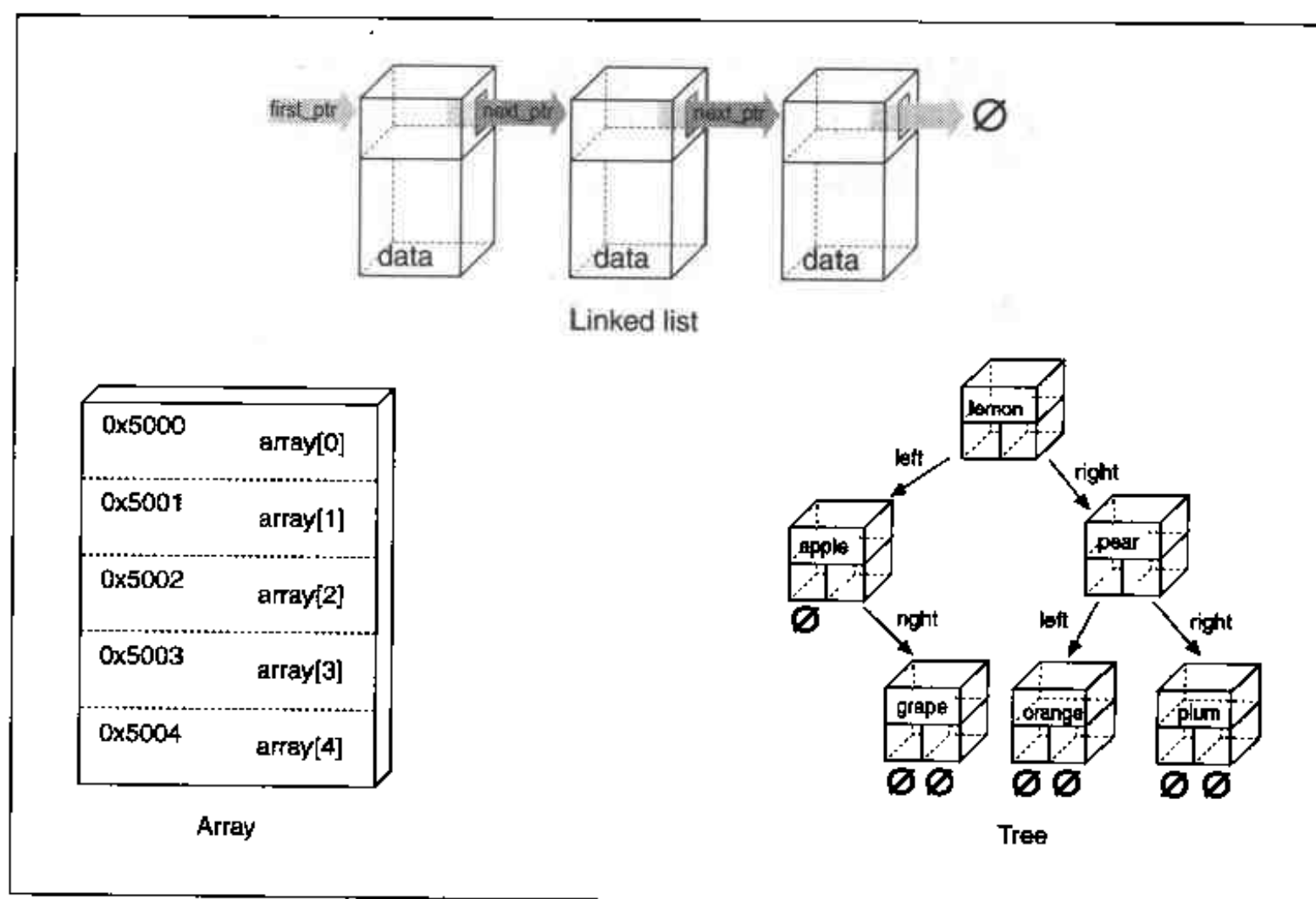


图20.1 指针使用示例

指针、结构和类

结构和类可以包含指针，甚至是一个指向同一结构其它实例的指针。在下面的例子中：

```
class item {  
    public:  
        int value;  
        item *next_ptr;  
};
```

item结构图示于图20.2中。

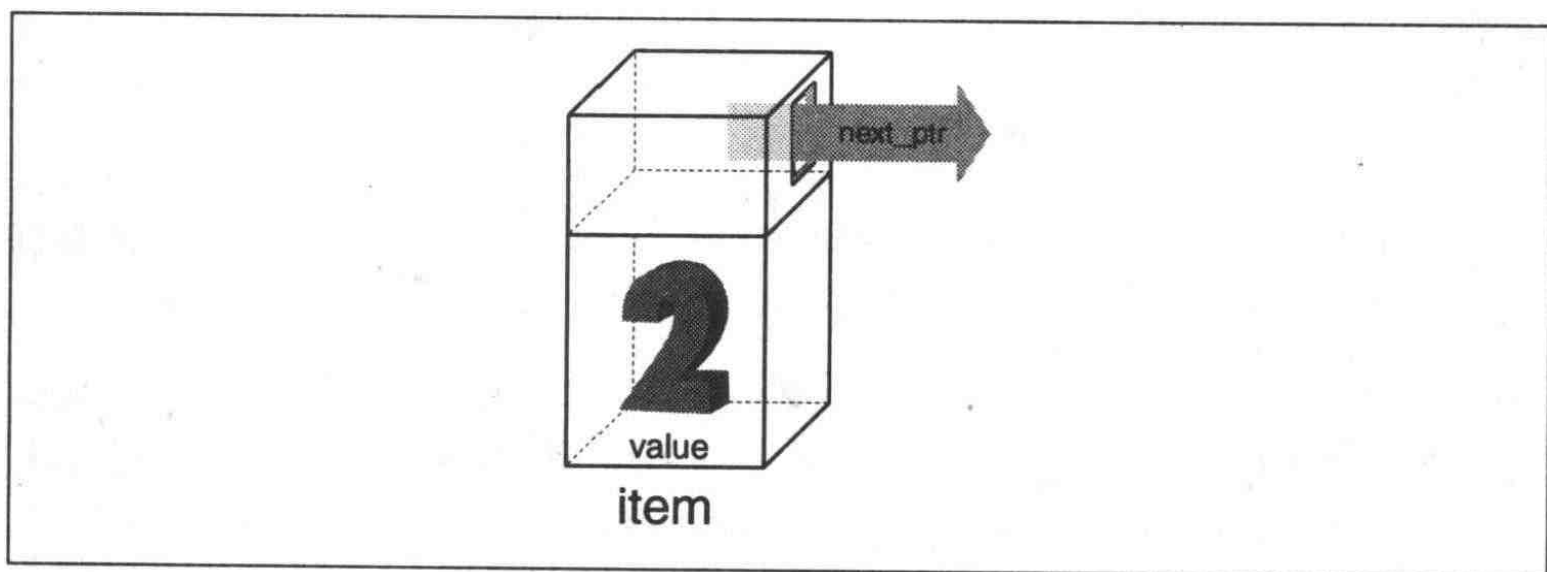


图 20.2 一个item

new运算符为一个变量分配存储并返回一个指针。它用于在不大的地方（实际上是在一个称为堆的内存空间）中创建新的物体。到目前为止，我们仅用指针指向一个有名变量。所以如果我们使用下面这样的语句：

```
int data;  
int *number_ptr;  
  
number_ptr = &data;
```

指针指向的就是一个有名（**data**）变量。运算符**new**建立一个新的无名变量，并返回一个指向该变量的指针。**new**创建的“物体”可以只通过指针来引用，而从不使用名字。

在下面的例子中，我们用**new**从堆中分配一个整数。变量**element_ptr**将指向这个新整数。

```
int *element_ptr;    // Pointer to an integer  
  
element_ptr = new int;    // Get an integer from the heap
```

运算符**new**带有一个自变量：分配项的类型。根据最新的C++标准，如果内存都被占满

了，`new`将进行例外处理，中断程序的执行。（处理的细节请详见第22章“例外”。）在早期C++系统中，当内存没有时，`new`返回一个空指针。

假定我们使用一个复杂的数据库，它含有邮件表（还有其它的项目）。我们想让存储空间占用得最少，所以只有当一个项真正存在时，我们才分配真正的空间。建立一个`person`类数组的话，要静态分配数据，占用的空间太多了。所以我们按需分配空间。一个`person`结构是：

```
class person {
    public:
        char    name[30];           // Name of the person
        char    address[30];        // Where he lives
        char    city_state_zip[30]; // Part 2 of address
        int     age;                 // His age
        float   height;              // His height in inches
}
```

我们想为这个`person`分配空间。然后指向这个记录的指针将放在数据库中。

要创建一个新的`person`，使用下面的语句：

```
struct person *new_ptr;
```

```
new_ptr = new person;
```

运算符`new`还可以分配更复杂的数据类型，如数组。示例20.1为一个字符数组分配空间，数组中每个字符有80个字节长（包括'\0'）。变量`string_ptr`指向这个数组。

示例20.1

```
main()
{
    char *string_ptr;

    string_ptr = new char[80];
```

我们所要做的就是简单类型（如`person`）换为数组说明（`char[80]`）。

delete运算符

运算符`new`从堆中得到内存。要把内存归还给堆则使用运算符`delete`。`delete`运算符的一般格式为：

```
delete pointer;    // Where pointer is a pointer to a simple object
pointer = NULL;
```

这里，`pointer`是前面用`new`分配的一个指针。如果`new`运算符分配的是一个数组的话，

则你可以用下面的形式:

```
delete pointer[];    // Where pointer is a pointer to an array
pointer = NULL;
```

注 意

`delete`运算符之所以有两种形式,是因为C++没有办法区分一个指向对象的指针和一个指向对象数组的指针。`delete`运算符指望程序员使用“[]”来区别两者。

严格地讲,下面这一行:

```
pointer = NULL;
```

不是必需的。但是,在删除指向的内容后,把指针置空是个好习惯。这样做后,你就不会再企图用指针去删除内存了,而且还可避免你两次删除同一内存的情况发生。

下面是使用`new`得到存储,用`delete`释放存储的例子。

```
const DATA_SIZE = (16 * 1024);

void copy(void)
{
    char *data_ptr;    // Pointer to large data buffer

    data_ptr = new char[DATA_SIZE];    // Get the buffer

    /*
     * Use the data buffer to copy a file
     */
    delete[] data_ptr;
    data_ptr = NULL;
}
```

如果我们忘了释放内存会怎么样呢?缓冲区会死掉。因为,内存管理系统认为它仍在使用,但是没有程序在使用它。(描述这种现象的技术术语是“内存渗漏”。)如果从`copy`函数中删除`delete`语句,那么每次成功地调用这个函数都会吃掉16K的内存。

当使用已被释放的内存时,可能会发生其它的问题。使用`delete`时,内存返回给存储池,并可再次使用。在调用`delete`后再使用指针,类似于一个数组下标越界错误,你使用的是属于别的程序的内存。这可能会得出意想不到的结果,或者使程序崩溃。

链 表

假定你在写一个程序,要使用通信线路把一串名字送给另一台计算机。操作员一整天

都在录入这些名字，然后拨号到另一台计算机，并发送这些名字。问题是，你不能预先得知有多少名字要录入。使用链表结构，就可以建立一串名字；每录入一个名字，链表就变长一点。使用链表，你还可以在链表的中间方便地插入一个名字（如果用数组的话，这样的操作又慢又困难）。而且，正如你在后面将看到的，链表可以与其它数据结构联合使用，以处理更复杂的数据。

链表是一连串的项，每一项都指向串中的下一项。想想儿时玩的找财宝的游戏。你得到一个提示说，“在邮箱中找。”你跑到邮箱前，又发现下一条现索，“在后院的大树下找”。这样一直下去，直到你找到财宝（或者迷路）为止。在这个游戏中，每条现索都指向下一条。

图20.3中用图说明了一个链表。

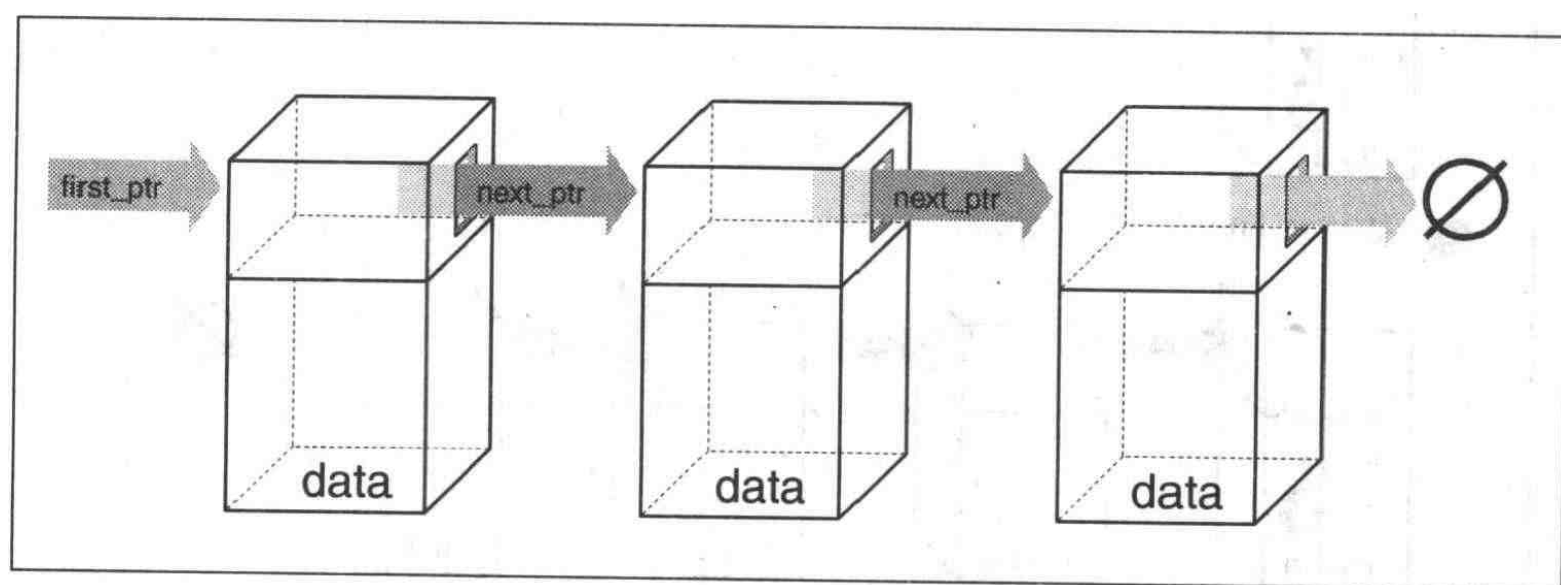


图 20.3 链表

链表中的类说明如下：

```
class linked_list {
public:
    class linked_list_element{
    public:
        char    data[30];          // Data in this element
    private:
        linked_list_element *next_
        ptr; // Pointer to next element
        friend class linked_list;
    };

public:
    linked_list_element *first_ptr;    // First element in the list

    // Initialize the linked list
    linked_list(void) {first_ptr = NULL;}
}
```

```

// ... Other member functions
};

```

变量`first_ptr`指向表中的第一个元素。开始，在向表（它是空的）中插入元素之前，该变量被初始化为`NULL`。

图20.4说明了如何在表头插入一个新元素。现在，我们要做的就是把它表示为C++代码。

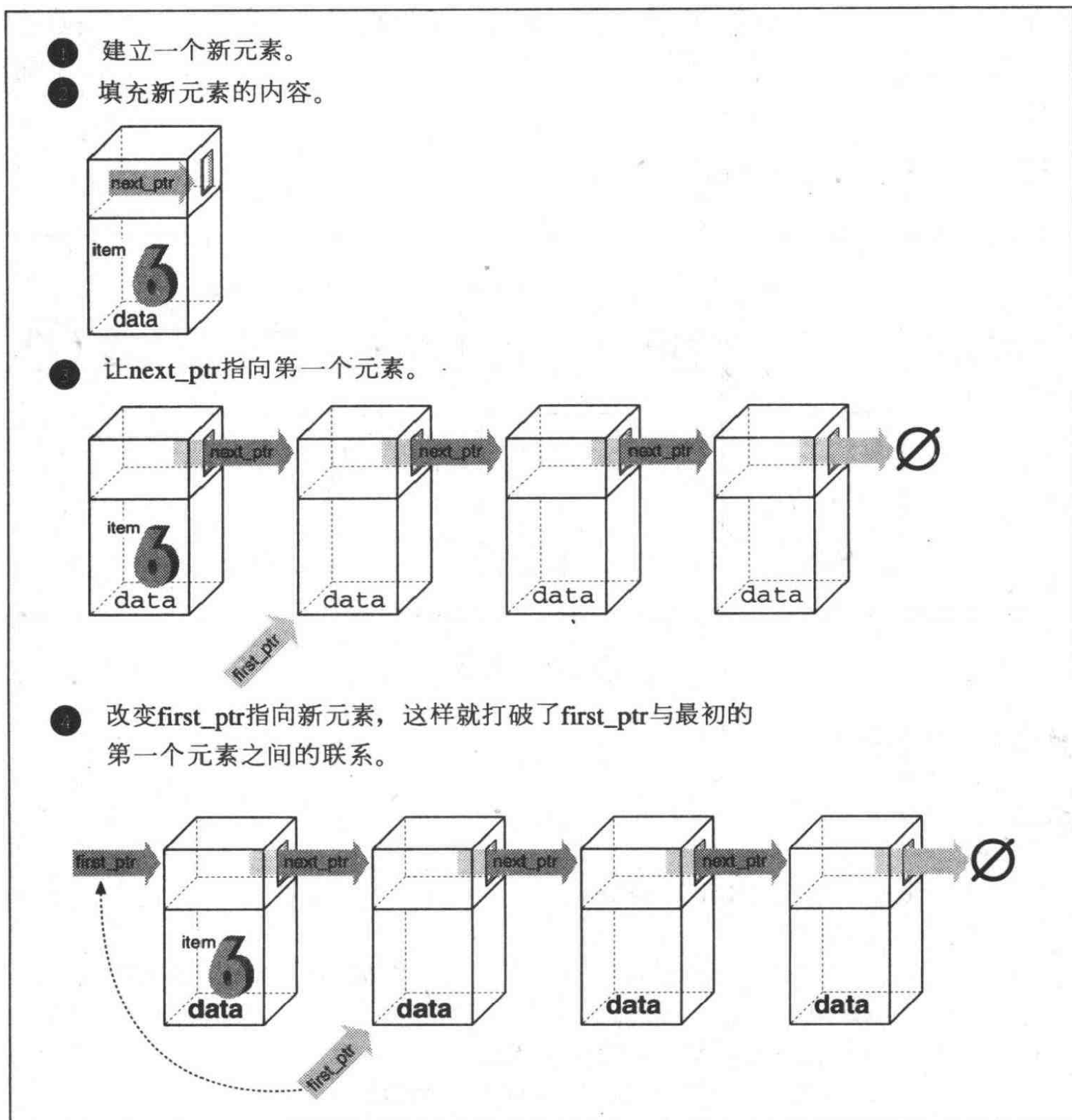


图20.4 新元素

要用C++实现这个过程，需完成下面各步：

1. 建立一个要插入的项。

```
new_ptr = new linked_list;
```

2. 填充项中各数据项的内容。

```
(*new_ptr).data = item;
```

3. 让表中的第一个元素指向新元素。

```
(*new_ptr).next_ptr = first_ptr;
```

4. 现在，新元素是第一个元素。

```
first_ptr = new_ptr;
```

实际的程序如下：

```
void linked_list::add_list(int item)
{
    // Pointer to the next item in the list
    linked_list_element *new_ptr;

    new_ptr = new linked_list;
    strcpy((*new_ptr).data, item);
    (*new_ptr).next_ptr = first_ptr;
    first_ptr = new_ptr;
}
```

现在我们已经能在表中插入项了，让我们利用这个功能。编写一个短小的函数，在表中查找一个项，或者表中不存在这个项。示例20.2中是新的find函数。

示例20.2 *find/find.cc*

```
#include <iostream.h>
#include <string.h>
#include "linked.h"
/*****
 * Find -- look for a data item in the list
 *
 * Parameters
 *     name -- name to look for in the list
 *
 * Returns
 *     1 if name is found
 *     0 if name is not found
 *****/
int linked_list::find(char *name)
{
    /* Current structure we are looking at */
    linked_list_element *current_ptr;

    current_ptr = first_ptr;
```

```

while ((strcmp(current_ptr->data, name) != 0) &&
      (current_ptr != NULL))
    current_ptr = current_ptr->next_ptr;

/*
 * If current_ptr is null, we fell off the end of the list and
 * didn't find the name
 */
return (current_ptr != NULL);
}

```

问题20.1: 为什么运行该程序时，有时会得到一个错误？而有时，对一个不在表中的项，它会报告“找到”（返回1）了呢？

在find程序中，我们使用了一个很笨的写法(*current_ptr).data来访问结构中的数据域。C++为此提供了一种简便的写法，即使用->运算符。点(.)表示的是结构中的域，结构指针运算符(->)表示的是结构指针中的域。

下面两个表达式是等价的：

```

(*current_ptr).data = value;
current_ptr->data = value;

```

顺序链表

至此，我们仅向表头添加了新元素。假定我们想按顺序添加元素。图20.5是一个顺序链表的例子。

图20.6 显示的是向表中添加新元素“53”所需的各步。

下面的成员函数实现了这个算法。第一步是确定要插入的位置。first_ptr指向表的第一个元素。变量before_ptr沿表移动，直到它找到合适的插入位置为止。变量after_ptr是指向前一个位置的指针。要插入的新元素位于这两个元素之间。

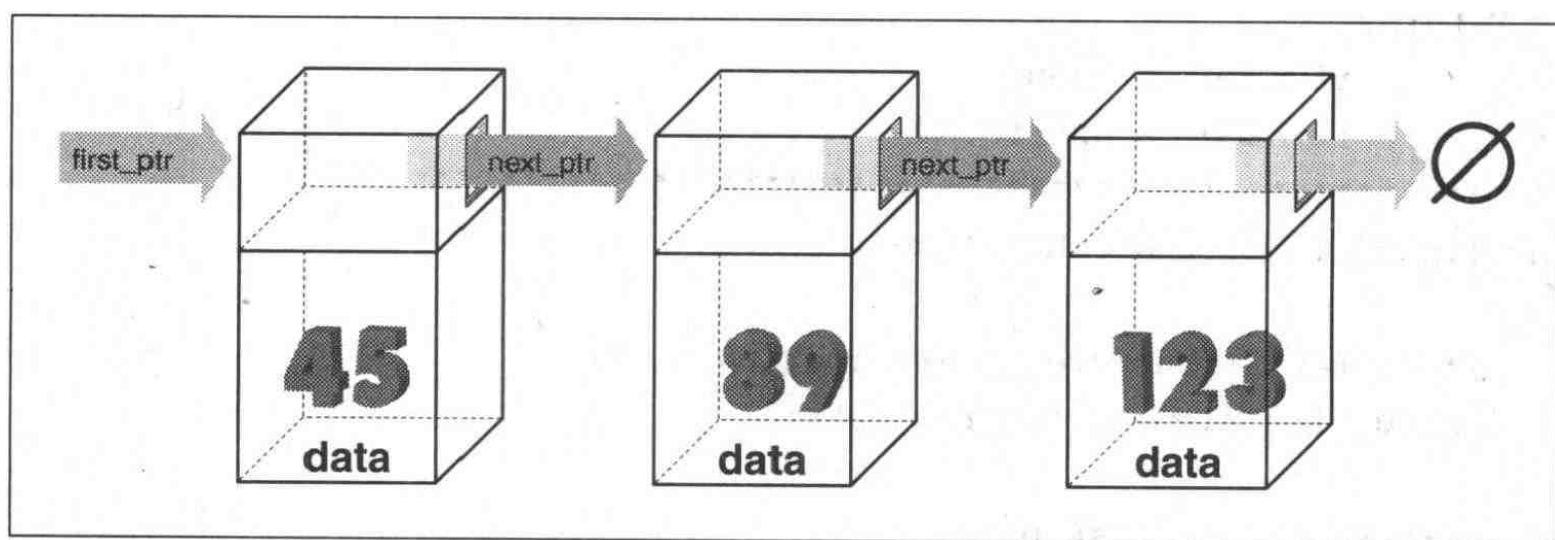


图 20.5 有序表

```

void linked_list::enter(int item)
{
    item *before_ptr; // Insert before this element
    item *after_ptr;  // Insert after this element

    /*
     * Warning: This routine does not take
     * care of the case where the element is
     * inserted at the head of the list
     */

    before_ptr = first_ptr;
    while (1) {
        before_ptr = insert_ptr;
        insert_ptr = insert_ptr->next_ptr;

        // Did we hit the end of the list?
        if (insert_ptr == NULL)
            break;

        // Did we find the place?
        if (item >= insert_ptr->data)
            break;
    }
}

```

现在，我们知道了新元素的插入位置，剩下的就是插入它。我们从新元素的前指针（before_ptr）入手。该指针应该指向新元素，所以：

```
before_ptr->next_ptr = new_ptr;
```

下一个是新元素（new_ptr）。它要指向其后的元素，即after_ptr。由下列代码来实现：

```
new_ptr->next_ptr = after_ptr;
```

元素after_ptr需要指向其余的链。因为它已经是这样了，所以不用再做什么处理。插入新元素的完整代码如下：

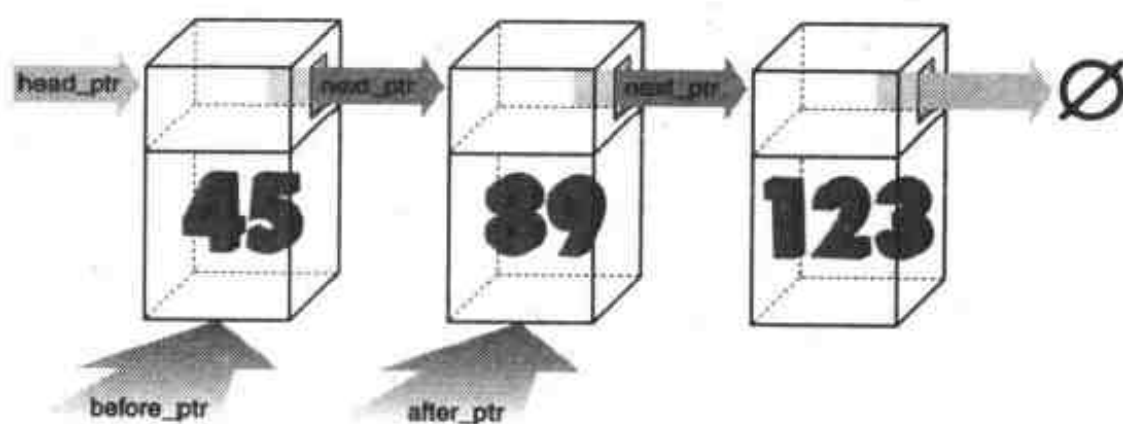
```

// Create new item
new_ptr = new list;
new_ptr->data = item;

// Link in the new item
before_ptr->next_ptr = new_ptr;
new_ptr->next_ptr = after_ptr;

```


- before_ptr 指向插入点前的元素，after_ptr指向指向插入点后的元素。



- 建立新元素。



- 让新元素的next_ptr 指针指向after_ptr所指的地方。
- 修改before_ptr->next_ptr, 让before_ptr所指的元素与新元素链接起来。

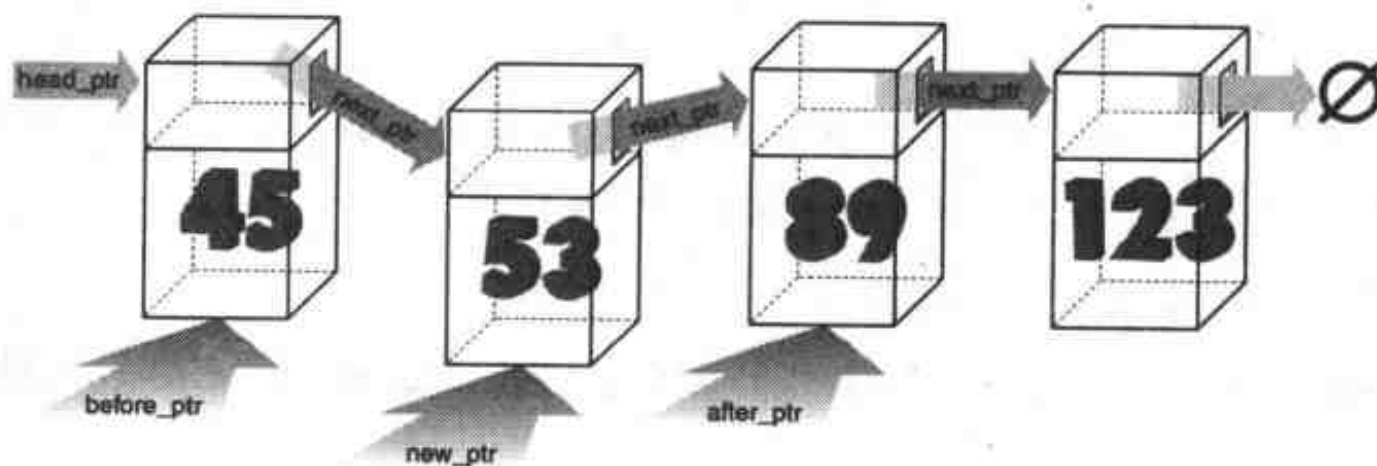


图 20.6 向一个有序表中添加元素“53”

双向链表

双向链表包含两个链。一个链从前向后指，另一个链从后向前指。当程序需要向前和向后遍历链表时，双向链表非常有用。

双向链表的类如下：

```
class double_list {
```

```

private:
    class double_list_element {
    public:
        int data;          // Data item
    private:
        double_list_element *next_ptr;    // Forward link
        double_list_element *previous_ptr; // Backward link
    friend class double_list;
    };
public:
    double_list_element *head_ptr;    // Head of the list

    double_list(void) {head_ptr = NULL;}

    // ... Other member functions

```

图20.7中用图表示了它们的关系。

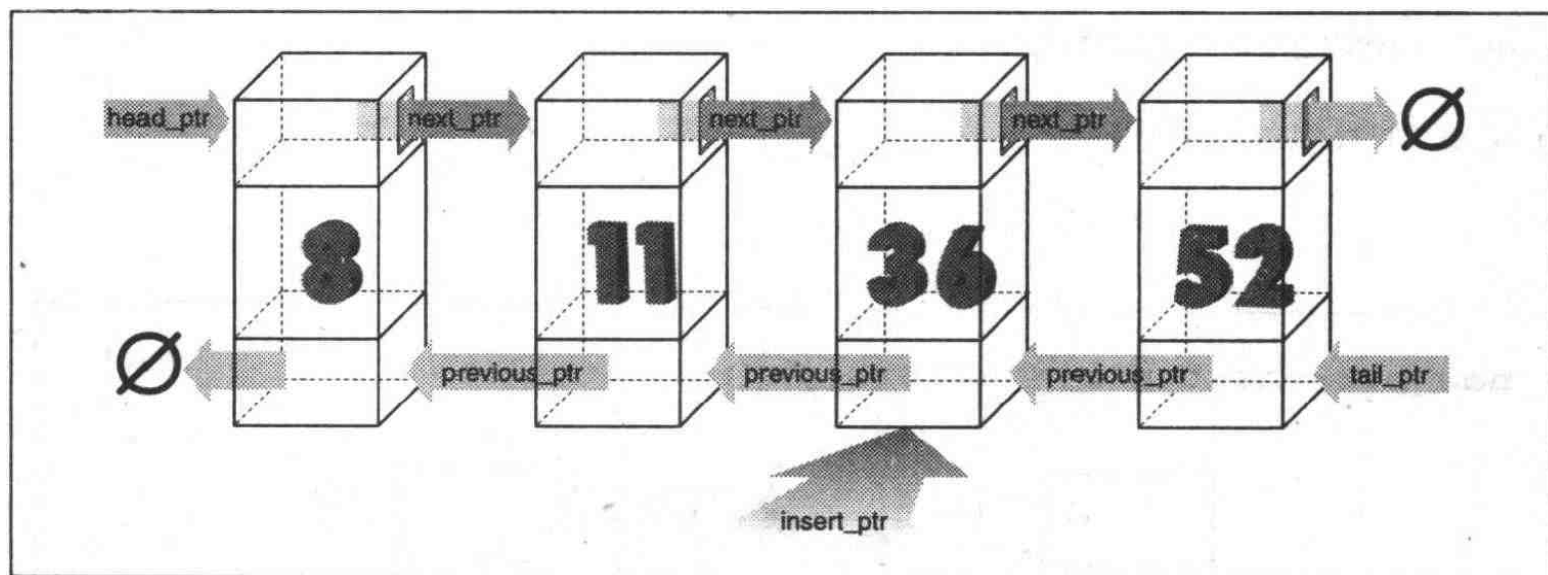


图20.7 双向链表

要在表中插入一项，先要定位插入指针的位置：

```

void double_list::enter(int item)
{
    double_list_element *insert_ptr; // Insert before this element

    /*
     * Warning: This routine does not take
     * care of the case where the element is
     * inserted at the head of the list
     * or the end of the list
     */

    insert_ptr = head_ptr;
    while (1) {

```



```

insert_ptr = insert_ptr->next;

// Have we reached the end?
if (insert_ptr == NULL)
    break;

// Have we reached the right place?
if (item >= insert_ptr->data)
    break;
}

```

注意，我们不用保留变量`after_ptr`的历史值。指针`insert_ptr->previous_ptr`指向前一个元素的位置。要插入新元素，必须调整两组指针。首先创建新元素：

```

// Create new element
new_ptr = new double_list;

```

下一步，设置新元素的后指指针：

```

new_ptr->next_ptr = insert_ptr;

```

这个过程用图20.8来说明。

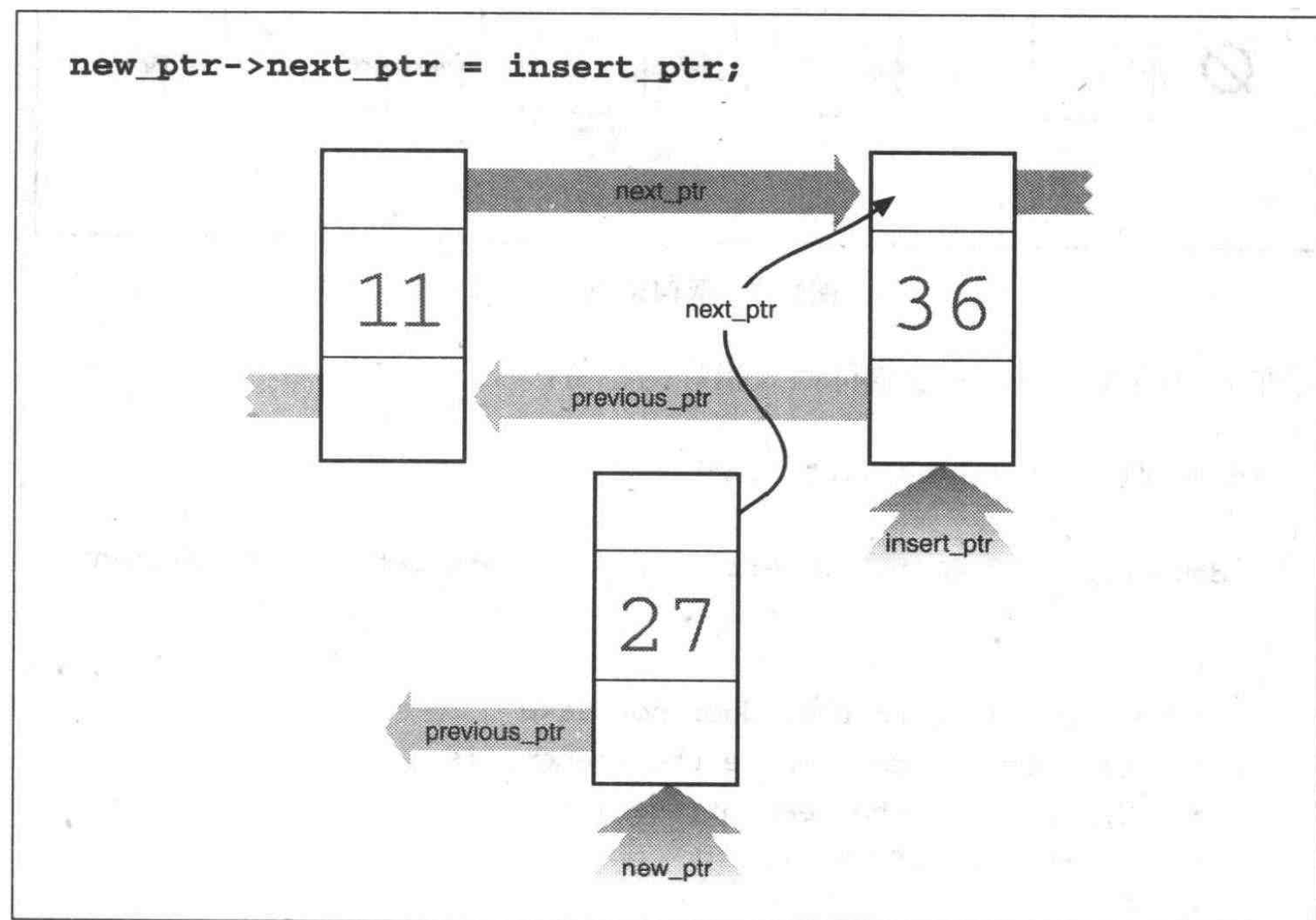


图 20.8 双向链表插入的第一部分

下面，与前一个元素相连：

```
new_ptr->previous_ptr = insert_ptr->previous_ptr;
```

这个过程用图20.9来说明。

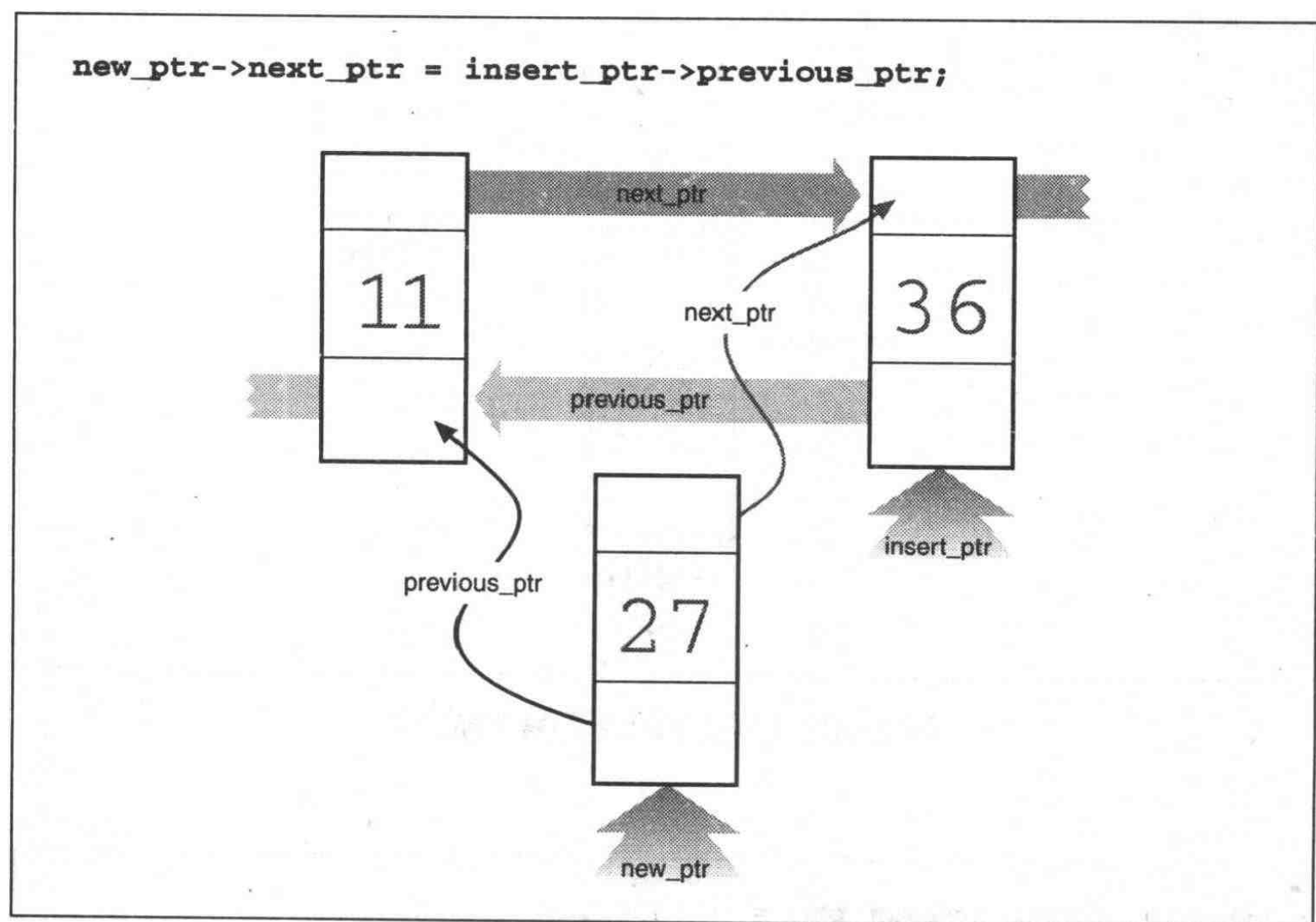


图 20.9 双向链表插入的第二部分

新元素的链已经设置完毕。现在要在旧链表的11和36两项之间断开，把它们与新项（27）连接在一起。

找到项11需要一点技巧，我们只有指向项36的指针（insert_ptr）。不过如果我们沿着前指链向前（insert_ptr->previous_ptr），就可以得到想要的项（11）。现在我们把next_ptr改为指向该项的指针。

做这件事情，C++代码出奇地简单：

```
insert_ptr->previous_ptr->next_ptr = new_ptr;
```

实际上，我们可以在图20.10中看到这个操作。

我们还需修改一个指针：insert_ptr的previous_ptr。在C++中，用下面的语句来实现：

```
insert_ptr->previous_ptr = new_ptr;
```

这个操作过程见图20.11。

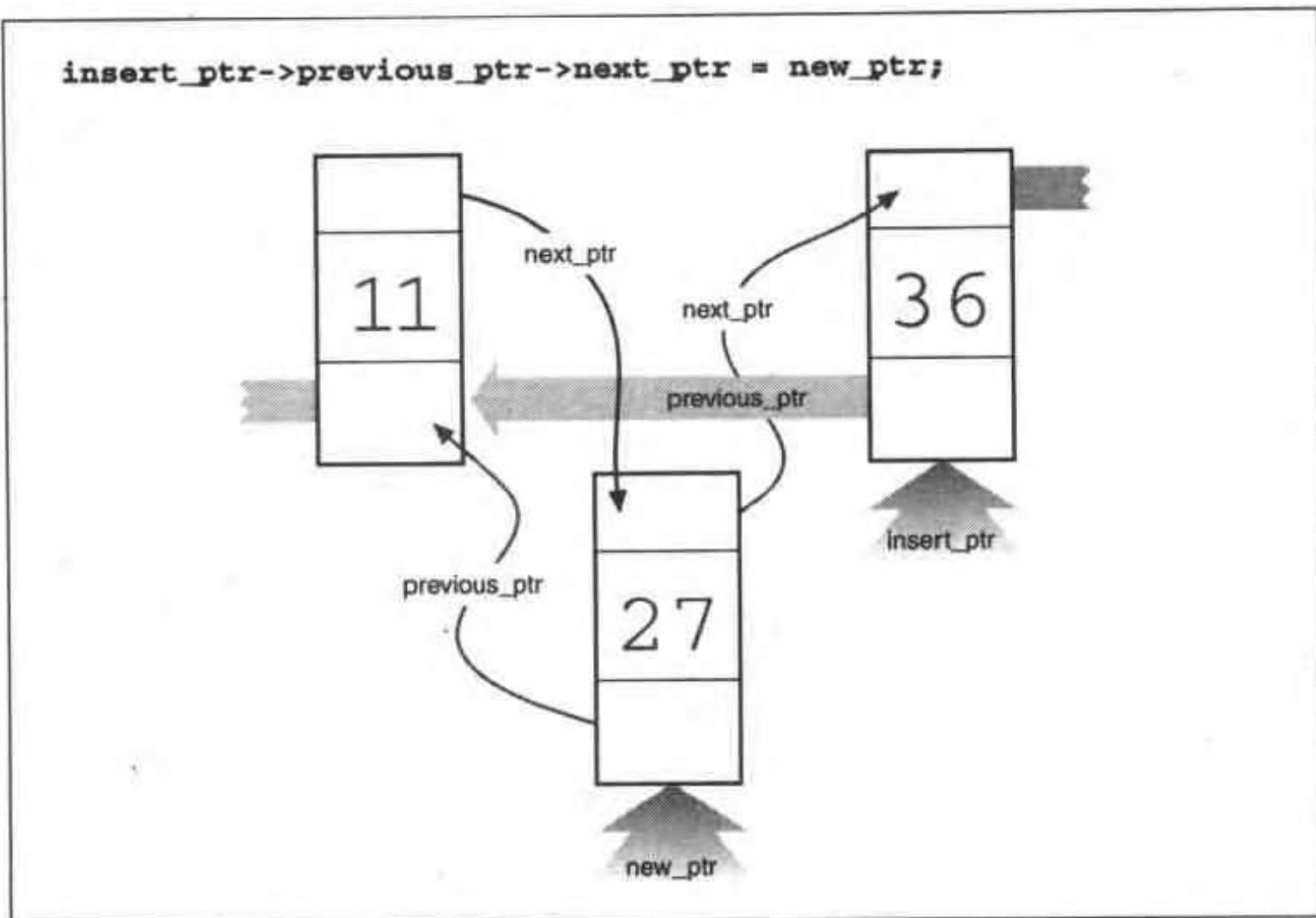


图 20.10 双向链表插入的第三部分

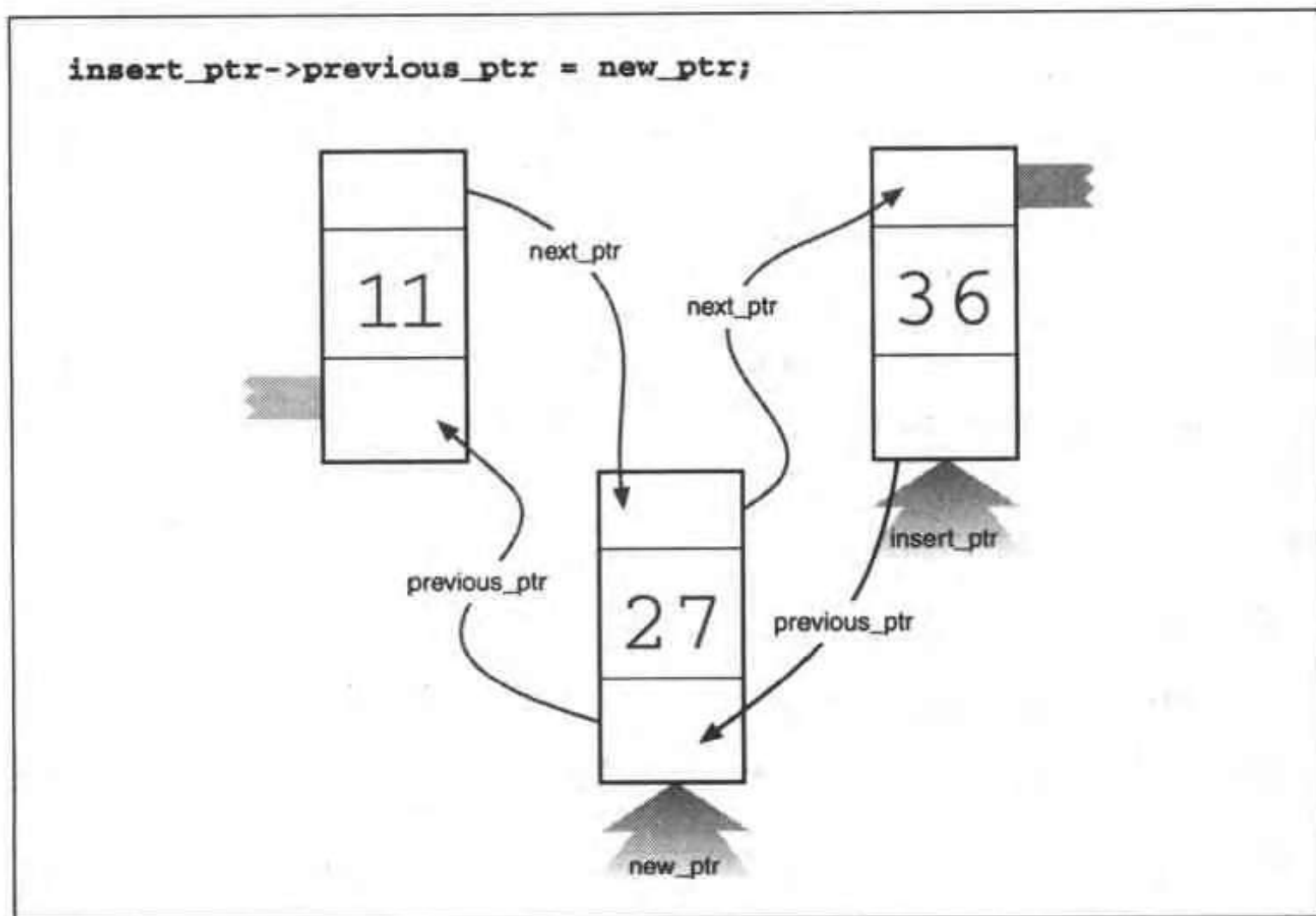


图20.11 双向链表插入的第四部分

总之，要往双向链表中插入一个新项，你必须重置四个链的值：

1. 新项的前指指针：

```
new_ptr->previous_ptr = insert_ptr->previous_ptr;
```

2. 新项的后指指针：

```
new_ptr->next_ptr = insert_ptr;
```

3. 新项后面一项的前指指针：

```
insert_ptr->previous_ptr->next_ptr = new_ptr;
```

4. 新项前面一项的后指指针：

```
insert_ptr->previous_ptr = new_ptr;
```

树

假定你想建立一个出现在一篇文章中所有字的字表，我们可以用一个链表来实现。但是链表的查找很慢，因为必须检查每个元素，直到找到正确的插入位置为止。使用一种称为树的数据类型，可以减少比较的次数。

一个二分树结构如图20.12所示。

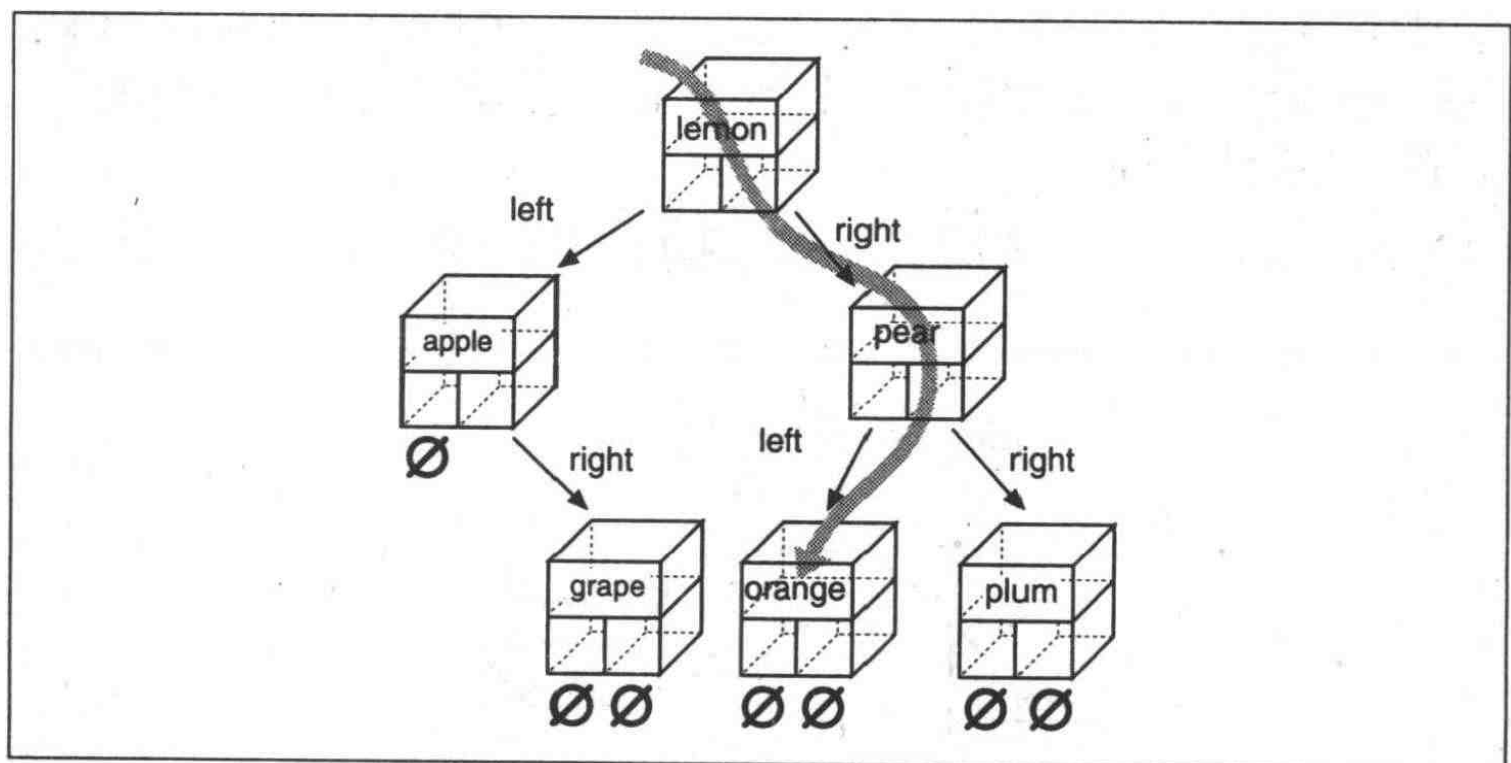


图20.12 树

每个盒子称为树的一个节点。最上面的一个称为树的根，最下面的称为叶子*。每个节点有两个指针：一个左指针和一个右指针，它们分别指向左右子树。

*程序设计中的树有别于自然树，根画在上面，叶子画在下面。从常识可知这是颠倒的。万一你没有注意到也没关系，常识不会对编程有太大的帮助的。

树的结构为:

```
class tree {
    class node {
    public:
        char *data;    // Word for this tree
    private:
        node *right;   // Tree to the right
        node *left;    // Tree to the left
        friend class tree;
    };
public:
    node *root; // Top of the tree (the root)

    tree(void) {root = NULL;};
    // ... Other member function
};
```

树经常用来存储一张符号表(程序中使用的变量表)。本章中,我们将用树来存储一个字表,然后按字母顺序打印该表。树优于链表的特性是,它的查找时间明显地少。

在这个例子中,每个节点存储一个字。左子树存储的所有字小于当前字,而右子树存储的所有字大于当前字。

例如,图20.13是我们在树中查找字“orange”的过程。我们从根“lemon”开始。因为“orange” > “lemon”,所以顺右链向下,找到“pear”。因为“orange” < “pear”,又沿左链向下,在这儿找到“orange”。

在树中,递归很适用。递归的规则是:1) 函数必须使事情变得更简单,2) 必须有结

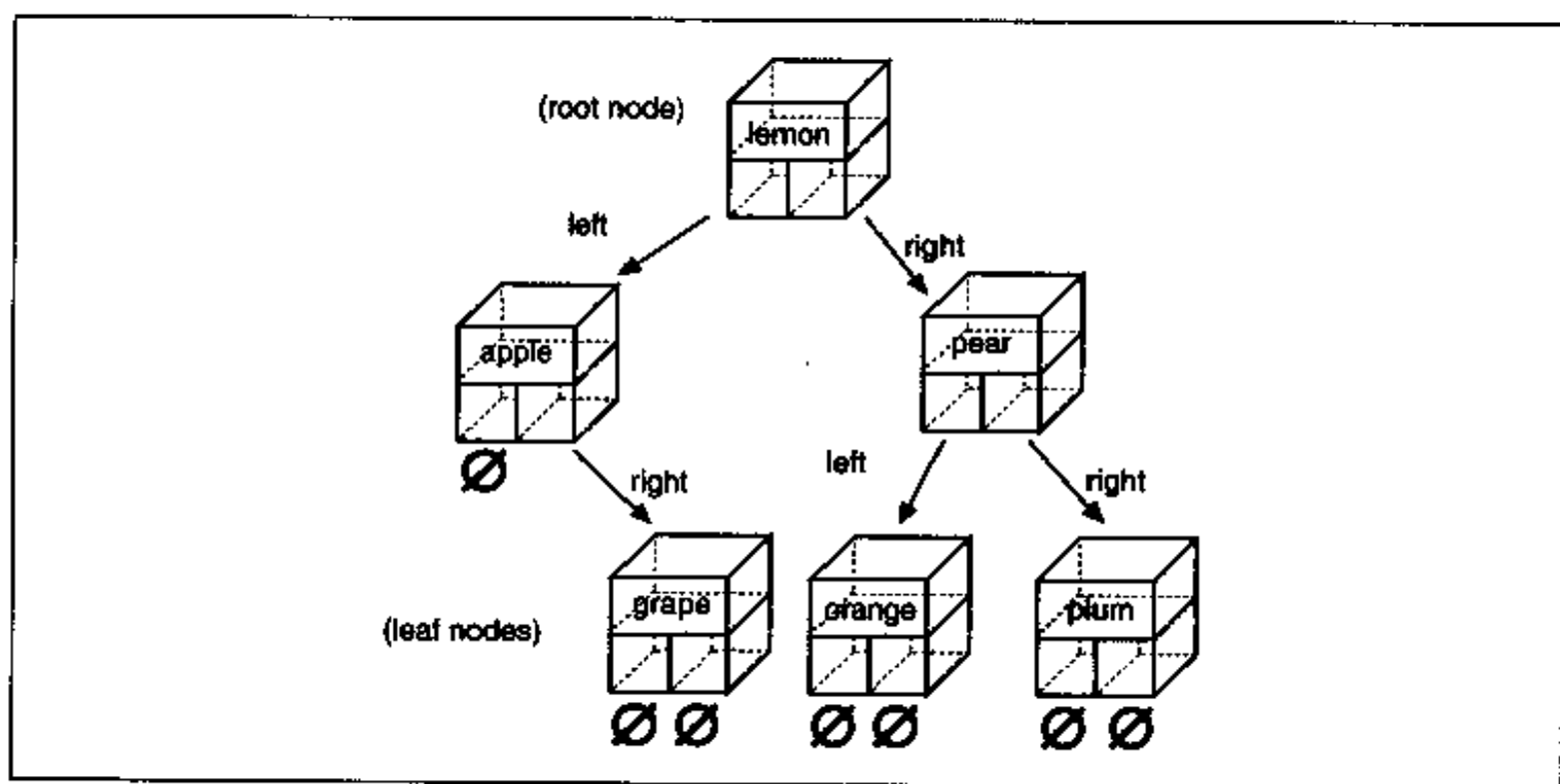


图 20.13 树查找



束点。

在树中插入一个字的算法是：

1. 如果树（或子树）为空，用该字建立单节点的树。
2. 如果该节点含有这个字，不做任何事情。
3. 否则，根据这个字的值，把该字插到左或右子树中。

这个算法是否满足递归规则？函数有两个结束点：

1. 找到匹配。
2. 是空节点。

否则，我们把该字插到子树中（这比整个树要简单一些）。

我们看看它是怎么工作的。考虑当我们在树中插入“fig”时会发生什么事情。首先，我们比较“fig”与“lemon”。“fig”小，所以让它与“apple”进行比较，因为“fig”大，又去比较“grape”。因为“fig”比“grape”要小，所以又去比较左链。左链是NULL，这样要创建一个新节点。

该节点利用了新函数strdup。这个函数在堆中创建一个串的拷贝，并返回指向新串的指针。稍后，用delete运算符可以把串返回给堆。*

在树中插入值的函数如下：

```
void tree::enter_one(node *&node, char *word)
{
    int result;          // Result of strcmp

    // See if we have reached the end
    if (node == NULL) {
        node = new node;

        node->left = NULL;
        node->right = NULL;
        node->word = strdup(word);
    }
    result = strcmp(node->word, word);
    if (result == 0)
        return;
```

* 函数strdup不是推荐的C++ ANSI标准的一部分。但它在我所见过的编译器中都能使用。它是不成文标准的一部分。


```

        if (result < 0)
            enter(node->right, word);
        else
            enter(node->left, word);
    }

```

函数的调用如下：

```

void tree::enter(char *word {
    enter_one(root, word);
};

```

该函数把一个指向树根的指针传给enter_one。如果根为NULL，则enter_one建立一个节点。因为我们改变了一个指针的值，所以必须传送指针的引用。

树的打印

虽然树的结构很复杂，但它的打印却很容易。这里我们再一次使用递归。打印算法如下：

1. 对空树，不打印任何内容。
2. 打印该节点前面的数据（左子树）。
3. 打印该节点。
4. 打印该节点后面的数据（右子树）。

树的打印代码如下：

```

void tree::print_one(node *top)
{
    if (top == NULL)
        return;                // Short tree

    print_tree(top->left);
    cout << top->word << '\n';
    print_tree(top->right);
}

void tree::print(void) {
    print_one(root);
}

```



程序的其余部分

现已定义了数据结构，余下的还有几个函数。`main`函数检查自变量个数，并调用扫描程序和`print_tree`程序。

扫描函数读入文件，把它分为一个个的字。它使用标准的宏`isalpha`。如果自变量是一个字母，宏返回1，否则返回0。它定义在标准的包含文件`ctype.h`中。找到一个字后，调用`enter`函数，把这个字放到树中。`strdup`在堆中为一个串建立空间，并返回指向它的指针。

示例20.3中列出`words.cc`中的源代码。

示例20.3 `words/words.cc`

```
/* *****  
 * Words -- scan a file and print out a list of words *  
 *           in ASCII order *  
 * * * * *  
 * Usage: *  
 * words <file> *  
 * * * * */  
  
#include <iostream.h>  
#include <fstream.h>  
#include <ctype.h>  
#include <string.h>  
#include <stdlib.h>  
  
class tree {  
    private:  
        // The basic node of a tree  
        class node {  
            private:  
                node    *right;    // Tree to the right  
                node    *left;     // Tree to the left  
            public:  
                char    *word;     // Word for this tree  
  
            friend class tree;  
        };  
  
        // The top of the tree  
        node *root;  
  
        // Enter a new node into a tree or subtree  
        void enter_one(node *&node, char *word);  
};
```



```

        // Print a single node
        void print_one(node *top);
public:
        tree(void) {root = NULL;}

        // Add a new word to the tree
        void enter(char *word) {
                enter_one(root, word);
        }

        // Print the tree
        void print(void) {
                print_one(root);
        }
};

static tree words;      // List of words we are looking for

/*****
 * Scan -- scan the file for words
 *
 * Parameters
 *      name -- name of the file to scan
 *****/
void scan(char *name)
{
        char word[100];    // Word we are working on
        int  index;        // Index into the word
        int  ch;           // Current character
        ifstream in_file;  // Input file

        in_file.open(name, ios::in);
        if (in_file.bad()) {
                cerr << "Error: unable to open " << name << '\n';
                exit(8);
        }
        while (1) {
                // Scan past the white space
                while (1) {
                        ch = in_file.get();

                        if (isalpha(ch) || (ch == EOF))
                                break;
                }

```



```

        if (ch == EOF)
            break;

    word[0] = ch;
    for (index = 1; index < sizeof(word); ++index) {
        ch = in_file.get();
        if (!isalpha(ch))
            break;
        word[index] = ch;
    }
    // Put a null on the end
    word[index] = '\0';

    words.enter(word);
}

main(int argc, char *argv[])
{
    if (argc != 2) {
        cerr << "Error: wrong number of parameters\n";
        cerr << "      on the command line\n";
        cerr << "Usage is:\n";
        cerr << "      words 'file'\n";
        exit(8);
    }
    scan(argv[1]);
    words.print();
    return (0);
}

/*****
 * tree::enter_one -- enter a word into the tree      *
 *                                                    *
 * Parameters                                          *
 *      new_node -- current node we are looking at   *
 *      word -- word to enter                        *
 *****/
void tree::enter_one(node *&new_node, char *word)
{
    int result;          // Result of strcmp

    // See if we have reached the end
    if (new_node == NULL) {
        new_node = new node;
    }

```

```

        new_node->left = NULL;
        new_node->right = NULL;
        new_node->word = strdup(word);
    }
    result = strcmp(new_node->word, word);
    if (result == 0)
        return;

    if (result < 0)
        enter_one(new_node->right, word);
    else
        enter_one(new_node->left, word);
}

/*****
 * tree::print_one -- print out the words in a tree
 *
 * Parameters
 *   top -- the root of the tree to print
 *****/
void tree::print_one(node *top)
{
    if (top == NULL)
        return;                // Short tree

    print_one(top->left);
    cout << top->word << '\n';
    print_one(top->right);
}

```

问题20.2: 有一次我编写了一个程序，它使用树结构把字典读入内存，然后在程序中用它来查找拼写错误的字。虽然认为树的操作很快，但是这个程序运行得非常慢，你会误认为我使用的是链表。为什么？

提示: 画图，用“able”、“baker”、“cook”、“delta”和“easy”构造一棵树，看看结果如何。

象棋程序中用到的数据结构

人工智能中经典的问题是象棋游戏。到目前为止，虽然计算机科学已经很先进了，但还没有一个人能写出打败所有象棋大师的程序。

我们为象棋程序设计一个数据结构。下棋时，总有几步棋可走。你的对手可能也有许多步，对每一步，你又有许多下法。依此类推，来来回回就有了下法的不同层。



我们的数据结构看上去很像是一棵树。但它不是二分树，因为每一个节点都有两个以上的分支（图20.14）。

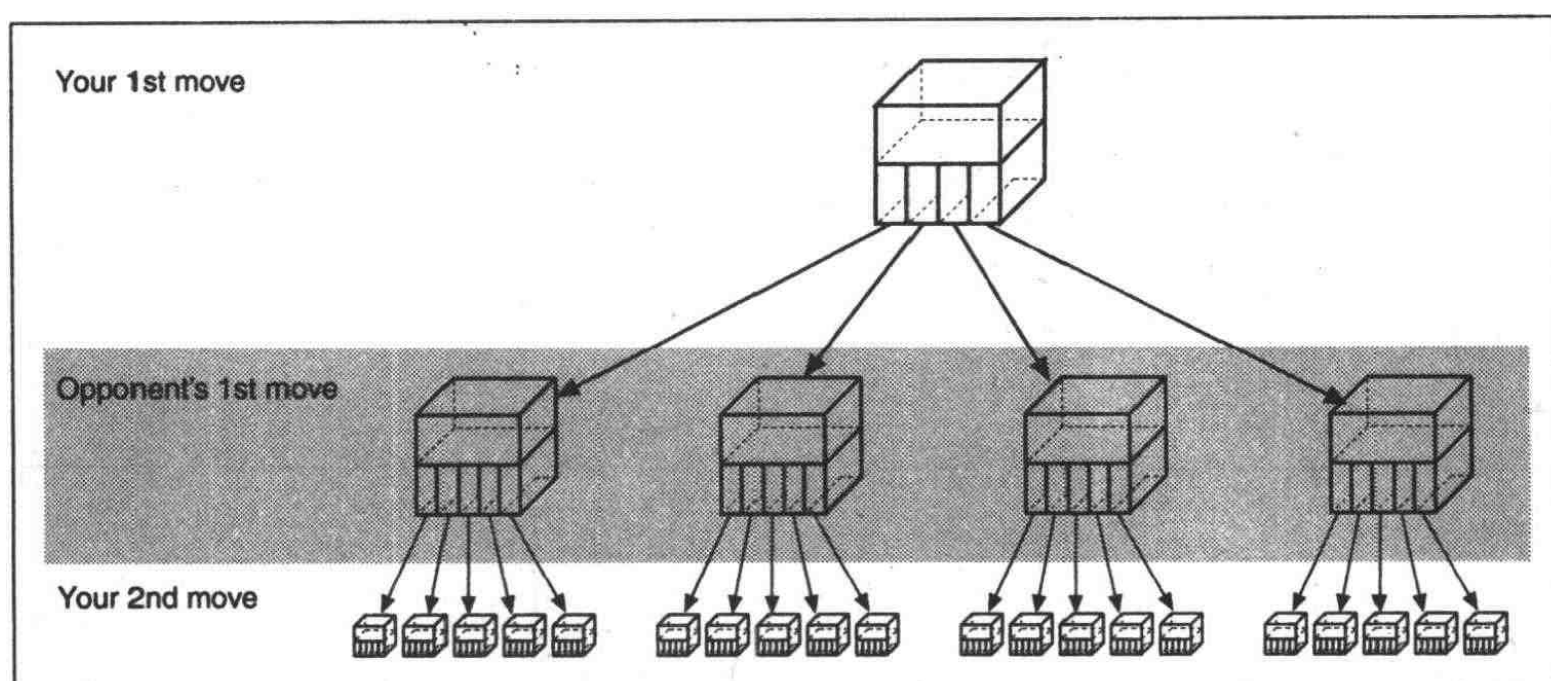


图 20.14 象棋树

我们使用下面的数据结构：

```
class chess {
public:
    class board_class board; // Current board position
    class next_class {
        class move_class move; // Our next move
        class chess *chess_ptr;
        // Pointer to the resulting position
    } next[MAX_MOVES];
};
```

问题是，对一个给定的棋局，可以走动的数目是不定的。例如，开局时你有很多步可走。*像车、后和象可以走到一条直线上的任何格子内。当你到残局时（在一场势均力敌的比赛中），双方可能只剩下几个兵和一个主要的棋子。可走的选择就大为减少了。

我们想尽可能地提高存储效率，因为下棋程序已用到了机器的极限。我们把下一步可走棋数组变为链表结构，就可以减少存储需求。最终的结构为：

```
class next_class {
    class move_class move; // Our next move
    class next_chess *chess_ptr; // Pointer to the resulting position
```

* 琐碎的问题：下棋开盘时，你可以走的21步棋是什么？（你可以让每个兵向前走一步（8种）或两步（又有8种），马可以向左向右跳起（又有4种）（8+8+4=20）。）什么是第21步棋？

```
};

struct chess {
    class board_class board;      // Current board position
    class next_class *list_
        ptr;      // List of moves we can make from here
    class next_class this_move;  // The move we are making
};
```

我们用图20.15表示这个结构。

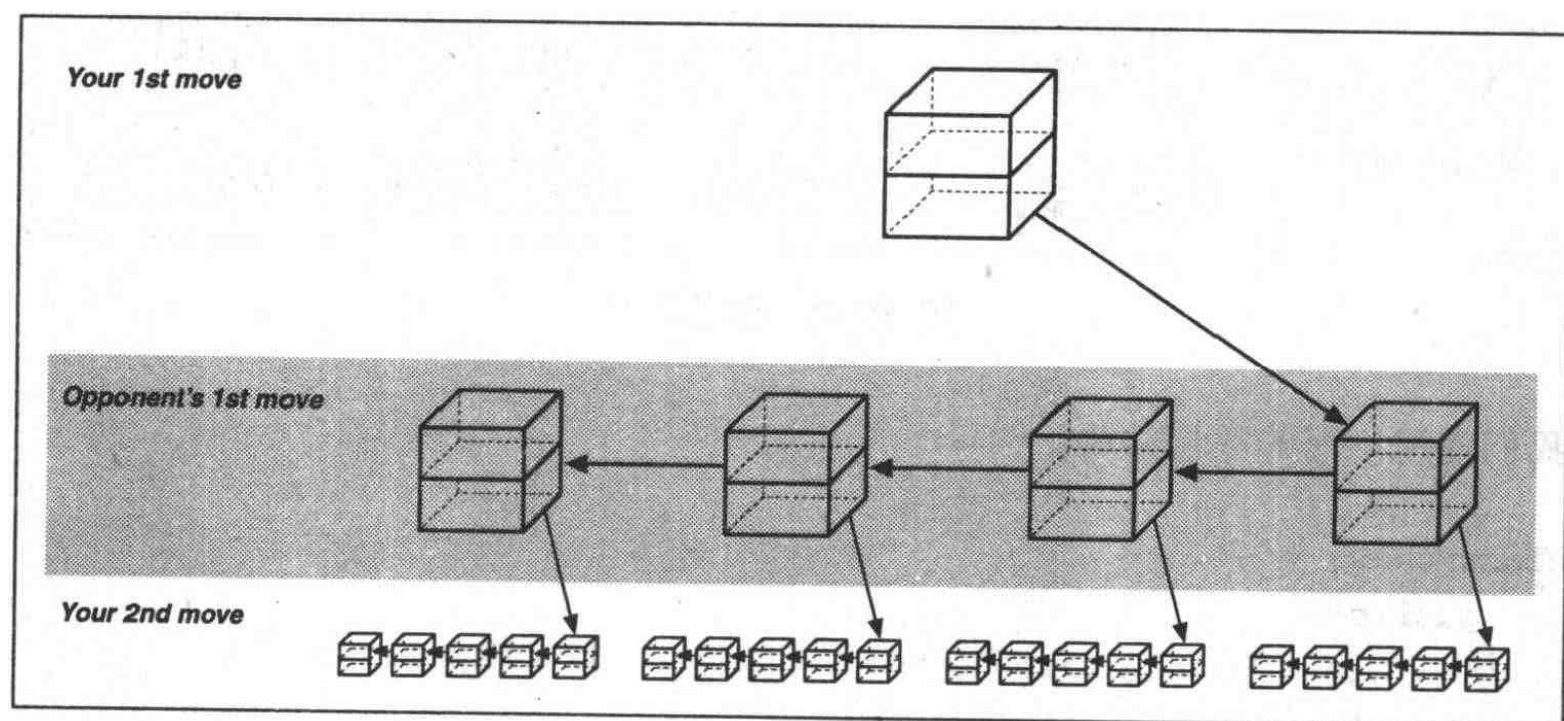


图 20.15 修改后的下棋结构

新的版本增加了一些复杂度，但是节省了大量的存储空间。

编程练习

练习20.1: 编写一个交叉引用程序。

练习20.2: 编写一个函数，它删除链表中的一个元素。

练习20.3: 编写一个函数，它删除双向链表中的一个元素。

练习20.4: 编写一个函数，它删除树中的一个元素。

答疑

问题20.1: 问题出在下面这条语句:

```
while ((current_ptr->data != value) &&
        (current_ptr != NULL))
```

在我们检查current_ptr是不是一个无效指针 (!= NULL) 之前，就先检查了current_ptr-

>data。如果指针是NULL，我们就可能检查了内存中的一个随机地址，其中可能含有任何信息。解决的办法是在检查current_ptr指向什么之前，先检查它是否为空：

```
while ((current_ptr != NULL) {  
    if (current_ptr->data == value)  
        break;
```

问题20.2：问题出在下面这一点：因为字典中的第一字总是最小，所以所有其它的字都排在右子树上，事实上，因为整个链表是有序的，所以只用了右子树。虽然定义的是树结构，但结果还是一个链表。见图20.16。

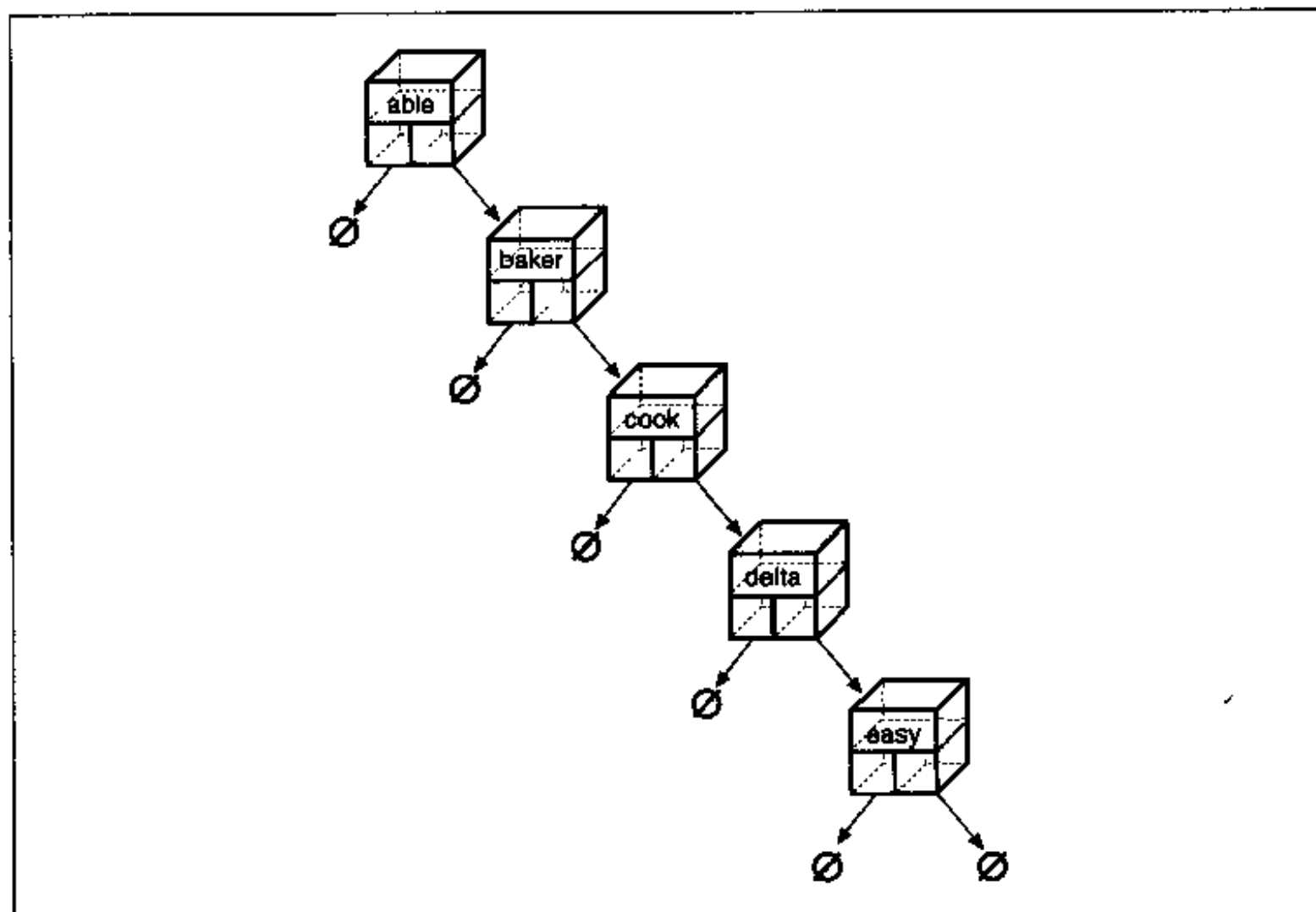


图 20.16 字典树

更详细讨论数据结构的书，如Knuth所著的《算法+数据结构=程序》讨论了使用平衡二叉树避免这种情况产生的算法。

琐碎问题的回答：你可以投降。对了，第21步棋表示认输。

本章内容:

- 派生类
- 虚函数
- 虚类
- 隐藏在派生类中的函数
- 派生类中的构造函数和析构函数
- 小结
- 编程练习
- 答疑

第21章

高级类

派生类

在示例13.1中定义的stack类有一个很大的局限性：它不检查数据的正确性。例如，没有办法阻止用户向栈中输入过多的数据。

我们需要定义一个新的有边界检查的栈（b_stack）。这个新栈除了与简单栈执行同样的操作外，还进行边界检查。C++允许你在旧类的基础上建立一个新类。这里，我们将在一个已存在栈（stack）的基础上，建立带边界检查的栈（b_stack）。我们使用stack类作为基类（base class），来创建这个带边界检查的派生类（derived class）。

开始，我们告诉C++从stack创建b_stack。

```
class b_stack: public stack {
```

关键字public告诉C++，在stack类外可以访问它所有的成员。如果我们把stack说明为private，则stack的public和protected的成员只可以在b_stack的内部访问。

这个说明告诉C++，我们要把stack作为一个b_stack的基类来使用。图21.1显示C++如何看待这个联合。

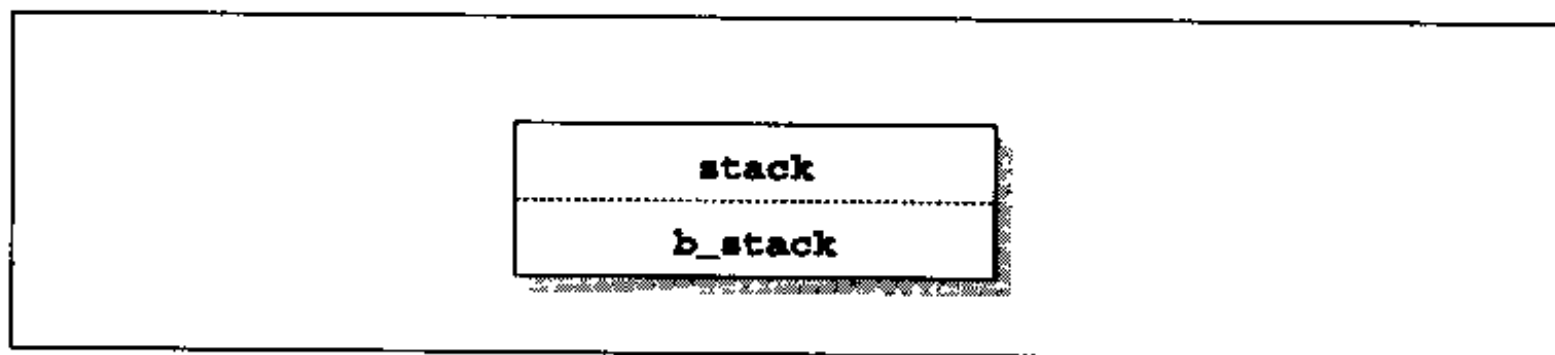


图 21.1 派生类b_stack和基类stack

现在我们需要定义新的push成员函数。我们首先检查栈中是否有空间。如果没有空间了，打印一条错误信息，程序中断。否则，把项压入栈中。下面是稍做改进的push函数。


```

inline void b_stack::b_push(const int item)
{
    if (count >= STACK_SIZE) {
        cout << "Error: Push overflows stack\n";
        exit(8);
    }
    // This calls the push member function of the stack class
    push(item);
}

```

我们很注意为这个成员函数选取名字。因此它叫**b_push**而不是**push**。我们愿意把它称为**push**，但是代码：

```

inline void b_stack::push(const int item)
{
    if (count >= STACK_SIZE) {
        cout << "Error: Push overflows stack\n";
        exit(8);
    }
    // The next line wrongly calls b_stack::push (this function),
    // not stack::push
    push(item);
}

```

将调用**b_stack**类中的**push**成员函数，而不是我们想要的**stack**类中的**push**函数。结果，我们调用**b_stack**的**push**，它执行一些检查，而后调用**push**。该**push**函数属于**b_stack**类，所以我们执行边界检查，并调用**push**，依此类推。结果，**push**将一遍一遍地调用自己，直到系统的栈空间用完为止。

这并不是我们想要的。我们要告诉C++的是，我们想调用**stack**中的**push**函数。使用范围运算符“**::**”就可以做到这一点。新的**b_stack::push**函数如下所示：

```

inline void b_stack::push(const int item)
{
    if (count >= STACK_SIZE) {
        cout << "Error: Push overflows stack\n";
        exit(8);
    }
    stack::push(item);
}

```

类**stack**和类**b_stack**的完整定义列在示例21-1中。

示例21.1 **stack_c/stack_d1.cc**

```

/*****
 * Stack
 */

```

```

*      a file implementing a simple stack class      *
*****/
#include <stdlib.h>
#include <iostream.h>

const int STACK_SIZE = 100;    // Maximum size of a stack

/*****
* Stack class
*
* Member functions
*   stack -- initialize the stack
*   push -- put an item on the stack
*   pop -- remove an item from the stack
*****/
// The stack itself
class stack {
    protected:
        int count;           // Number of items in the stack
    private:
        int data[STACK_SIZE]; // The items themselves
    public:
        // Initialize the stack
        stack(void);
        // ~stack() -- default destructor
        // Copy constructor defaults

        // Push an item on the stack
        void push(const int item);

        // Pop an item from the stack
        int pop(void);
};

/*****
* stack::stack -- initialize the stack
*****/
inline stack::stack(void)
{
    count = 0; // Zero the stack
}

/*****
* stack::push -- push an item on the stack
*
* Warning: We do not check for overflow
*****/

```

```

*
* Parameters
* item -- item to put in the stack
*****/
inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}
/*****
* stack::pop -- remove item from stack
*
* Warning: We do not check for stack underflow
*
* Returns
* the top item from the stack
*****/
inline int stack::pop(void)
{
    // Stack goes down by one
    --count;

    // Then we return the top value
    return (data[count]);
}
/*****
* b_stack -- bound checking stack
*
* Member function
* push -- push an item on the stack
* pop -- remove an item from the stack
*****/
class b_stack: public stack {
public:
    // b_stack -- default constructor
    // ~b_stack -- default destructor
    // Copy constructor defaults

    // Push an item on the stack
    void push(const int item);

    // Remove an item from the stack
    int pop(void);
};

```

```

/*****
 * b_stack::push -- push an item on the stack
 *
 * Parameters
 *     item -- item to put in the stack
 *****/
inline void b_stack::push(const int item)
{
    if (count >= STACK_SIZE) {
        cerr << "Error: Push overflows stack\n";
        exit (8);
    }
    stack::push(item);
}

/*****
 * b_stack::pop -- get an item off the stack
 *
 * Returns
 *     the top item from the stack
 *****/
inline int b_stack::pop(void)
{
    if (count <= 0) {
        cerr << "Error: Pop causes stack underflow\n";
        exit (8);
    }
    return (stack::pop());
}

```

虽然这两个类相对简单一些，但它们说明了C++中一些重要特性。我们首次把count说明为protected成员变量。这就是说，该变量仅能用于类stack及从stack派生的类，如b_stack类。类b_stack的函数push和pop可以使用count，但是stack和b_stack之外的任何类或函数都不能使用这个变量。

因为b_stack派生于stack，所以你能在可使用stack类型变量的地方，使用b_stack类型的变量。在下面的例子中，我们创建一个检查边界的栈bounded_stack，它用作push_thing函数的参数。push_thing的参数是一个普通的、不进行边界检查的栈。

```

void push_things(stack &a_stack) {
    a_stack.push(1);
    a_stack.push(2);
}

// ...
b_stack bounded_stack; // A random stack

```

```
// ...
push_things(bounded_stack);
```

`push_things`函数把一个`stack`作为它的参数。虽然变量`bounded_stack`是`b_stack`类型的变量，但是当调用`push_things`时，C++把它转换为`stack`类型。

对此的解释是，虽然`bounded_stack`是`b_stack`类型的，但当它被`push_things`使用时，它透过窥视孔只能看到`stack`类型的变量，见图21.2所示。

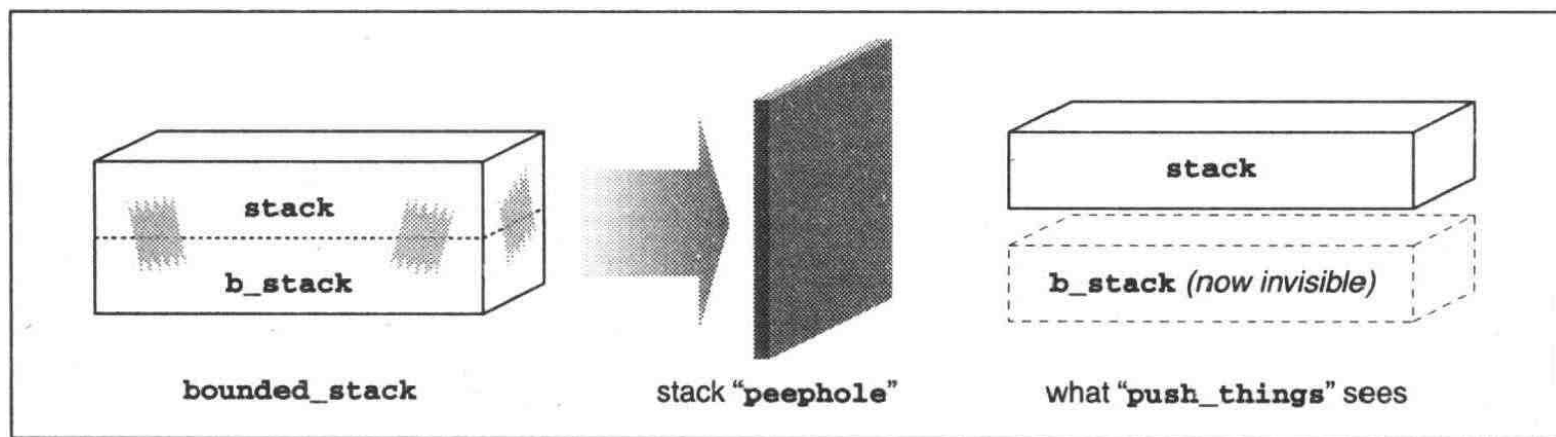


图 21.2 `push_things`会看到一个`b_stack`

下面我们改进`stack`，不让他总是分配固定大小的栈，而是动态分配栈空间。新的`stack`类如下：

```
class stack {
private:
    int *data;    // Pointer to the data in the stack
protected:
    int count;    // Current item on the stack
public:
    stack(const unsigned int size) {
        data = new int[size];
        count = 0;
    };
    virtual ~stack(void) {
        delete data;
        data = NULL;
    }
// ...
```

（我们将在本章的后面讨论关键字`virtual`。）

这个栈具有更大的灵活性。如要使用新的`stack`，在说明`stack`变量时，必须指定栈大小。例如：

```
stack big_stack(1000);
stack small_stack(10);
stack bad_stack; // Illegal, size required
```

回到带边界检查的栈。出于某种原因我们需要用一个参数来调用基类的构造函数（stack）。

实现的方法是把基类的构造函数放在派生类构造函数的说明后面。但是这种灵活性对于边界检查的栈来说，会带来若干问题：stack的构造函数包含一个参数。边界检查栈如何去初始化这个简单的栈呢？

解决的办法是使用类似于初始化常量数据成员的语法。

```
class b_stack: public stack {
    private:
        const unsigned int stack_size;    // Size of the simple stack
    public:
        b_stack(const unsigned int size) : stack(size),
                                          stack_size(size) {
        }
}
```

本例中，基类是stack，派生类是b_stack。b_stack的构造函数带有一个参数，即栈的大小。它必须把这个参数传给stack。下面一行代码：

```
b_stack::b_stack(const unsigned int size) : stack(size),
```

通过stack(size)来完成该功能。

注意

因为新的stack使用动态内存（new和delete），所以我们必须定义“四大”成员函数：构造函数、析构函数、拷贝构造函数和赋值运算符（=）。

虚函数

今天发一封信有多种不同的方法。我们可以通过邮局邮寄，可以使用特快专递，也可以使用传真。所有的方法都是把信送达另一人（大多数时候），但是它们的费用和速度各不相同。

让我们定义一个处理送信的类。我们先定义一个地址类，然后使用这个类去定义发信人和收信人的地址。（地址类的定义“仅仅是程序设计的开头”，余下的留给读者完成。）

```
class mail {
    public:
        address sender; // Who's sending the mail (return address)?
        address receiver; // Who's getting the mail?

        // Send the letter
        void send_it(void) {
```



```

        // ... Some magic happens here
    };
};

```

但是，这个类中有点小问题。我们根据“魔法”来发信。发信的过程因使用的服务不同而有所区别。处理这个问题的一个办法是，根据使用的服务，让send_it调用相应的程序：

```

void mail::send_it(void) {
    switch (service) {
        case POST_OFFICE:
            put_in_local_mailbox();
            break;
        case FEDERAL_EXPRESS:
            fill_out_waybill();
            call_federal_for_pickup();
            break;
        case UPS:
            put_out_ups_yes_sign();
            give_package_to_driver();
            break;
        //... and so on for every service in the universe
    }
}

```

这个办法有点笨。邮件类必须了解世界上所有的邮件服务，还要考虑当我们给类添加另外的函数时，会出现什么情况：

```

class mail {
public:
    // Returns the cost of mailing in cents
    int cost(void) {
        // ... more magic
    }
}

```

我们是不是要写另一个大switch语句？如果是的话，我们将要管理两个大switch语句。更不幸的是，发信命令和每种服务的费用也要涉及到两个函数中。如果我们能把邮局邮寄中的所有函数组织在一个类中，把特快专递中的所有函数组织在另一个类中就好了。

例如，用于邮递的类可以是：

```

class post_office: public mail{
public:
    // Send the letter
    void send_it(void) {
        put_in_local_mailbox();
    };
    // Cost returns cost of sending a letter in cents
    int cost(void) {

```

```

        // Costs 32 cents to mail a letter
        return (32);    // WARNING: This can easily become dated
    }
};

```

现在，我们有了每个类中每个服务的信息。这些信息以易于理解的格式存储。问题是它们不易使用。例如，我们写一个发信程序：

```

void get_address_and_send(mail &letter)
{
    letter.from = my_address;
    letter.to = get_to_address();
    letter.send();
}
// ...
class post_office simple_letter;
get_address_and_send(simple_letter);

```

麻烦的是，`letter`是`mail`类的。当我们调用`letter.send()`时，我们调用的是基类`mail`的`send`函数。我们要有一种办法告诉C++：“请调用派生类的`send`函数，不要使用基类中的。”

virtual（虚）关键字标识了可被派生类中成员函数所覆盖的一个成员函数。如果我们使用派生类，则C++将寻找派生类中的成员，然后再在基类中寻找，顺序就是这样的。如果我们使用的是基类变量（即使真正的实例是一个派生类），则C++将只在基类中寻找成员函数。当基类中定义了一个虚函数时，情况例外。本例中寻找派生类，然后寻找基类。

表21.1列出不同的寻找算法。

表21.1 成员函数寻找次序

类类型	成员函数类型	寻找次序
派生	正常	派生->基
基	正常	基
基	正常	派生->基

示例21.2说明虚函数的使用方法。

```

示例21.2 virt/virt.cc
// Illustrates the use of virtual functions

#include <iostream.h>

class base {
    public:

```



```

        void a(void) { cout << "base::a called\n"; }
        virtual void b(void) { cout << "base::b called\n"; }
        virtual void c(void) { cout << "base::c called\n"; }
};

class derived: public base {
    public:
        void a(void) { cout << "derived::a called\n"; }
        void b(void) { cout << "derived::b called\n"; }
};

void do_base(base &a_base)
{
    cout << "Call functions in the base class\n";

    a_base.a();
    a_base.b();
    a_base.c();
}

main()
{
    derived a_derived;

    cout << "Calling functions in the derived class\n";

    a_derived.a();
    a_derived.b();
    a_derived.c();

    do_base(a_derived);
    return (0);
}

```

派生类有三个成员函数。其中的两个是自定的：a和b。第三个c是继承于基类的。当调用a时，C++查看派生类，看看它是不是定义了这个函数。本例中，确实是这样，所以，代码：

```
a_derived.a();
```

将输出：

```
derived::a called
```

当调用b时也一样，我们得到：

```
derived::b called
```

基类是否定义了a和b并没有关系。C++调用派生类，并到此为止。

但是，派生类中没有一个叫c的成员函数。所以当我们执行到下面这行代码时：

```
a_derived.c();
```

C++试图在派生类中寻找，但失败了。然后它转到基类中寻找。在本例中，它成功了，我们得到：

```
base::c called
```

下面讨论do_base函数。因为它带有一个基类作为它的变量，所以C++只在基类中寻找成员函数。下面一行代码：

```
a_base.a();
```

将输出：

```
base::a called
```

当调用成员函数b时会出现什么情况呢？这是一个虚函数。它告诉C++寻找规则可以改变。C++首先在派生类中寻找有没有成员函数b，然后再在基类中寻找。本例中，b在派生类中，所以代码：

```
a_base.b();
```

将输出：

```
derived::b called
```

成员函数c也是虚函数。所以C++先在派生类中寻找函数。本例中，它没有定义在派生类中，所以C++再去基类中寻找。在那有定义，我们得到：

```
base::c called
```

现在再回到邮件程序中，我们需要一个简单的基类，它描述每种不同类型的服务中的基本邮件函数。

```
class mail {
public:
    address sender; // Who is sending the mail (return address)?
    address receiver; // Who is getting the mail?

    // Send the letter
    virtual void send_it(void) {
        cout << "Error: send_it not defined in derived class.\n"
        exit (8);
    };
    // Cost of sending a letter in pennies
    virtual int cost(void) {
```

```

        cout << "Error: cost not defined in derived class.\n"
        exit (8);
    };
};

```

现在对于每个不同类型的服务，定义一个派生类。例如：

```

class post_office: public mail {
public:
    void send_it(void) {
        put_letter_in_box();
    }
    int cost(void) {
        return (29);
    }
};

```

现在写一个发信的程序，不必再担心细节问题了。我们只需调用`send_it`，让虚函数去处理其它事宜。

`mail`类是一个抽象类，它描述一般的发邮件过程。为了让它与真正的邮件服务相互引用，我们用它作为基类，得到派生类。如果程序员忘记在派生类中放入正确的成员函数怎么办呢？例如：

```

class federal_express: public mail {
public:
    void send_it(void) {
        put_letter_in_box();
    }
    // Something is missing
};

```

当我们想知道通过特快专递发信的费用时，C++注意到在`federal_express`类中没有`cost`函数，所以它调用`mail`中的函数。`mail`中的`cost`函数知道自己应该永远不被调用，所以它发出一条错误信息，程序中断。得到一条错误信息好是好，但在编译时得到比在运行时得到更好。

C++允许你指定必须在派生类中被覆盖的虚函数。以本例为例，新的改进后的代码如下：

```

class mail {
public:
    address sender; // Who is sending the mail (return address)?
    address receiver; // Who is getting the mail?

    // Send the letter

```

```

    virtual void send_it(void) = 0;
    // Cost of sending a letter in pennies
    virtual int cost(void) = 0;
};

```

“= 0”告诉C++这些成员函数是纯虚函数。即，永远不直接调用它们。含有一个或多个纯虚函数的任何类称为抽象类。如果你想把抽象类用作普通类型的话，如：

```

void send_package(void) {
    mail a_mailer;    // Attempt to use an abstract class

```

你会得到一条编译时错误信息。

虚 类

让我们设计一些类来处理税费表格。表格的右上角是空格，让你填入姓名、地址和身份证号。所有的表格都包含同样的信息，所以可以定义处理表格的右上角信息的类。

```

class name {
public:
    char *name;    // Name of the taxpayer
    // ... Rest of the class
};

```

现在用这个类设计另一个用于1040表格的类。

```

class form_1040: public name {
public:
    int income;    // Wages, tips, and other income
    // ...
};

```

这两个类结构如图21.3所示。

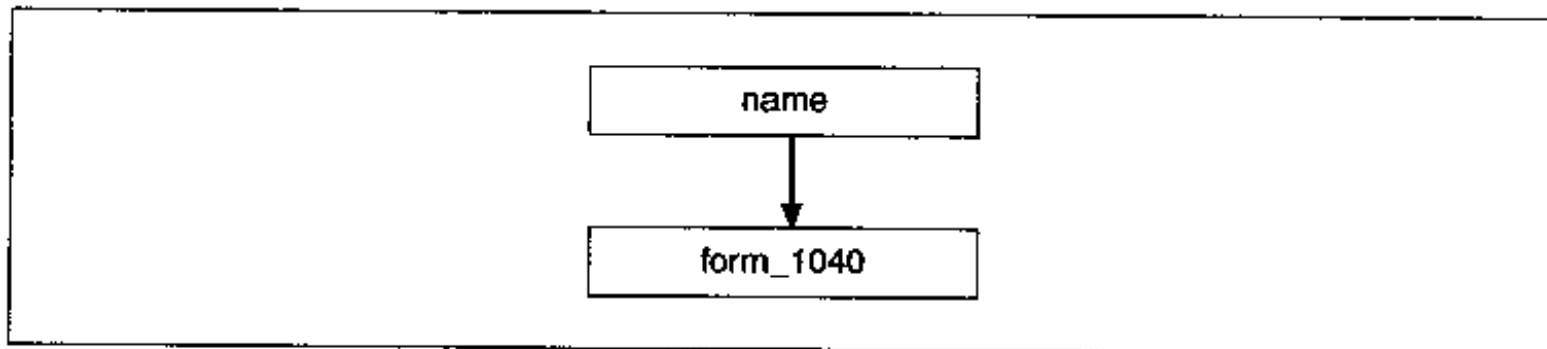


图 21.3 1040类结构

不幸的是，我们的纳税申报单比一个表格要复杂得多。我们还需要一个Schedule_A来

处理扣除额，所以定义这样一个类。

```
class schedule_a: public name {  
    public:  
        int home_interest;    // interest deduction for home mortgage  
        // ... Rest of class  
};
```

把两个表格放在一起，得到了一个简单的申报单。

```
class tax_return: public form_1040, schedule_a {  
    // ... Rest of class  
};
```

图21.4 说明了这个类的结构。

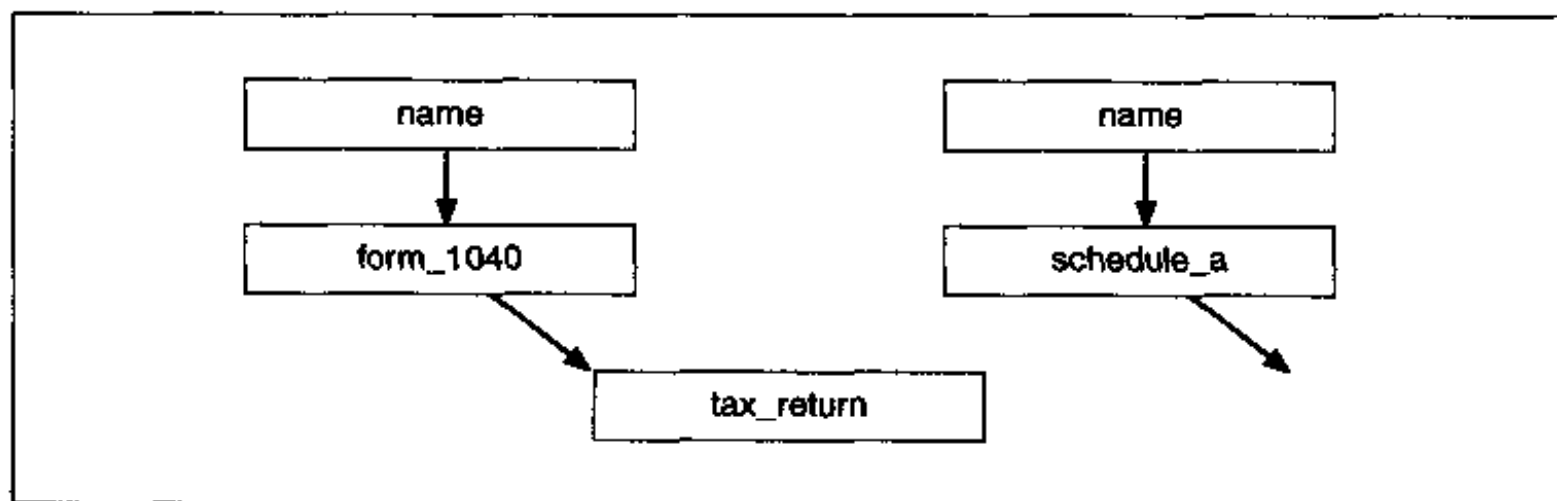


图 21.4 纳税申报单结构

这个结构的问题是，我们有两个name类。但是纳税人的名字不随表格的改变而改变。我们想要一种图21.5中所示的类结构。

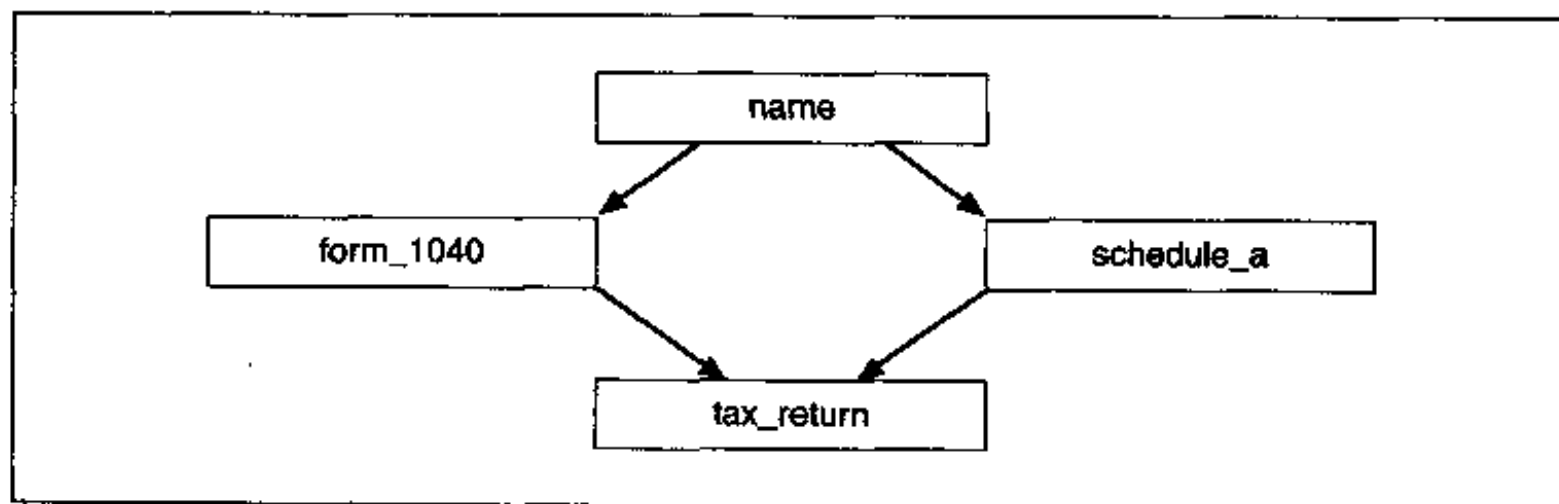


图 21.5 更好的纳税申报单结构

把基类说明为虚类，就是告诉C++把相同的基类组合在一起。用虚基类重新定义 tax_return，得到：

```
class form_1040: virtual public name {  
    public:
```

```

        int income;    // Wages, tips, and other income
        // ...
};

class schedule_a: virtual public name {
public:
    int home_interest;    // Interest deduction for home mortgage
    // ... Rest of class
};

class tax_return: public form_1040, schedule_a {
    //... Rest of class
};

```

注意，`name`类用作两个派生类的基类；派生类调用它们基类的构造函数来初始化该类。这是不是意味着`name`的构造函数将被调用两次呢？回答是否定的。C++知道`name`使用了两次，它自动略去第二次初始化。

隐藏在派生类中的函数

示例21.3定义了一个基类，它有重载函数`do_it`，该函数有用于整数和浮点数的两个版本。程序还定义了一个派生类，类中只有用于整数的`do_it`。

```

class simple {
public:
    int do_it(int i, int j) {return (i*j);}
    float do_it(float f) {return (f*2);}
};

class derived: public simple {
public:
    int do_it(int i, int j) {return (i+j);}
};

```

很明显，当我们使用派生类并调用整数`do_it`时，我们调用的是派生类中的函数。但是如果调用浮点数`do_it`时会出现什么状况？派生类中没有浮点数`do_it`。一般地，如果在派生类中不存在所要的成员函数，C++将去基类中寻找。

但是，因为派生类中定义了`do_it`的一个版本，C++将在派生类中寻找`do_it`的所有版本。换句话说，如果在派生类中定义了`do_it`的一种格式，那么会把函数的所有格式摒弃在外。

```

main() {
    derived test;    // Define a class for our testing
    int i;           // Test variable
    float f;         // Test variable

```



```

        i = test.do_it(1, 3);    // legal; returns 4 (1 + 3)
        f = test.do_it(4.0);    // Illegal; "do_it(float)" not defined in
                                // the class "derived"
    }

```

派生类中的构造函数和析构函数

构造函数和析构函数不同于普通的成员函数，特别是当用于派生类中时更是如此。当创建一个派生类变量时，首先调用基类的构造函数，然后调用派生类的构造函数。

示例21.4定义了一个简单的基类，并用它创建一个派生类。

示例21.4 cons/class.cc

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

```

class base_class {
public:
    base_class() {
        cout << "base_class constructor called\n";
    }
    ~base_class() {
        cout << "base_class destructor called\n";
    }
};

class derived_class:public base_class {
public:
    derived_class() {
        cout << "derived_class constructor called\n";
    }
    ~derived_class() {
        cout << "derived_class destructor called\n";
    }
};

```

当我们运行代码：

```
derived_class *sample_ptr = new derived_class;
```

时，程序会输出：

```

base_class constructor called
derived_class constructor called

```

删除该变量后，调用析构函数。首先调用派生类的析构函数，然后调用基类的析构函

数。所以当用下面的语句删除该变量时：

```
delete sample_ptr;
sample_ptr = NULL;
```

得到：

```
derived_class destructor called
base_class destructor called
```

但是C++有个奇怪的潜在问题。记住派生类可以当基类来用。例如：

```
base_class *base_ptr = new derived_class;
```

完全合法。但是，当删除变量时，会有问题：

```
delete base_ptr;
base_ptr = NULL;
```

你知道，`base_ptr`是指向基类的指针。这里，所有的代码涉及的都是基类。没有办法让C++知道还有一个派生类存在。所以当删除该变量时，C++不去调用派生类的析构函数。

`delete`语句的输出是：

```
base_class destructor called
```

我们只能让C++在不调用相应的析构函数的情况下删除一个类。

我们要有一些办法告诉C++：“喂，那儿还有一个派生类，你可能想调用它的析构函数。”这样做是为了让基类的析构函数成为一个虚函数。

```
class base_class {
public:
    base_class() {
        cout << "base_class constructor called\n";
    }
    virtual ~base_class() {
        cout << "base_class destructor called\n";
    }
};
```

关键字`virtual`的一般含义是：调用派生类中的函数，而不去调用基类中的那一个。对于析构函数来说，含义稍有不同。当C++见到虚析构函数时，它将调用派生类的析构函数，然后再调用基类的析构函数。

所以，写上虚析构函数后，我们可以安全地删除`base_class`变量了，程序将输出正确的信息：

```
derived_class destructor called
```


base_class destructor called

问题21.1: 当我们删除变量list_ptr时, 示例21.5为什么失败了? 当程序在第20行调用clear时, 好象很乱。

```
示例21.5 blow/blow.cc
1 #include <iostream.h>
2 #include <stdlib.h>
3
4 class list {
5     private:
6         int item;          // Current item number
7
8     public:
9         virtual void clear() = 0;
10
11         void next_item(void) {
12             ++item;
13         }
14
15         list(void) {
16             item = 0;
17         }
18
19         virtual ~list() {
20             clear();
21         }
22 };
23
24 class list_of_integers : public list {
25     public:
26         int array[100];    // Place to store the items
27
28         void clear(void) {
29             int i;          // Array index
30
31             for (i = 0; i < 100; ++i)
32                 array[i] = 0;
33         }
34 };
35
36 main()
37 {
38     list_of_integers *list_ptr = new list_of_integers;
39
```

```

40     // Cause problems
41     delete list_ptr;
42     list_ptr = NULL;
43     return (0);
44 )

```

小 结

从程序设计开始，程序员就一直在寻找建立可重用代码的方法。通过使用派生类，C++允许你在已有代码的基础上创建类。它提供了很大的灵活性，并使代码易于组织及维护。

编程练习

练习21.1：把练习13.2中的`checkbook`类和练习13.3中的`queue`类组合在一起，实现一个`checkbook`类，它可以输出输入到`checkbook`中最后10项的内容。

练习21.2：定义一个“字符串匹配”基类。

```

class string_matcher {
public:
    // Returns true if string matches, false if not
    int match(const char *const string);
...

```

定义匹配字、数和空串的派生类。

练习21.3：定义一个基类`shape`，它可以描述任何简单的形状，如正方形、圆或等边三角形。所有这些形状的大小可用面积来表示。为上述三个形状定义派生类。

在基类中创建一个虚类，返回每个形状的面积。

注：你需要更精确地定义存储在基类中的面积。（对于基类中圆所使用的长度，是指半径还是指直径？）

练习21.4：编写一个`person`基类，它描述一个人，不管是男性还是女性。定义两个派生类，分别叫`man`和`woman`，描述男性和女性。对于男性和女性公共的操作，在基类的纯虚函数中实现，不一致的操作分别实现。

练习21.5：编写一个基类`number`，它有一个整数值，还有一个成员函数`print_it`。定义三个派生类，分别以十六进制、八进制和十进制输出该值。

答 疑

问题21.1：记住，析构函数的调用次序是：先派生类后基类。本例中，调用派生类的析构函数`list_of_integers`来删除类。这个类已经没有了。

下一步，调用基类list的析构函数。它调用函数clear。这是个纯虚函数，所以C++必须调用派生类中的clear函数。但是派生类已经没有了，所以没有clear函数了。这样C++就乱了，程序中断。（实际上，只有好的编译器才中断程序。其它的编译器或许会得到很怪的结果。）

你永远不要从析构函数中调用纯虚函数。

第五部分

语言的其它

特性

1. 1. 1.

1. 1. 1.

1. 1. 1.

本章内容:

- 栈例外
- 运行时库例外
- 编程练习

第22章

例外

飞机从一地飞往另一地，99%的时候没有什么麻烦。但是当飞机的轮子和发动机出现故障时，飞机驾驶员就要进行紧急处理。

让我们详细看看飞行时遇到像发动机着火这样的紧急情况的处理。这是正常飞行时的例外情况。驾驶员座舱响起着火警报。这将引起驾驶员的注意，他们将开始火警紧急处理过程。在着火的时候，要做许多的事情。航空公司预先已经罗列清楚，驾驶员也已熟记在心。驾驶员做必须要做的事：打开灭火器、关闭发动机、迅速着陆，等等。

我们把这个过程分解为C++伪代码。当驾驶员起飞时，他们将试着无故障地从一地飞往另一地。C++“代码”为：

```
try {  
    fly_from_point_a_to_point_b();  
}
```

关键字try表明，我们将试着完成一次可能引起例外的操作。

但是，当遇到一次例外时怎么办？我们必须处理它。C++代码为：

```
catch (fire_emergency &fire_info) {  
    active_extinguisher(fire_info.engine);  
    turn_off(fire_info.engine);  
    land_at_next_airport();  
}
```

关键字catch表明，这段代码处理一个例外。本例中，要处理的例外是fire_emergency。这是紧急情况类型。它可能是1号发动机、2号发动机或3号发动机着火（假定是一驾三发动机的飞机）。变量fire_info中存储的是哪个发动机着火。

fire_emergency类描述发生的是哪类火灾。它的定义为：

```
class fire_emergency {  
    public:
```



```

        int engine;    // Which engine is on fire
        // Other information about the fire
    };

```

我们概述了每件事，但还没有找到火灾的具体位置。每个发动机的外面都有火灾传感器。用于这个传感器的代码为：

```

// Watch for fire in engine #2
void sensor_2(void) {
    while (engine_running()) {
        if (engine_on_fire()) {
            fire_emergency fire_info;

            fire_info.engine = 2;
            throw(fire_info);
        }
    }
}

```

当这段代码发现有火情时，它把信息放到`fire_emergency`类的变量`fire_info`中，并用`throw`语句触发例外处理。

当执行`throw`语句时，正常的过程停止了。首先，当火灾发生时，停止正常飞行，转去执行`fire_emergency`的`catch`语句。

总之，例外处理包括：

- 可能出现问题的描述，本例中是`fire_emergency`类。
- 可能发生例外的一段代码，它包含在`try`语句中。本例中，该语句是`fly_from_point_a_to_point_b()`。
- 引起例外并通过`throw`语句触发紧急过程的事情。
- `catch`块中的例外处理代码。

栈例外

在第21章“高级类”中，我们定义了一个带有边界检查的栈。如果用户试图向栈中压入过多的数据，或从栈中弹出的数据过多，类都会发出一条错误信息，并且中断程序。这不是处理例外的好办法。如果每次着火飞机都显示一条错误信息，并关闭发动机，驾驶员该做何感想。

我们做的第一件事是决定我们要处理的是哪种例外，并把它们描述为类。在栈示例中，我们能想出的唯一例外是越界错误。我们用一个简单的串来描述这个错误。越界错误的类为：

```

const int WHAT_MAX = 80;           // Longest possible error message
class bound_err {
public:
    char what[WHAT_MAX];           // What caused the error

    // Initialize the bound error with a message
    bound_err(char *_what) {
        // WARNING: This does not check for "_what" being too long
        strcpy(what, _what);
    }
    // bound_err(bound_err) -- default copy constructor
    // ~bound_err -- default destructor
};

```

例外检查从关键字**try**开始。它告诉C++，随后的这段代码中可能会出现例外情况，而且立即在**try**块后处理例外。例如，如果我们要执行一个大栈操作，代码如下：

```

try {
    do_big_stack_operation();
};

```

在**try**块后，我们需立即告诉C++将处理哪些问题，这里用的是**catch**语句。该语句的语法是：

```

catch (问题类型 &参数) {
    语句;
}

```

问题类型是描述发生了什么类。对于越界错误，**catch**语句如下：

```

catch (bound_err &what_happened) {
    cerr << "Error: Bounds exceeded\n";
    cerr << "Reason: " << what_happened.what << '\n';
}

```

可能要用几个**catch**语句来处理遇到的不同类型的例外。如果有一个例外没有处理到，它就被称为意外的例外，将调用意外例外的处理程序。该处理程序按缺省方式中断原程序的执行。如果你想处理所有的例外，使用“...”来表示例外类型。例如：

```

catch (bound_err &what_happened) {
    // .... Body of catch
}
catch (...) {
    cerr << "Something strange happened\n";
}

```


现在我们要修改原来的栈程序，把所有的“error-message-and-abort”代码都换为throw语句。新的push过程如下：

```
inline void b_stack::push(const int item)
{
    if (count >= STACK_SIZE) {
        bound_err overflow("Push overflows stack");
        throw overflow;
    }
    stack::push(item);
}
```

实际上，我们不需要用特殊的变量表示上溢。代码还可以进一步改进。在前面的示例中，我们用两个语句明确地显示正在做什么。下面的代码执行同样的操作：

```
inline void b_stack::push(const int item)
{
    if (count >= STACK_SIZE) {
        throw bound_err("Push overflow stack");
    }
    stack::push(item);
}
```

我们使用的基本函数定义告诉C++，“随时等待例外的发生。”push函数只能产生bound_err例外。C++允许你在函数说明的后面写throw命令，列出该函数中可能出现的所有例外：

```
inline void b_stack::push(const int item) throw(bound_err) {
```

但是如果我们发生了一种没有列在例外表中的例外时，会出现什么情况呢？C++将转去调用函数unexpected()。示例22.1中是能处理例外情况的带边界检查栈的源代码。

示例22.1 stack_c/stack_el.cc

```
/* *****
 * Stack
 * a file implementing a simple stack class
 * ***** */
#include <stdlib.h>
#include <iostream.h>

const int STACK_SIZE = 100; // Maximum size of a stack

/* *****
 * Stack class
 *
 * Member functions
 * ***** */
```



```

*      stack -- initialize the stack      *
*      push -- put an item on the stack   *
*      pop -- remove an item from the stack *
*****/
// The stack itself
class stack {
    protected:
        int count;          // Number of items in the stack
    private:
        int data[STACK_SIZE]; // The items themselves
    public:
        // Initialize the stack
        stack(void);
        // ~stack() -- default destructor
        // Copy constructor defaults

        // Push an item on the stack
        void push(const int item);

        // Pop an item from the stack
        int pop(void);
};

/*****
* stack::stack -- initialize the stack      *
*****/
inline stack::stack(void)
{
    count = 0; // Zero the stack
}

/*****
* stack::push -- push an item on the stack   *
*                                             *
* Warning: We do not check for overflow      *
*                                             *
* Parameters                                *
*      item -- item to put in the stack      *
*****/
inline void stack::push(const int item)
{
    data[count] = item;
    count++;
}

/*****
* stack::pop -- get an item off the stack    *

```

```

*
* Warning: We do not check for stack underflow
*
* Parameters
*     the_stack -- stack to initialize
*
* Returns
*     the top item from the stack
*****/
inline int stack::pop(void)
{
    // Stack goes down by one
    count--;

    // Then we return the top value
    return (data[count]);
}

const int WHAT_MAX = 80;        // Largest possible error message
/*****
* bound_err -- a class used to handle out-of-bounds
*               exceptions.
*****/
class bound_err {
public:
    char what[WHAT_MAX];        // What caused the error

    // Initialize the bound error with a message
    bound_err(char *_what) {
        if (strlen(_what) < (WHAT_MAX - 1))
            strcpy(what, _what);
        else
            strcpy(what, "Internal error: _what is too long");
    }
    // bound_err(bound_err) -- default copy constructor
    // ~ bound_err -- default destructor
};

/*****
* b_stack -- bound-checking stack
*
* Member function
*     push -- push an item on the stack
*     pop -- remove an item from the stack
*****/

```

```

class b_stack: public stack {
public:
    // b_stack -- default constructor
    // ~b_stack -- default destructor
    // Copy constructor defaults

    // Push an item on the stack
    void push(const int item) throw(bounds_err);

    // Remove an item from the stack
    int pop(void) throw(bounds_err);
};

/*****
 * b_stack::push -- push an item on the stack
 *
 * Parameters
 *     item -- item to put in the stack
 *****/
inline void b_stack::push(const int item) throw(bounds_err)
{
    if (count >= STACK_SIZE) {
        bounds_err overflow("Push overflows stack");
        throw overflow;
    }
    stack::push(item);
}

/*****
 * b_stack::pop -- get an item off the stack
 *
 * Returns
 *     the top item from the stack
 *****/
inline int b_stack::pop(void) throw(bounds_err)
{
    if (count <= 0) {
        throw bounds_err("Pop causes stack underflow");
    }
    return (stack::pop());
}

b_stack test_stack;    // Define a stack for the bounds checking

void push_a_lot(void) {
    int i;              // Push counter

```

```

        for (i = 0; i < 5000; i++) {
            test_stack.push(i);
        }
    }

    main()
    {
        try {
            push_a_lot();
        }
        catch (bound_err &err) {
            cerr << "Error: Bounds exceeded\n";
            cerr << "Reason: " << err.what << '\n';
            exit (8);
        }
        catch (...) {
            cerr << "Error: Unexpected exception occurred\n";
            exit (8);
        }
        return (0);
    }
}

```

运行时库例外

例外处理机制相对较新。ANSI C++标准草案定义了运行库程序应该产生的例外。但是，这些“标准”例外太新了，它们还需重新定义和修改。而且，还有很多细节有待补充。

编译器编制人员采用这个标准还需要时间。写本书的时候，在目前可用的C++编译器中，如果在它们的运行库中有例外的话，也只有很少的几个，没有一个产生“标准”的例外。当ANSI标准更完善并且编译器得到改进后，这种状况将改变。我能给你的最好建议是，去读你所用编译器的参考手册并小心使用。

编程练习

练习22.1：修改练习13.3中的queue类。当往队列中送过多的数据时，让其触发一个例外。

练习22.2：修改练习18.3中的fraction类。当被0除时，产生一个例外。当读入一个坏数据时，也让其产生例外。

练习22.3：修改练习13.2中的checkbook类。当余额小于0时，产生例外。

练习22.4：编写函数count_letter。它带有一个字符参数，该函数统计辅音和元音的数目。如果函数没有接受字符，产生一个例外。



本章内容:

- 模块
- 公用和私用
- extern修饰符
- 头文件
- 模块体
- 使用无限数组的程序
- 用于多文件的Makefile
- 无限数组的使用

第23章

模块化程序

设计

到目前为止，我们涉及的都是小程序。当程序变得越来越大时，它们可以被划分为部分，或叫模块(module)。C++允许程序分到多个文件中，对它们分别编译，然后再组合（连接）为一个单一程序。

本章中，我们用一个程序作为例子，讨论创建好模块必需的C++技术。另外还向你介绍如何用make把这些模块组织为一个单一程序。

模 块

模块是函数或执行相关函数的类的集合。例如，处理lookup、enter和sort这些数据库函数可以组成一个模块。另外还有处理复数的模块，等等。

而且，由于编程题目很大，越来越多的程序员必须加入其中才能完成。划分大工程的有效办法是，给每个程序员分配一个不同的模块。这样，每个程序员只需考虑他或她自己代码内部的细节问题。

本章中，我们讨论一个处理无限数组(infinite array)的模块。这个程序包中的函数允许用户把数据存储到一个数组中，而不必考虑数组的大小。无限数组按照需要可以变大（它只受计算机内存大小的限制）。无限数组中存储用于直方图的数据，但也可以存储交叉引用程序中的行数或其它类型的数据。

公用和私用

模块分为两部分：公用和私用。公用部分告诉用户如何在模块内调用函数，还包含模块外可以使用的数据结构和函数的定义。公用定义放在包含于用户程序中的一个头文件里。在这个无限数组例子中，我们把公用定义放在文件ia.h中（见第394页的程序清单）。

模块内部的任何内容都是私用的。模块外不能直接使用的所有内容都应该说明为私用的。

extern修饰符

extern修饰符用来指明在当前文件之外定义而在该文件中使用的变量或函数。示例23.1说明**extern**修饰符的简单用法。

示例23.1 *main.cc and count.cc*

File: main.cc

```
#include <iostream.h>
/* Number of times through the loop */
extern int counter;

/* Routine to increment the counter */
extern void inc_counter(void);

main()
{
    int    index; /* Loop index */

    for (index = 0; index < 10; ++index)
        inc_counter();
    cout << "Counter is " << counter << '\n';
    return (0);
}
```

File: count.cc

```
/* Number of times through the loop */
int counter = 0;

/* Trivial example */
void inc_counter()
{
    ++counter;
}
```

函数main使用了变量counter。因为counter没有定义在main中，它定义在文件count.cc中。在main.cc中用extern说明，counter在另外的地方已经被说明了，此处是文件count.cc。在这个文件中不使用extern修饰符，因为它是变量的“真正”说明。

实际上，有三个修饰符可用来指明定义变量的文件，如表23.1所示。

表23.1 修饰符

修饰符	含 义
<code>extern</code>	变量/函数定义在其它文件中。
<code><空></code>	变量/函数定义在本文件中（公用），它们可以在其它文件中使用。
<code>static</code>	变量/函数局部于本文件（私用）。

注意，关键字`static`有两层含义。（它是C++语言中用处最广的修饰符。`static`的所有含义列在表14.1中。）对于局部定义的数据，`static`是指“本文件私用”。对于函数内部定义的数据，它是指“从静态内存（而不是从临时栈）分配的变量”。

C++在`static`、`extern`和`<空>`规则的使用上非常自由。可以在程序的开头把一个变量说明为是`extern`的，过后又把它定义为`<空>`。

```
extern sam;
int sam = 1;    // This is legal
```

当你把所有的外部变量定义在头文件中时，该功能很有用。程序包括头文件（并把变量定义为`extern`），然后定义真正的变量。

另外一个问题涉及到在两个不同的文件中说明同一个变量的事情。

```
File: main.cc
int flag = 0;    // Flag is off
main()
{
    cout << "Flag is " << flag << '\n';
}
File: sub.cc
int flag = 1;    // Flag is on
```

此时会出现什么情况？有几种可能性：

- 因为`main.cc`先调入，所以`flag`被初始化为0。
- 因为`sub.cc`中的内容覆写了`main.cc`中的内容，所以`flag`被初始化为1。
- 编译器可能非常仔细地分析两个程序，然后去掉最可能是错误的那一个。

本例中只有一个局部变量，即`flag`。它将被初始化为1或0，这将依据你使用的编译器来决定。即使我们把它初始化为0，程序`main`完全可能输出：

```
flag is 1
```

为了避免初始化隐藏的问题，使用关键字`static`，把变量的作用域限制在定义它的文件内。

如果我们写：


```
File: main.cc
    static int flag = 0;    // Flag is off
    main()
    {
        cout << "Flag is " << flag << '\n';
    }
```

```
File: sub.cc
    static int flag = 1;    // Flag is on
```

则在`main.cc`中的`flag`完全不同于`sub.cc`中的`flag`。但是，你仍应为变量取不同的名字，以免引起混淆。

头文件

模块中共享的信息应该放在头文件中。按照惯例，所有的头文件名结尾是“.h”。在无限数组例子中，我们使用文件`ia.h`。

头文件应该包含所有的公用信息，如：

- 清楚地描述模块做什么及能为用户提供什么功能的注释块
- 公用类定义
- 公共常量
- 公用结构
- 所有公用函数的原型
- 公用变量的`extern`说明

在无限数组例子中，文件`ia.h`中有一多半是注释。这些注释并不多，真正的代码放在程序文件`ia.cc`中。`ia.h`文件既是程序文件，又是文档文件。

注意，在`ia.h`的注释中并没有提到无限数组是如何实现的。在这一级别上，我们并不关心事情是如何做成的，而只是关心提供了哪些函数。

示例23.2 `ia/ia.h`

```
/* *****
 * Definitions for the infinite array (ia) class      *
 *                                                     *
 * An infinite array is an array whose size can grow  *
 * as needed. Adding more elements to the array      *
 * will just cause it to grow.                       *
 * -----
 * class infinite_array                               *
 *     Member functions                               *
 *     infinite_array(void)  -- default constructor  *
 * *****
```

```

*      ~infinite_array(void) -- destructor      *
*      int &operator [](int index)              *
*          gets an element of the infinite array *
*
*****/

// Number of elements to store in each cell of the infinite array
const unsigned int BLOCK_SIZE = 100;

class infinite_array {
    private:
        // The data for this block
        int    data[BLOCK_SIZE];

        // Pointer to the next array
        class infinite_array *next;
    public:
        // Default constructor
        infinite_array(void)
        {
            next = NULL;
            memset(data, '\0', sizeof(data));
        }

        // Default destructor
        ~infinite_array(void);

        // Return a reference to an element of the array
        int &operator[] (const unsigned int index);
};

```

有关这个文件还要说明几点。文件中只有常量定义、数据结构定义或是外部定义。定义的所有代码都是内联的。在头文件中没有定义真正的代码或存储。

模块体

模块体包含模块使用的所有的函数和数据。模块外不能调用的私用函数应该说明为 `static`。函数外说明但在模块外不能使用的变量应该说明为 `static`。

使用无限数组的程序

无限数组模块 (`ia.cc`) 如图23.1所示。该程序使用一个简单链表来存储数组中的元素。链表可以在需要的时候变大 (直到内存用完)。每个元素表, 或称桶, 都可以存放100个数。要找到308号元素, 程序从头开始, 跳过前三个桶, 然后从当前桶中取出第8号元素。

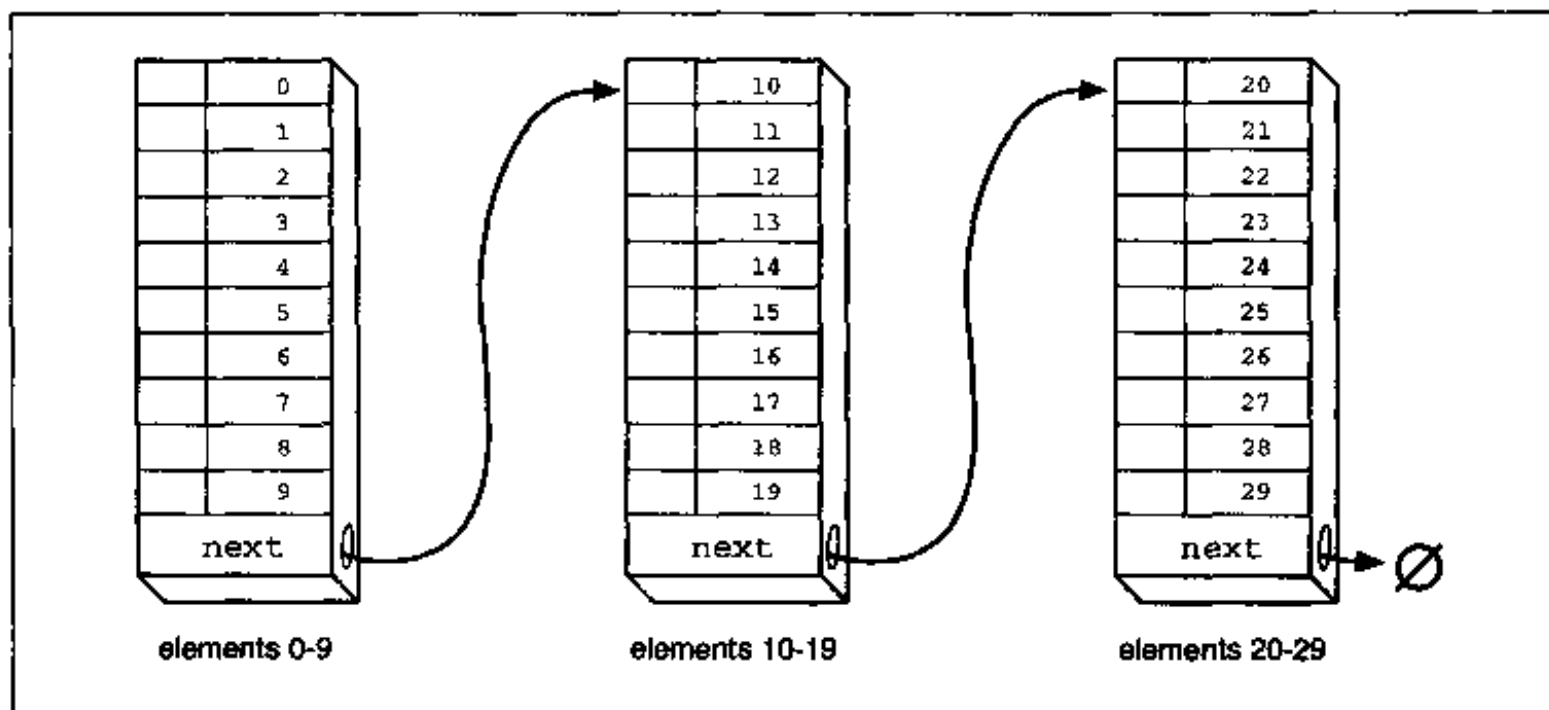


图 23.1 无限数组结构

示例23.3 ia/ia.cc

```

/*****
 * infinite-array -- routines to handle infinite arrays *
 *
 * An infinite array is an array that grows as needed. *
 * There is no index too large for an infinite array *
 * (unless you run out of memory). *
 *****/
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

#include "ia.h"          // Get common definitions

/*****
 * operator [] -- find an element of an infinite array *
 *
 * Parameters *
 *   index -- index into the array *
 *
 * Returns *
 *   reference to the element in the array *
 *****/
int &infinite_array::operator [] (const unsigned int index)
{
    // Pointer to the current bucket
    class infinite_array *current_ptr;

    int current_index; // Index we are working with

```

```

    current_ptr = this;
    current_index = index;

    while (current_index >= BLOCK_SIZE) {
        if (current_ptr->next == NULL) {
            current_ptr->next = new infinite_array;
            if (current_ptr->next == NULL) {
                cerr << "Error: Out of memory\n";
                exit(8);
            }
        }
        current_ptr = current_ptr->next;
        current_index -= BLOCK_SIZE;
    }
    return (current_ptr->data[current_index]);
}

/*****
 * ~infinite_array -- destroy the infinite array      *
 *****/
infinite_array::~infinite_array(void)
{
    /*
     * Note: We use a cute trick here.
     *
     * Because each bucket in the infinite array is
     * an infinite array itself, when we destroy
     * next, it will destroy all that bucket's "next"s
     * and so on recursively clearing the entire array.
     */
    if (next != NULL) {
        delete next;
        next = NULL;
    }
}

```

用于多文件的Makefile

make实用程序用来帮助程序员编译并连接程序。在有**make**文件之前，每次修改程序后，用户都必须精确地键入编译命令：

```
g++ -Wall -g -ohello hello.cc
```

本章，我们使用GNU g++编译器的命令。你系统上的C++编译器可以与此有不同的名字，语法上也可能略有不同。

因为程序变大，创建它们的命令数也变大。键入10个或20个一串的命令令人乏味，而且容易出错，所以程序员开始写外壳脚本(shell script) (在MS-DOS中叫.BAT文件)。然后，程序员键入的就只是do-it，计算机将编译所有的程序。但是，这也还大多。因为不管需要不需要，所有的文件都被重新编译一遍。

当一个工程中的文件数很多时，重新编译就成为一个很重要的问题。修改一个小文件，编译开始，然后必须等到第二天，而已经执行了几百个编译命令的计算机却遇到了障碍——如只短缺一个编译命令。

make程序是为智能编译(intelligent compile)而建立的。它的目的是先决定哪些命令是必须执行的，然后执行这些命令。

Makefile文件 (在UNIX系统中区分大小写) 包含make用来决定如何建立程序的规则。Makefile包含下列各部分：

- 注释
- 宏
- 精确规则
- 缺省规则

以#开头的行是注释行。

宏有下列格式：

名字 = 数据

名字是任何有效的标识符，数据是make见到\$(名字)时要替换的正文。

例如：

```
#
# Very simple Makefile
#
MACRO = Doing All

all:
    echo $(MACRO)
```

精确规则告诉make建立程序时需要哪些命令。这些规则可以有几种格式。最常用的格式为：

目标：源 [源2] [源3]

命令

[命令]

[命令]

...

目标是要建立的文件名。它由源文件“制造”或创建。如果目标是由几个文件创建的，则这些文件都应该列出。

创建目标的命令列在下一行。有时要有多个命令创建目标。每行列一个命令。每个命令用水平制表符缩进几格。

例如，规则：

```
hello: hello.cc
    g++ -Wall -g -o hello hello.cc
```

告诉make用：

```
g++ -Wall -g -o hello hello.cc
```

命令，从hello.cc文件创建hello文件。

只要需要，make将创建hello。创建hello所用的文件列在表23.2中，文件按时间（修改时间）顺序排列。

表23.2 文件修改时间

UNIX	MS-DOS/Windows	修改时间
hello.cc	HELLO.CPP	最老
hello.o	HELLO.OBJ	老
hello	HELLO.EXE	最新

如果程序员修改了源文件hello.cc，那么该文件的修改时间改变了，将影响到其它文件。make将得知这一信息，并重新创建其它文件。

精确规则的另一种格式为：

源：

命令

[命令]

这种情况下，每次运行make时，都将无条件地执行其中的命令。

如果规则中没有写命令，make将使用一组内置的规则来决定运行哪些命令。

例如，规则：


```
hist.o: ia.h hist.cc
```

告诉make使用标准规则，从`hist.cc`和`ia.h`两文件创建`hist.o`。该规则是：

```
g++ $(CFLAGS) -c file.cc
```

(make预定义了宏\$(CFLAGS)。)

我们要创建一个主程序`hist.cc`来调用模块`ia.cc`。两个文件都包含头文件`ia.h`，所以它们将依赖于它。从`hist.cc`和`ia.cc`创建`hist`程序的UNIX *Makefile*文件列在示例23.4中。

示例23.4 *ia/makefile*

```
# Make file needs debugging
CFLAGS = -g -Wall
SRC = ia.cc hist.cc
OBJ = ia.o hist.o

all: hist

hist: $(OBJ)
    g++ $(CFLAGS) -o hist $(OBJ)

hist.o:ia.h hist.cc
    g++ $(CFLAGS) -c hist.cc

ia.o:ia.h ia.cc
    g++ $(CFLAGS) -c ia.cc

clean:
    rm hist io.o hist.o
```

宏SRC是所有C++文件的列表。OBJ是所有目标（.o）文件的列表。命令行：

```
hist: $(OBJ)
    g++ $(CFLAGS) -o hist $(OBJ)
```

告诉make从目标文件建立`hist`程序。如果任一个目标文件过时了，make将重新创建它。

命令行：

```
hist.o:is.h
```

告诉make从`ia.h`和`hist.cc`（`hist.cc`已经被编译过）创建`hist.o`。因为没有指定命令，所以使用缺省命令。

示例23.5显示用于MS-DOS/Windows系统中Turbo-C++环境下的Makefile文件。

示例23.5 *ia/makefile.dos*

```
#
# Makefile for Borland's Turbo-C++ compiler
```

```

#
CC =tcc
#
# Flags
#      -N  -- Check for stack overflow
#      -v  -- Enable debugging
#      -w  -- Turn on all warnings
#      -ml -- Large model
#      -A  -- Force ANSI compliance
#
CFLAGS = -N -v -w -ml -A
#
SRCS = hist.c ia.c
OBJS = hist.obj ia.obj

ia: $(OBJS)
    $(CC) $(CFLAGS) -oia.exe $(OBJS)

hist.obj: hist.cpp ia.h
    $(CC) $(CFLAGS) -c hist.cpp

ia.obj: ia.c ia.h
    $(CC) $(CFLAGS) -c ia.cpp

```

该文件类似于UNIX *Makefile*文件；只有一点不同，Turbo-C++的make不提供任何缺省的规则。

使用make文件有一个很大的弊端。它只检查文件是否被改动，而不看规则是否有修改。如果你在调试时使用CFLAGE=-g命令编译所有的程序，而当使用CFLAGS = -O创建产品版本时，make将不重新编译。

命令touch改变文件的修改时间。（它并不修改文件，它只是让操作系统认为它已经修改了文件。）如果你对像hello.cc这样的文件使用touch命令，然后运行make，则程序将重新编译。如果你已经修改了编译时标志，并想强制重新编译时，这点很有用。

make提供一组创建程序的丰富命令。这里只讨论了很有限的几个。

无限数组的使用

直方图程序(hist)使用无限数组包。它把一个文件作为自变量。该文件含有一串0到99之间的数。输入的任何数都可能被用到。程序输出一个直方图，显示每个数出现的次数。（一个直方图是一个表示数据出现频率的图形。）

这个文件包含几个有意思的编程技巧。

第一个是：只要可能，就让计算机来做。例如，不要编写下面这样的代码：


```

const int LENGTH_X = 300;    // Width of the box in dots
const int LENGTH_Y = 400;    // Height of the box in dots
const int AREA = 12000;      // Total box area in dots

```

这里，程序员来计算 $300 * 400$ 得出了面积。如果让计算机来做乘法会更好：

```

const int LENGTH_X = 300;    // Width of the box in dots
const int LENGTH_Y = 400;    // Height of the box in dots

const int AREA = (LENGTH_X * LENGTH_Y);    // Total box area in dots

```

这样，如果LENGTH_X或LENGTH_Y被修改了，AREA会自动改变。而且，计算机的计算更精确。（如果你注意，你会发现程序员犯了一个错误：他计算的AREA小了10倍。）

在直方图程序中，每个输出行中的数据点数由定义来计算：

```

const float FACTOR = (HIGH_BOUND - LOW_BOUND) / ((float) (NUMBER_OF_LINES-
1));

```

这对用户是很有帮助的。在hist程序中，如果用户在命令行键入的参数个数不正确，则会显示一条信息，告诉正确的命令将是什么。

程序使用库函数memset来初始化counters数组。当把数组的所有元素都设置为0时，该程序的效率非常高。代码行：

```
memset(counters, '\0', sizeof(counters));
```

即把整个数组counters置为0。sizeof(counters)确保数组的所有元素都被初始化。示例23.6中的程序将用无限数组存储产生直方图的数据。

示例23.6 ia/hist.cc

File: hist.cc

```

/*****
 * hist -- generate a histogram of an array of numbers *
 *
 * Usage
 *      hist <file>
 *
 * Where
 *      file is the name of the file to work on
 *****/
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

#include <stdlib.h>
#include <string.h>

```

```

#include "ia.h"

/*
 * The following definitions define the histogram
 */
const int NUMBER_OF_LINES = 50; // # lines in the result
const int LOW_BOUND       = 0;  // Lowest number we record
const int HIGH_BOUND      = 99; // Highest number we record
/*
 * if we have NUMBER_OF_LINES data to
 * output then each item must use
 * the following factor
 */
const int FACTOR =
    ((HIGH_BOUND - LOW_BOUND + 1) / (NUMBER_OF_LINES));

// Number of characters wide to make the histogram
const int WIDTH = 60;

// Array to store the data in
static infinite_array data_array;
// Number of items in the array
static int data_items;

main(int argc, char *argv[])
{
    void read_data(char *name); // Get the data into the array
    void print_histogram(void); // Print the data

    if (argc != 2) {
        cerr << "Error: Wrong number of arguments\n";
        cerr << "Usage is:\n";
        cerr << "  hist <data-file>\n";
        exit(8);
    }
    data_items = 0;

    read_data(argv[1]);
    print_histogram();
    return (0);
}

/*****
 * read_data -- read data from the input file into
 *              the data array
 */

```

```

*
* Parameters
*   name -- the name of the file to read
*
*****/
void read_data(char *name)
{
    ifstream in_file(name); // Input file
    int data;                // Data from input

    if (in_file.bad()) {
        cerr << "Error: Unable to open " << name << '\n';
        exit(8);
    }
    while (!in_file.eof()) {
        in_file >> data;

        // If we get an eof we ran out of data in last read
        if (in_file.eof())
            break;

        data_array[data_items] = data;
        ++data_items;
    }
}

/*****
* print_histogram -- print the histogram output
*
*****/
void print_histogram(void)
{
    // Upper bound for printout
    int counters[NUMBER_OF_LINES];
    int low;                // Lower bound for printout
    int out_of_range = 0; // Number of items out of bounds
    int max_count = 0; // Biggest counter
    float scale;           // Scale for outputting dots
    int index;             // Index into the data

    memset(counters, '\0', sizeof(counters));

    for (index = 0; index < data_items; ++index) {
        int data; // Data for this point

        data = data_array[index];

        if ((data < LOW_BOUND) || (data > HIGH_BOUND))

```

```

        ++out_of_range;
    else {
        // Index into counters array
        int    count_index;

        count_index = int (float(data - LOW_BOUND) / FACTOR);

        ++counters[count_index];
        if (counters[count_index] > max_count)
            max_count = counters[count_index];
    }
}

scale = float(max_count) / float(WIDTH);

low = LOW_BOUND;

for (index = 0; index < NUMBER_OF_LINES; ++index) {
    // Index for outputting the dots
    int    char_index;
    int    number_of_dots;    // Number of * to output

    cout << setw(2) << index << ' ' <<
        setw(3) << low << "- " <<
        setw(3) << low + FACTOR - 1 << " (" <<
        setw(4) << counters[index] << "): ";

    number_of_dots = int(float(counters[index]) / scale);
    for (char_index = 0; char_index < number_of_dots;
        ++char_index)
        cout << '*';
    cout << '\n';
    low += FACTOR;
}
cout << out_of_range << " items out of range\n";
}

```

运行该程序，它输出：

```

% hist test
0: 0- 2 ( 100): *****
1: 2- 4 ( 200): *****
2: 4- 6 ( 100): *****
3: 6- 8 ( 100): *****
4: 8- 10 (   0):
5: 10- 12 ( 100): *****

```

```

6: 12- 14 ( 50): *****
7: 14- 16 ( 150): *****
8: 16- 18 ( 50): *****
9: 18- 20 ( 50): *****
10: 20- 22 ( 100): *****
11: 22- 24 ( 100): *****
12: 24- 26 ( 50): *****
13: 26- 28 ( 100): *****
14: 28- 30 ( 50): *****
15: 30- 32 ( 100): *****
16: 32- 34 ( 50): *****
17: 34- 36 ( 0):
18: 36- 38 ( 100): *****
19: 38- 40 ( 1):
20: 40- 42 ( 150): *****
21: 42- 44 ( 50): *****
22: 44 - 46 ( 250): *****
23: 46- 48 ( 100): *****
24: 48- 51 ( 150): *****
25: 51- 53 ( 100): *****
26: 53- 55 ( 50): *****
27: 55- 57 ( 200): *****
28: 57- 59 ( 50): *****
29: 59- 61 ( 50): *****
30: 61- 63 ( 50): *****
31: 63- 65 ( 150): *****
32: 65- 67 ( 100): *****
33: 67- 69 ( 0):
34: 69- 71 ( 199): *****
35: 71- 73 ( 200): *****
36: 73- 75 ( 100): *****
37: 75- 77 ( 50): *****
38: 77- 79 ( 100): *****
39: 79- 81 ( 100): *****
40: 81- 83 ( 200): *****
41: 83- 85 ( 100): *****
42: 85- 87 ( 0):
43: 87- 89 ( 0):
44: 89- 91 ( 50): *****
45: 91- 93 ( 150): *****
46: 93- 95 ( 100): *****
47: 95- 97 ( 50): *****
48: 97- 99 ( 100): *****
49: 99-101 ( 0):
500 items out of range

```

把任务划分为模块

不幸的是，计算机编程是科学加艺术。没有任何严格快速的规则可以教你如何把任务划分为模块。知道哪些能产生好模块，而哪些不能产生好模块只能来自于经验和实践。

本节介绍一些用于模块划分的一般规则，并讨论它们如何适用于真正的程序。这里描述的技巧非常适合于我。你可以选择适用于你的规则来用。

信息是任何程序的要害。一个程序的关键是你决定想使用什么样的信息及你想对信息进行哪些处理。在开始设计之前，一定要先分析信息。

模块应设计为：模块间必须传递的信息量最少。如果你看看军队的组织，可以发现，它们划分为模块。这些模块是步兵、炮兵、坦克兵团等等，它们之间传递的信息量应最小。例如，如果一名步兵中士想让炮兵轰炸敌方一位置，他就向炮兵发命令：“位置Y-94有一个碉堡。消灭它。”炮兵司令官处理所有细节，他决定使用哪个炮兵连、根据其它任务的需要投入多少火力、后续支援，以及其它更详细的问题。*

程序应该用同样的方法来组织，信息应该放在应放之处。模块间的通讯量越少，就可能减少通讯错误，也可以减少模块升级时的维护所带来的问题。

模块划分示例：文本编辑器

你已经很熟悉文本编辑器的使用了。它是可以让用户显示并修改文本文件的一个程序。信息的主要部分是我们正在编辑的文本文件。大多数编辑器都着重于显示，并在屏幕上连续地显示当前文件中大约24行的内容。

最后，还需要一条更重要的信息：编辑命令。命令由用户键入。该信息必须进行语法分析，这样，计算机才能懂。然后，它可以被执行。单个的命令很小，并且都执行类似的功能（`delete line`非常类似于`delete character`）。命令执行模块中采用标准结构的话，可以改善可读性和可靠性。编辑器中的一部分图示于图23.2中。

模块之间仅需要最少量的通讯。显示部分只需要知道两件事：1)光标在哪，2)文件当前的样子。文件模块要做的只是读文件、写文件并记录下修改的历史。即使是修改的方式也可以最少化。所有的编辑命令，不论多么复杂，都可以分解为一系列的插入和删除。这由命令模块负责，它读入复杂的用户命令，并把它们转为简单的插入和删除操作，这些命令可以交给文件处理器来处理。模块之间传送的信息要最少。命令解码器和显示模块之间没

* 这只是真正部队的一个非常简单的划分。真正的部队系统很复杂，各有各的特色，即使将军也不能谈仔细。

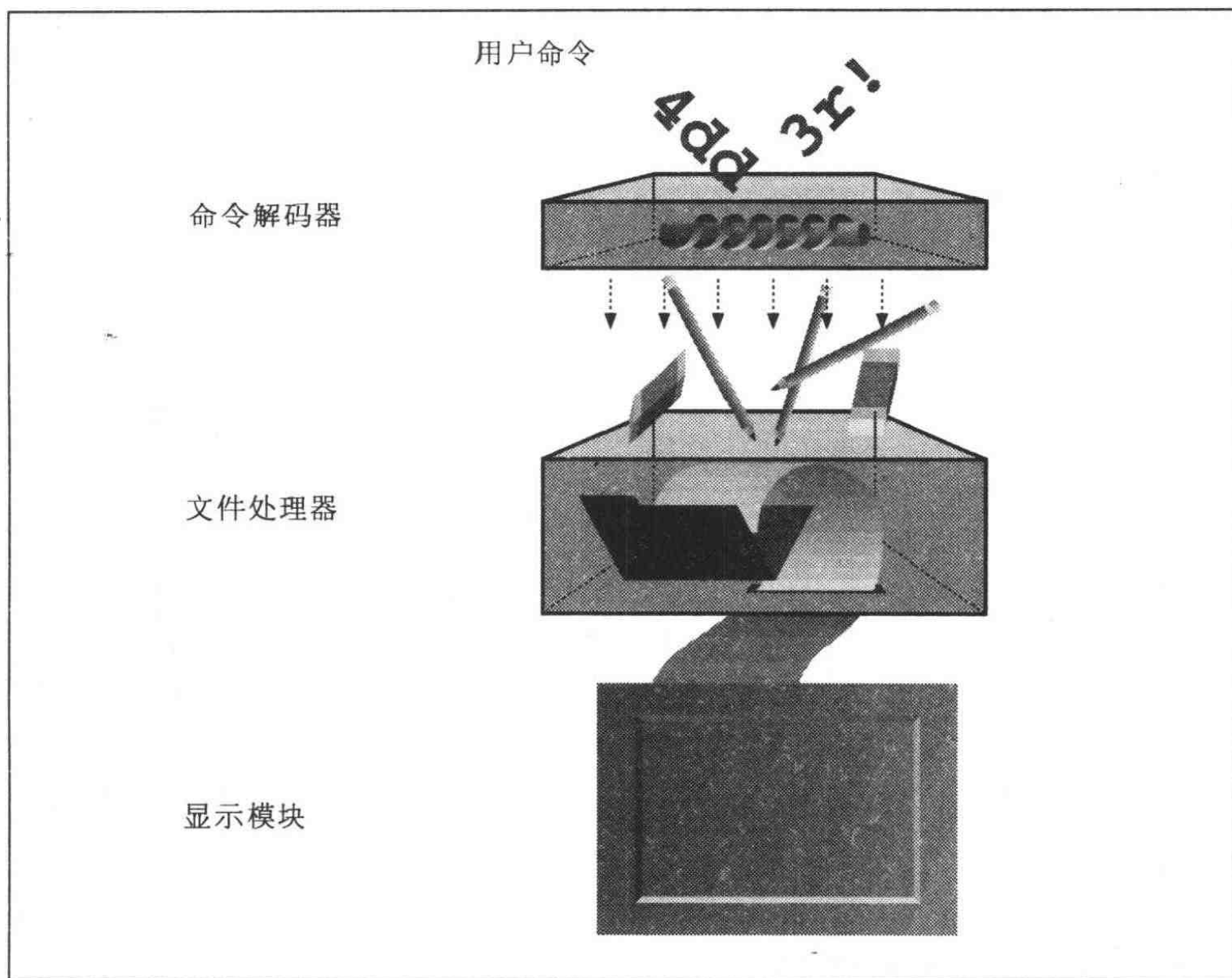


图 23.2 文本编辑器的模块划分

有信息传递。

字处理器就是一个奇特的文本编辑器。一个简单的编辑器可以只关注于ASCII字符（一种字体，一种大小），可是，字处理器必须能处理许多种不同大小和形状的字符。

编译器构造

在一个编译器中，要处理的信息是C++代码。编译器的工作是把这些信息从C++源程序转换为依赖机器的目标代码。在这个过程中有这样几步。首先，预处理器要运行代码，把宏展开，注意条件编译，并读入包含文件。下一步，要处理的文件传给编译器的第一个阶段——词法分析器。

词法分析器把它的输入看作字符流，并返回一串记号(token)。记号是一个计算机学术语，其含义是一个字或是一个运算符。例如，让我们看看英语命令：

Open the door.

该命令中有14个字符。词法分析将把它转为三个字和一个句号。然后这些符号将送给语法分析器，在那儿，它们将被汇编成句子。在这一步，要造一个符号表，这样语法分析器可以知道程序中要用到哪些变量。

现在编译器知道了程序打算要做什么。优化程序检查指令，并试着找出如何能使命令更高效。这一步即是代码的优化，除非在命令行中使用了-O标志，否则，这一步将略过。

代码生成器把高级语句转为机器特定的汇编代码。在汇编语言中，每个汇编语言语句都对应为一条机器指令。汇编器把汇编语言转为机器可以执行的二进制代码。一般的编译器信息流如图23.3所示。

词法分析和语法分析很常见，它们广泛地用于程序中。实用程序lex可以为一个程序生成词法分析器模块，给出程序中所用记号的描述。另外一个实用程序yacc，可以产生语法分析模块。

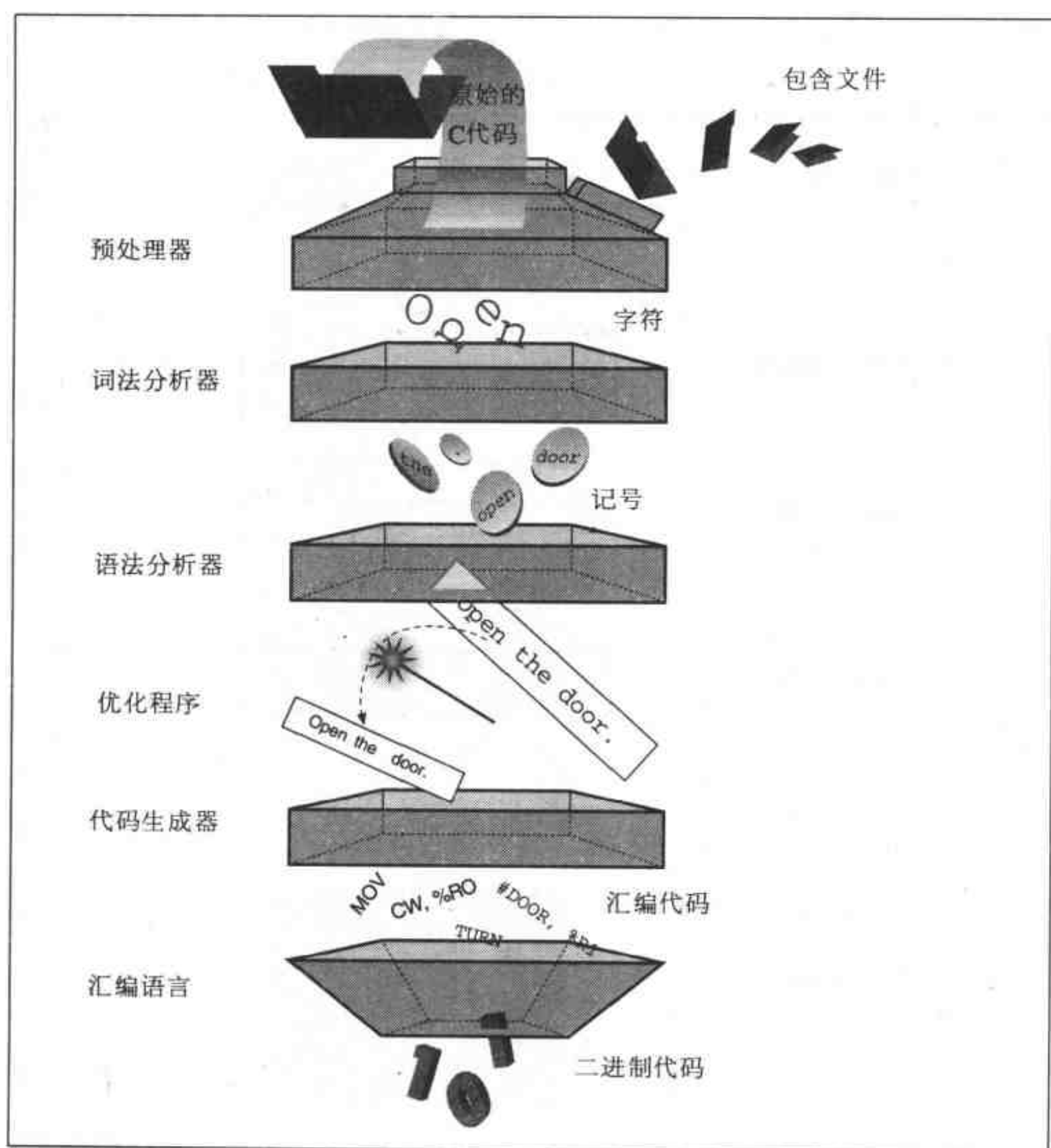


图 23.3 编译器模块

电子表格

简单的电子表格接受一个矩阵，矩阵元素是数和等式，并把结果显示在屏幕上。该程序管理的信息是等式和数据。

电子表格的核心是等式。要把等式修改为数，必须进行词法分析和语法分析，就像一个编译器一样。

但和编译器不一样的是，我们不必产生机器代码。相反，我们解释执行等式，并计算结果。

结果将传给显示管理器，它把这些结果显示在屏幕上。另外，还加上一个输入模块，它将允许用户编辑并修改等式，已有的一个电子表格如图23.4所示。

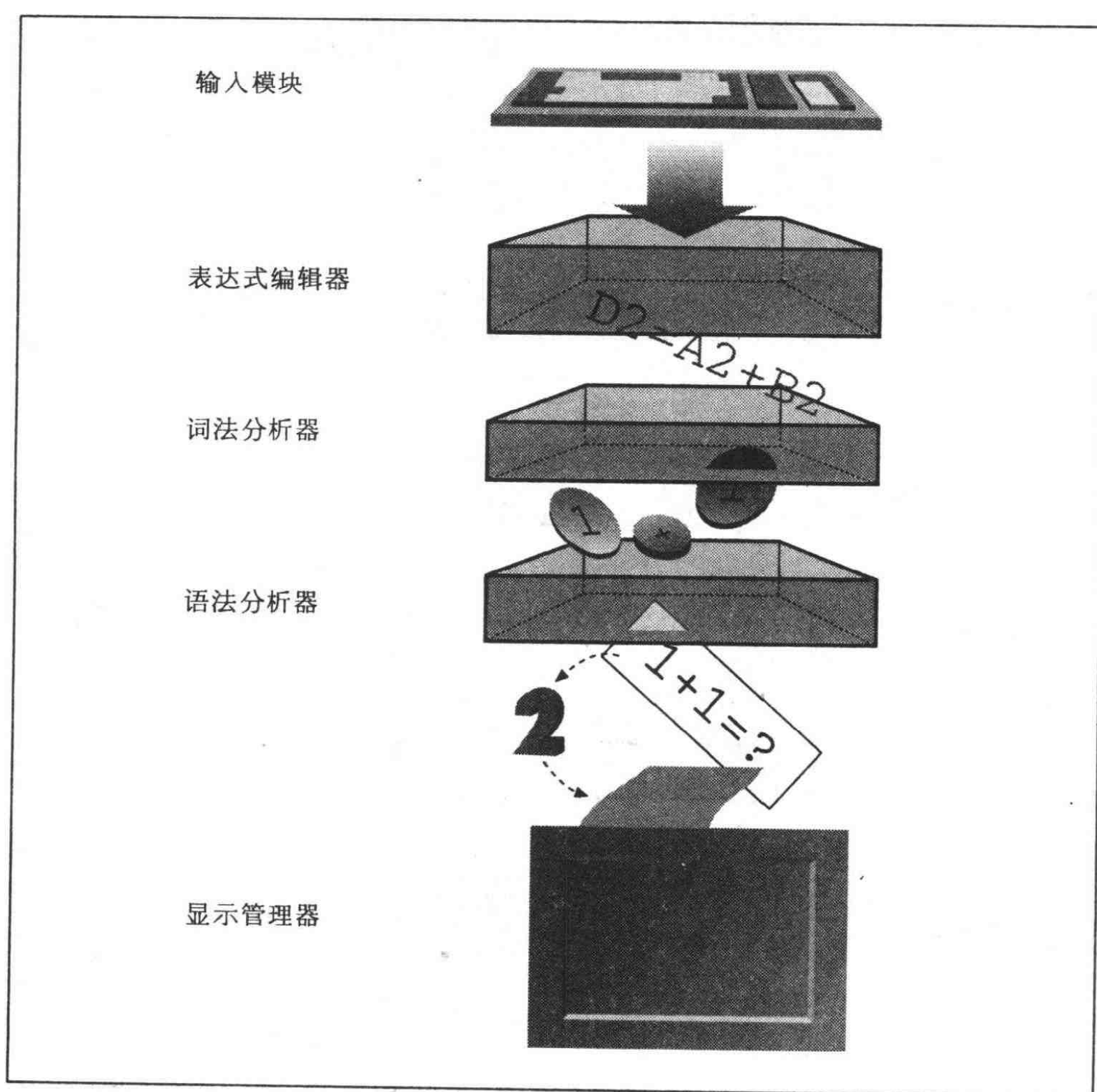


图 23.4 电子表格层

模块设计准则

虽然在为一个程序划分模块时没有严格快速的规则，但有几个一般的准则。

- 一个模块中的公用函数数目要少。
- 模块间传递的信息要少。
- 一个模块中的所有函数应执行相关的任务。
- 模块中的代码行不应多于1500行。当有更多的代码时，编辑、打印和理解时就很困难。

编程练习

练习23.1：编写一个处理页面格式的类。它将包含下列函数：

`open_file(char *name)` 打开打印文件。

`define_header(char *heading)` 定义头文件文本。

`print_line(char *line)` 向文件发送一行。

`page(void)` 开始一新页。

`close_file(void)` 关闭打印文件。

练习23.2：编写一个模块`search_open`。它先接受一个文件名数组，在已有文件中进行搜索，找到后，打开该文件。

练习23.3：编写一个符号表`class`，它含有下列函数：

`void enter(char *name)` 把一个名字送入符号表中。

`int lookup(char *name)` 如果表中已有该名字，返回1；否则返回0。

`void remove(char *name)` 从符号表中删除一个名字。

练习23.4：将第20章“高级指针”中的`words`程序，与无限数组模块联合起来，创建一个交叉引用程序。（顺便，让其了解C++注释和串的概念，以创建一个C++交叉引用程序。）

本章内容:

- 什么是模板
- 模板: 严格的方法
- 函数特例
- 类模板
- 类特例
- 实现困难
- 小结
- 编程练习

第24章

模 板

什么是模板

模板是C++新加入的内容。它允许你为几种不同的数据类型写一般的类和函数。

当程序增大时, 模板是C++语言中非常有用的部分。但是, 虽然ANSI C++标准草案规定了模板的完整语法, 但是却没有说明如何实现它们。结果, 每个编译器的实现方法都不相同, 所以使用模板的程序也不可移植。

模板: 严格的方法

假定我们想定义一个函数max, 它返回两个项中最大者。实际上, 我们并不想只定义一个max函数, 而是想定义一系列函数: 找出两个整数中最大者, 找出两浮点数中最大者, 找出两个字符中最大者, 等等。

我们先定义一个带参的宏来产生函数的代码。它称为定义阶段(*definition stage*)。宏的定义如下:

```
#define make_max(type) type max(type d1, type d2) ( \
    if (d1 > d2) \
        return (d1); \
    return (d2); \
)
```

注 意

除了最后一行外, 每行的结尾都是反斜杠 (\)。#define宏占了多行, 所以反斜杠表示这五行是一行。在同一列放置反斜杠, 是为了提醒我们不要漏掉一个。

该宏不产生代码。它只是提供用在下一阶段的定义, 下一阶段称为生成阶段(*generation phase*), 产生我们想要的函数。

```

define_max(int);
define_max(float);
define_max(char);

```

最后，在代码中使用已经定义过的函数。（当然该阶段称为使用阶段(use phase)。）

```

main(void) {
    flaoat f = max(3.5, 8.7);
    int    i = max(100, 800);
    char   ch = max('A', 'Q');
}

```

图24.1显示#define类型模板的源代码及由此生成的代码。

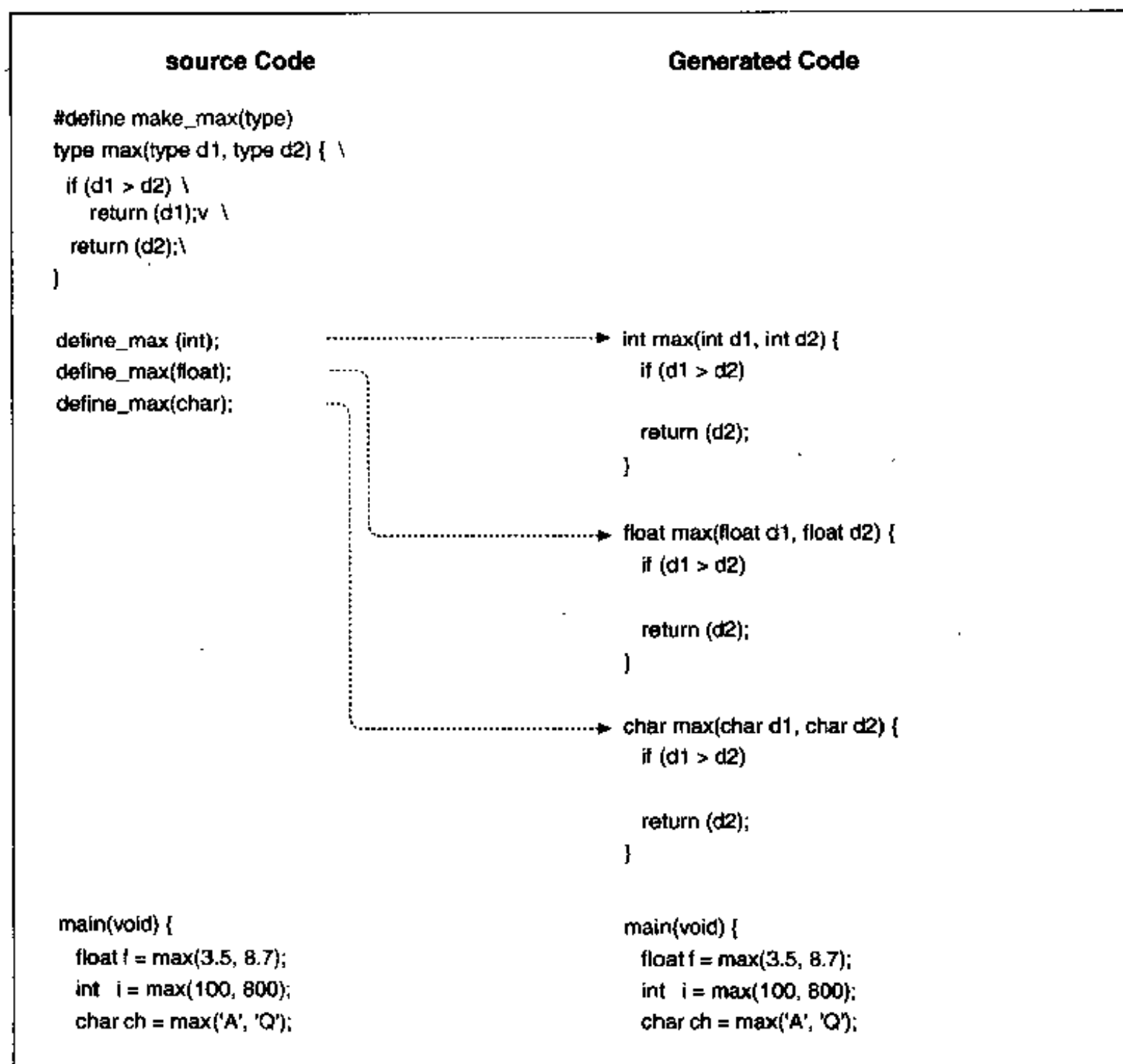


图 24.1 由#define 类型模板生成的代码

该方法很适用于像max这样的简单函数。它不太适用于较大的函数。该系统的一个缺点是，我们必须为想使用的每一数据类型调用define_max宏。如果C++能自动调用define_max就好了。

模板允许你定义一个一般函数。然后C++使用模板生成所需函数的一个具体实例。例如，下列代码把函数max定义为模板：

```
template<class kind>
kind max(kind d1, kind d2) {
    if (d1 > d2)
        return (d1);
    return (d2);
}
```

注意

结构<class kind>告诉C++，字kind可以被任一种类型所替代。（注：文中使用的关键字class表明，kind可以是任何的类型：不仅是类，也可以是简单类型。）

template说明相当于带参宏的定义。和带参宏一样，它不生成代码，它只提供用于下一阶段的定义。

现在可以使用模板了，这很像我们使用由带参宏定义的函数：

```
main(void) {
    float f = max(3.5, 8.7);
    int i = max(100, 800);
    char ch = max('A', 'Q');
    int i2 = max(600, 200);
}
```

你可能已经注意到了，我们跳过了代码生成阶段。这是因为C++自动为我们生成代码。换句话说，C++看到代码：

```
float f = max(3.5, 8.7);
```

之后，明白它使用函数max(float, float)。然后，它检查是不是生成了该函数的代码；如果需要的话，生成它。换句话说，每件事都自动完成。（自动完成的事情也有一些实际的限制，在实现部分我们将详细讨论。）

图24.2显示由max实现模板生成的代码。从这里你可以看出，第一次max用于浮点数，它生成的是用于浮点数的max函数。下一次，max用于整数，则建立用于整数的max函数。注意最后一行：

```
int i2 = max(600, 200);
```

并没有生成代码。这是因为我们已经生成了整数max函数，不需要再生成一次。

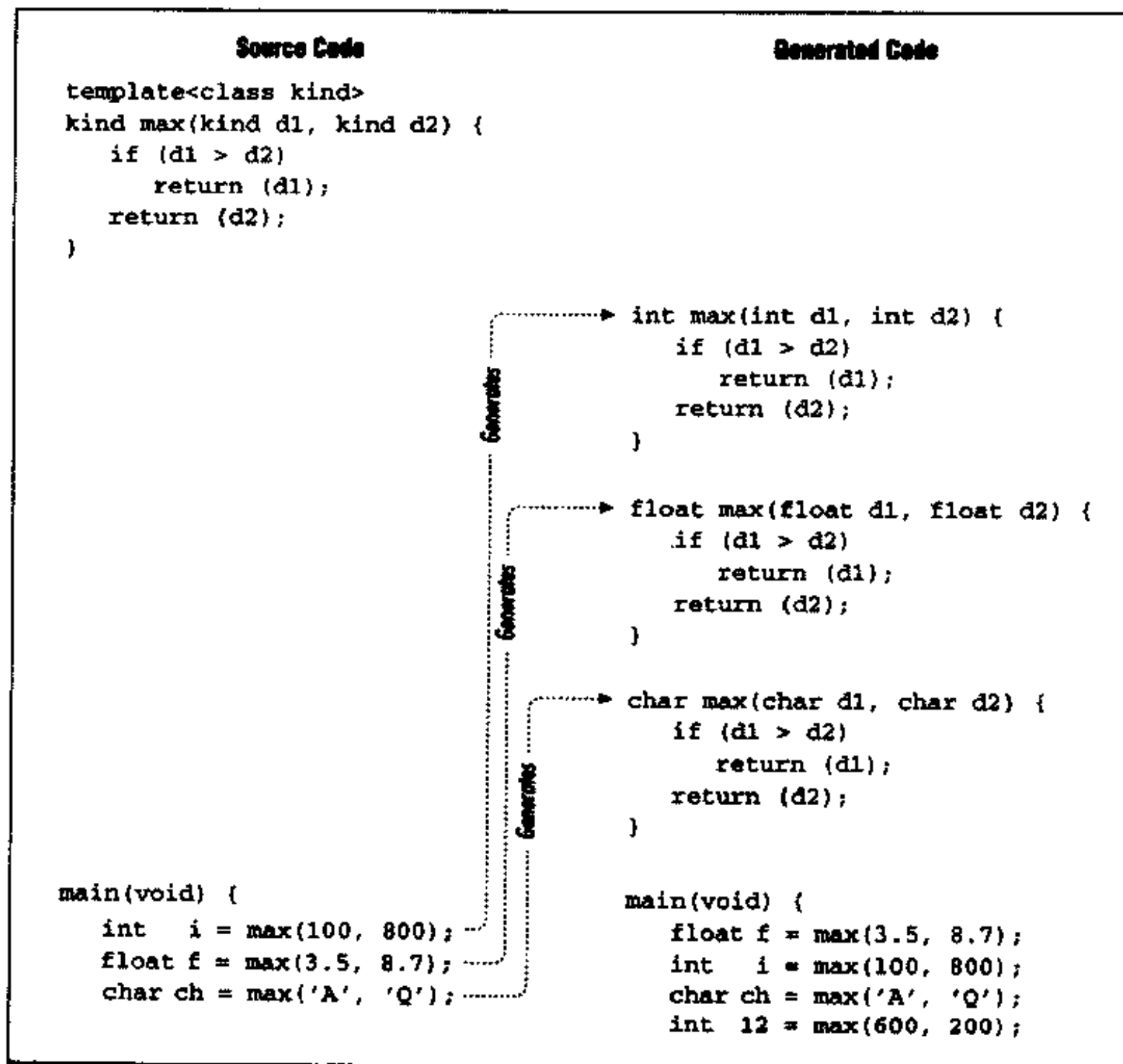


图 24.2 模板代码生成

函数特例

模板不仅仅是简单的代码生成。它们还可以处理特殊的情形。

假定我们想用`max`函数来比较串：

```
char *name1 = "Able";
char *name2 = "Baker";
```

```
cout << max(name1, name2) << '\n';
```

我们有一个问题，因为串表示为一个字符指针（`char *`）。表达式：

```
if (d1 > d2)
```

将比较指针的值，而不是它们指向的数据的值。我们是想告诉C++，“当数据类型是一个串时使用strcmp，其余使用正常的比较。”

这由一个称为特例(Specialization)的过程来完成。我们说明只用于串的max的一个特殊版本：

```
char *max(char *d1, char *d2) {
    if (strcmp(d1, d2) < 0)
        return (d1);
    return (d2);
}
```

当C++第一次见到max函数的使用时，它先查看简单函数列表，然后查看模板列表。所以当我们使用代码：

```
cout << max(name1, name2) << '\n';
```

时，C++在试图扩展模板max(kind d1, kind d2)之前，找到了简单函数max(char *, char *)。

示例24.1说明模板函数的使用。

```
示例24.1 max-t/max.cc
#include <iostream.h>
#include <string.h>

// A template for the "max" function

template<class kind>
kind max(kind d1, kind d2) {
    if (d1 > d2)
        return (d1);
    return (d2);
}

// A specialization for the "max" function
// because we handle char * a little differently
char *max(char *d1, char *d2) {
    if (strcmp(d1, d2) > 0)
        return (d1);
    return (d2);
}

main()
{
    // Let's test max
```



```

    cout << "max(1, 2) " << max(1, 2) << '\n';
    cout << "max(2, 1) " << max(2, 1) << '\n';

    cout << "max(\"able\", \"baker\") " << max("able", "baker") << '\n';
    cout << "max(\"baker\", \"able\") " << max("baker", "able") << '\n';
    return (0);
}

```

类模板

类模板比函数模板要复杂一些。它们的说明很容易。它们的定义很像函数模板的定义。示例24.2中用模板实现了第13章“简单类”中的stack类。

示例24.2 max-t/stack1.cc

```

#include <stdlib.h>
#include <iostream.h>

const int STACK_SIZE = 100;    // Maximum size of a stack

/*****
 * Stack class
 *
 * Member functions
 *   stack -- initialize the stack
 *   push -- put an item on the stack
 *   pop -- remove an item from the stack
 *****/
// The stack itself
template<class kind>
class stack {
    private:
        int count;           // Number of items in the stack
        kind data[STACK_SIZE]; // The items themselves
    public:
        // Initialize the stack
        stack(void) {
            count = 0; // Zero the stack
        }

        // Push an item on the stack
        void push(const kind item) {
            data[count] = item;
            ++count;
        }
}

```

```

        // Pop an item from the stack
        kind pop(void) {
            // Stack goes down by one
            --count;

            // Then we return the top value
            return (data[count]);
        }
    };

```

但是，这里有个问题。要使用类，我们必须说明该类的一个实例。过去，我们可以用下列语句说明一个栈：

```
stack a_stack;    // This won't work
```

问题是，现在stack是一个一般模板。现在栈可以包含任何内容。当C++见到这个说明时，它就要问：“栈是什么？”我们必须指明要存储的数据类型。新的说明为：

```
stack<int> a_stack;    // A stack of integers
```

<int>告诉C++在栈中用“int”来替代“kind”。现在可以使用新的类变量了：

```

a_stack.push(1);
x = a_stack.pop();

```

在stack类中，我们在类定义内部定义了所有的成员函数。我们也可以在类外指明过程。为此，必须在每个过程前写上模板子句template<class kind>，并在类名中写上模板参数(<kind>)。例如，pop程序如下：

```

/*****
 * stack::push -- push an item on the stack          *
 *                                                    *
 * Warning: We do not check for overflow              *
 *                                                    *
 * Parameters                                          *
 *      item -- item to put on the stack              *
 *****/
template<class kind>
inline void stack<kind>::push(const kind item)
{
    data[count] = item;
    ++count;
}

```

类特例

你可以把如下的类模板看作是一条指令：

```
template <class kind>stack {...
```

它告诉C++如何生成一组名为`stack<int>`、`stack<double>`、`stack<float>`等的类。C++还自动生成成员函数：`stack<int>::push`、`stack<double>::push`和`stack<float>::push`。

但是，如果你自己明确地说明了一个成员函数，C++在生成自己的函数之前，将使用你定义的函数。假定我们想用栈来存放串（`char *`）。我们不想存放指针，而想存放真正的串。要做到这些，我们要写一个特殊的`push`函数，它在把串压到栈中之前，先复制该串：

```
inline void stack<char *>::push(const char * item)
{
    data[count] = strdup(item);
    ++count;
}
```

注意，我们在函数的开头并没有使用`template<class kind>`。`template`关键字告诉C++：“这是一个一般的类，由它生成特殊的代码。”不使用`template`，则告诉C++的是：“这是真正的函数。直接使用它。”

实现困难

Ellis和Stroustrup所著的《Annotated C++ Reference Manual》被公认为是C++语言的隐含标准。它含有`templates`语法的完整定义。但是，它没有解释编译器将如何实现它们。

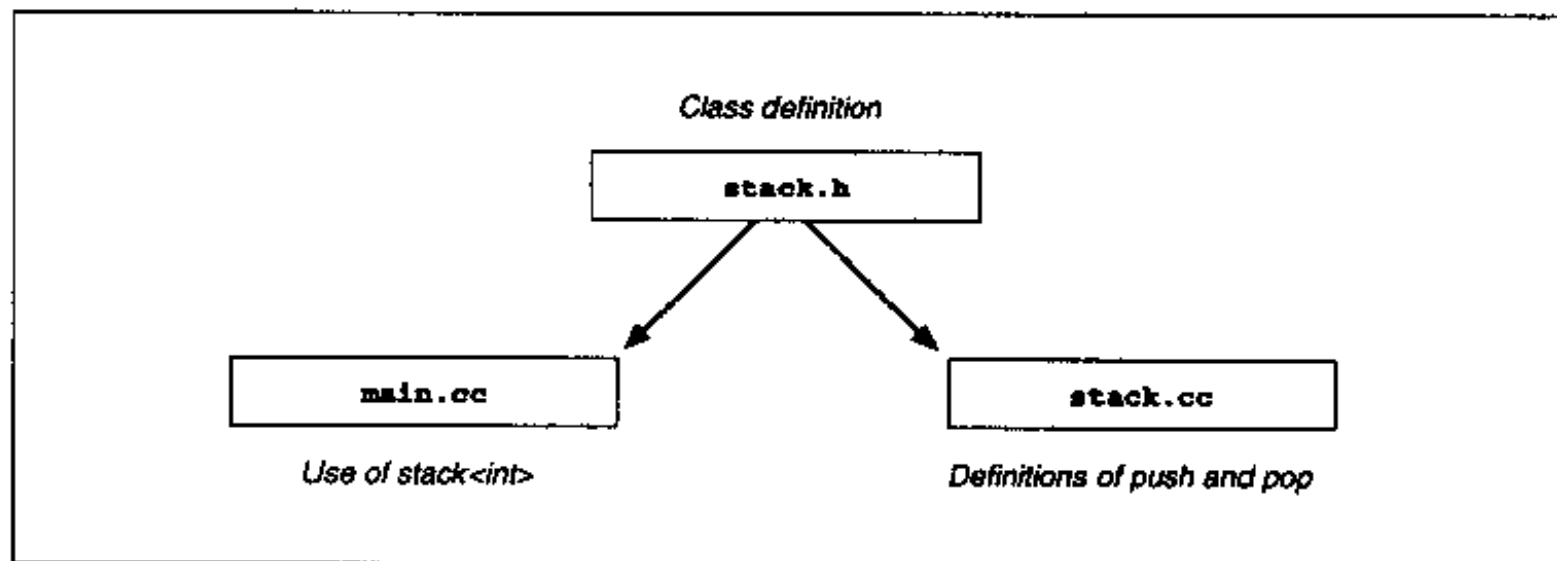


图 24.3 使用stack模板的源代码层

正如你将看到的，`templates`并不简单。例如，假定我们把`stack`模板放到一个模块中。文件`stack.h`定义类，而文件`stack.cc`定义成员函数。（我们假定成员函数都不是内联的。）



现在，我们想在文件`main.cc`中使用用于浮点栈（`stack<float>`）的模板。图24.3所示为源代码层。

这正是编译器实现时困难之所在。

我们知道，在`main.cc`中，我们要使用`stack<int>`。事实上，文件`stack.h`定义了两个成员函数`push`和`pop`的原型。所以，当编译器见到`stack<int>`时，它将自动生成`push`和`pop`的代码。

这两个成员函数体隐藏在文件`stack.cc`中。编译器正在处理`main.cc`，它不了解其它文件的内容，所以它不知道如何生成`push<int>`和`pop<int>`。

但是当编译器编译`stack.cc`时，它能不能生成`push<int>`和`pop<int>`呢？如果它知道已经说明了`stack<int>`变量，回答是肯定的。但是说明该变量的正是`main.cc`文件，而编译器现在并没有处理这个文件。管理人员把这个现象称为“技术挑战”，程序员称之为“恶梦”。

涉及模板生成的问题不容易解决。不同的编译器编制人员已经选择了不同的方法。问题是，所有这些方法都不兼容。为一个编译器所写的代码可能不能用于另一个编译器。有时，即使为编译器的一个版本而写的代码，都可能不能用于该编译器的另一个版本。

模板是C++编译器中的尖端技术。和所有新程序一样，它们也存在一些错误。在接下来的内容中，我们讨论编译器编制人员解决这些问题所采取的不同方法。编译器技术和标准一直在发展，所以本节讨论的内容可能已过时了。

微软公司的实现

到本书出版时，Microsoft C++和Visual C++编译器并没有模板。虽然非常简单，但这种“实现”方式太有限了。换句话说，我使用Microsoft编译器时从没有遇到过任何麻烦。

Turbo-C++和GNU g++模板

Turbo-C++和GNU g++编译器的实现方法很相似。如果你想使用`stack<float>`的话，必须在定义成员函数的文件：`stack.cc`中有`stack<float>`。在`stack.cc`文件的开头，要写下面这行代码：

```
typedef stack<float> floating_point_stack;
```

实际上，你从不用类型`floating_point_stack`。这里的`typedef`只是告诉编译器生成所需的成员函数。

这两个编译器都有开关，用来告诉编译器如何对待不能处理为内联函数的内联函数。假定你在头文件`max.h`中定义了函数：

```
template<class kind>
inline kind max(kind d1, kind d2) {
    if (d1 > d2)
        return (d1);
    return 9d2);
```

}

然后在三个不同文件中使用`max<float>`。还假定函数太复杂了，编译器决定忽略`inline`命令。（这有点太主观了，但这是说明这个问题的唯一办法。）

当编译你的三个文件时，生成三个`max<float>`。Turbo-C++连接器能发现这个事实，并删除其中的两个。GNU g++连接器不太好，在你的代码中你会发现三个同样的程序。

Turbo-C++和GNU都为你提供了编译器选项，可以有助于你解决这个问题。如果你在代码中写这样的一行：

```
#pragma interface // GNU g++ only option
```

则它告诉GNU：“不要为该头文件中的所有模板函数生成代码体。”稍后，在你的文件中，你必须写入命令：

```
#pragma implementation // GNU g++ only option
```

它告诉GNU：“这是生成代码体的地方了。”

Turbo-C++使用类似的`pragma`。但是“接口”命令是：

```
#pragma option -Jgx // Turbo-C++ only option
```

而“实现”命令是：

```
#pragma option -Jg // Turbo-C++ only option
```

选项很笨，不可移植，使用也较困难。这意味着，实现问题解决得并不太好。

基于CFront的编译器

CFront是AT&T C++到C转换器的名字，它是第一个C++编译器的基础。大多数C++编译器编制人员都考虑他们自己的标准是否与最新的CFront相兼容。

CFront处理模板生成问题的方法是，在连接之前不考虑它，所以你只能编译并连接文件`main.cc`。到连接时，CFront注意到`main.cc`使用了`stack<float>`。此时它才去查找`stack.cc`文件。（注：你的文件必须叫`<class>.cc`。）该文件用于模板的生成。

虽然这种方法看上去简单，但要注意很多细节问题。例如，当连接器要去编译`stack.cc`时，将自动包含文件`stack.h`。但是，如果你用`#include "stack.h"`语句自己包含该文件的话，编译器就弄不清楚了。

CFront方法是解决模板问题的好方法，但是也仍有不足之处。

编写可移植的模板

如何编写可移植的模板呢？简单的答案是：“不要用它们。”但是，创建真正可移植模板的最好方法是把所有的函数都定义为内联函数，并把它们放到单一的一个头文件中。到目前为止，我可以说，这个方法适用于有模板的每一种编译器。它可能不是最有效的方

法，但却具有最佳移植性。

小 结

模板提供了写一般类和函数的便利方法。但是，模板的实现仍在努力之中。实际上，你可能想等语言有了一定之规之后再使用模板。

编程练习

练习24.1：编写一个模板`min`，它返回两个值中的最小者。确保你能正确地处理串。

练习24.2：编写一个模板类，实现任何数组的边界检查。

练习24.3：定义一个模板类，实现一个集合。类允许你设置、清除并检测集合元素。（这个类的整数版本请见示例13.2。）

本章内容:

- 模块化
- 字大小
- 字节顺序问题
- 对齐问题
- NULL指针问题
- 文件名问题
- 文件类型
- 小结
- 答疑

第25章

移植问题

你已经完成了一件杰作，在Cray超级计算机上，用30MB内存和5GB硬盘空间，表示复杂三维阴影图形的光线描绘程序。当某个人要求你把这个程序移植到仅有640K内存和100MB硬盘空间的IBM PC上时，你有何感想？“杀”了他。这不仅是非法的，而且，也不专业。你唯一的选择只有抱怨，然后开始移植。移植过程中，你会发现原来工作良好的程序有许多奇怪的不可思议的问题。

一般认为C++程序是可移植的，但是C++包含许多依赖机器的特性。而且，由于UNIX和MS-DOS/Windows之间存在巨大的不同，所以在许多程序中常常发现与系统相关的特征。本章讨论写真正可移植程序时有关的一些问题，也讨论你可能遇到的一些陷阱。

模块化

编写可移植程序的一个技巧是，把所有不可移植代码放到一个模块中。例如，在MS-DOS/Windows和UNIX系统中，屏幕处理存在较大差异。要设计一个可移植程序，你只能为不同的机器编写各自的屏幕处理模块。

例如，HP-98752A终端有一组功能键，标记为F1到F8。PC也有一组功能键。问题是，这些键对应的代码不一样。HP中F1键发出的代码是“<esc>p<return>”，而PC机发出的是“<null>”。这种情况下，你会想写一个get_code程序，它从键盘得到一个字符（或是功能键串），并转换功能键。因为转换依机器的不同而不同，所以需要为每种机器编写一个依赖于机器的模块。对于HP机器，你可能把程序放到main.cc和hp-tty.cc中，而在PC中，你使用的可能是main.cc和pc-tty.cc。

字大小

长整数为32位，短整数为16位*。一般的整数可能是16位，也可能是32位，这要看使用

的是哪种机器。这可能带来意想不到的问题。例如，下面的代码适用于32位的UNIX系统，但移植到MS-DOS/Windows上时失败了：

```
int zip;

zip = 92126;
cout << "zip code " << zip << '\n';
```

问题出在，在MS-DOS/Windows系统中，zip只有16位——不够存储92126。为了解决这个问题，我们把zip说明为32位的整数：

```
long int zip;

zip = 92126;
cout << "zip code " << zip << '\n';
```

现在zip是32位的变量，可以存储92126了。

字节顺序问题

短整数包含两个字节。考虑一下数0x1234，两个字节中的值分别是0x12和0x34。第一个字节中存储的是哪个值？回答是这将依机器而定。

当你想写可移植的二进制文件时，这可能带来很大的麻烦。Motorola 68000系列机器使用的字节顺序为（ABCD），而Intel和DEC机器使用另外一种字节顺序（BADC）。

对于二进制文件的一种解决方法是，避免使用它，而让你的程序读写ASCII码文件。ASCII码文件有两方面的好处，一是程序更易于移植，二是程序更具有可读性。

不好的一点是正文文件要长一些。有些文件使用ASCII码时太大了。这种情况下，文件的开始所放的魔术数很有用。假定魔术数是0x11223344（虽是一个坏魔术数，却是一个好例子）。当程序读到这个数时，它能够检测出字节正确的顺序。程序可以自动解决这个问题；

```
const int MAGIC      = 0x11223344; // File identification number
const int SWAP_MAGIC = 0x22114433; // Magic number byte swapped

ifstream in_file; // File containing binary data
long int magic;    // Magic number from file
```

* ANSI标准草案没有指明长整数和短整数的真正大小。但是，据我所知的每种机器上，长整数都是32位，短整数是16位。

```

in_file.open("data");

in_file.read((char *)&magic, sizeof(magic));

switch (magic) {
    case MAGIC:
        // No problem
        break;
    case SWAP_MAGIC:
        cout << "Converting file, please wait\n";
        convert_file(in_file);
        break;
    default:
        cerr << "Error: Bad magic number " << magic << '\n';
        exit (8);
}

```

对齐问题

有些计算机限制了可用于整数和其它类型数据的地址。例如，68000系列机要求所有的整数要两字节两字节的对齐。如果你想用奇数地址访问一个整数，会得到一个错误。有些处理器没有对齐规则，而有些则更严格些，要求整数对齐四字节。

对齐规则不仅限于整数，浮点数和指针也必须正确对齐。

C++向你隐藏对齐规则。例如，如果你在68000上说明下列结构：

```

struct funny {
    char    flag;    // Type of data following
    long int value;  // Value of the parameter
};

```

C++为该结构分配的地址如图25.1中左图所示。

在8086系列机上，没有对齐规则，它的分配如图25.1中右图所示。

问题是，结构的大小将依机器而改变。在68000上，结构大小为6个字节，而在8086上，是5个字节。所以如果你在68000上写一个包含100个记录的二进制文件，它将有600个字节长；而在8086上，它只有500字节长。很明显，在两种机器上不能用同一方法写文件。

一种解决方法是使用ASCII码文件。正如我们前面提到的，二进制文件存在许多问题。另一个解决方法是明确地说明一个pad字节：

```

struct new_funny {
    char    flag;    // Type of data following
    char    pad;     // Not used
};

```

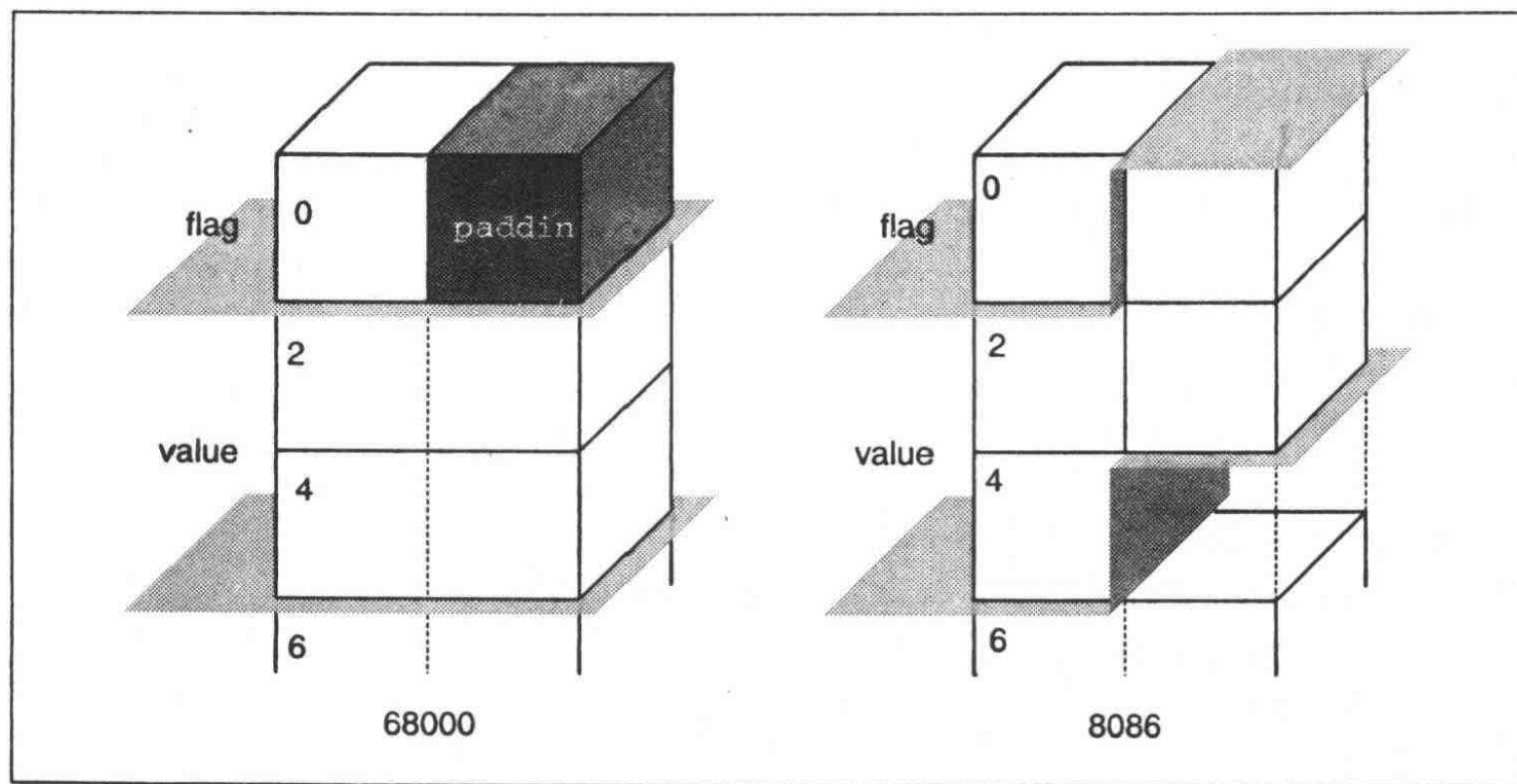


图 25.1 在68000和8086体系结构下的结构

```
long int value; // Value of the parameter
};
```

pad字符使得在68000机器上正确地对齐域值；而在8086机器上，能保证结构的大小正确。

使用**pad**字符很困难，也易出错。例如，虽然**new_funny**在32位整数对齐一或二字节的两种机器上是可移植的，但是，它不能移植到四字节对齐的机器上。

NULL指针问题

许多程序和应用程序是使用UNIX在VAX计算机上书写的。在这样的计算机上，任何程序的第一个字节是0。这种机器上书写的许多程序包含一个陷阱：它们把空指针用作一个串。

例如：

```
#define NULL 0

char *string;

string = NULL;
cout << "String is '" << str "'\n";
```

实际上，这是非法使用**string**。空指针永远不能被逆引用。在VAX机器，这个错误不会有什么问题。因为0字节的程序就是0，**string**指向一个空串。这是凭运气，而不是故意设计的。

在VAX机上，代码将输出：

```
String is ..
```

在原来的Celerity机上，程序的第一个字节是“Q”。当在Celerity上运行该程序时，它将输出：

```
String is 'Q'
```

在其它的计算机上，这类代码可能会得出意外的结果。从VAX机移植到Celerity机的许多实用程序都存在“Q”陷阱。

文件名问题

UNIX规定文件名为/root/sub/file格式，而MS-DOS/Windows使用\root\sub\file格式。当从UNIX向MS-DOS/Windows移植时，必须修改文件名。例如：

```
#ifndef __MSDOS__
#include <sys/stat.h> /* UNIX version of the file */
#else __MSDOS__
#include <sys\stat.h> /* DOS version of the file */
#endif __MSDOS__
```

问题25.1： 示例25.1为什么能在UNIX上运行，而在MS-DOS/Windows下运行时我们得到下面的信息：

```
oot ew    able: file not found
```

示例25.1 *ifstream in_file*

```
#ifndef __MSDOS__
#define NAME "/root/new/table"
#else __MSDOS__
#define NAME "\\root\\new\\table"
#endif __MSDOS__

in_file.open(NAME);
if (in_file.bad()) {
    cout << NAME << ": file not found\n";
    exit(8);
}
```

文件类型

在UNIX上，只有一种类型的文件。在MS-DOS/Windows上，有两种类型的文件：文本和二进制。在MS-DOS/Windows中，用O_BINARY和O_TEXT标志来指明文件类型。老版本的UNIX没有这样的标志。

处理这个问题的一种办法是，为每种系统书写不同的open调用语句：

```
#ifndef __MSDOS__
file_descriptor = open("file", O_RDONLY);
#else __MSDOS__
file_descriptor = open("file", O_RDONLY|O_BINARY);
#include <sys\stat.h> /* DOS version of the file */
#endif __MSDOS__
```

这有点混乱。更好的方法是定义哑元O_BINARY和O_TEXT标志：

```
#ifndef O_BINARY /* Do we have an O_BINARY? */
#define O_BINARY 0 /* If not define one (BINARY and TEXT) */
#define O_TEXT 0 /* so they don't get in the way */
#endif O_BINARY

...
file_descriptor = open("file", O_RDONLY|O_BINARY);
```

小 结

在C++中可能要写可移植的程序。因为C++运行在使用不同操作系统的许多不同类型的机器上，所以这很不容易。但是，如果你在编制代码时始终注意到移植这个问题，那么你遇到的问题可能最少。

移植四字节字

移植问题也存在于编程以外的领域中。当这本书被翻译为日语时，它的翻译者在编程问题上会遇到一点小麻烦。

题目是：“写一个程序，把下流话（四字母字）换为更文雅的字。”

麻烦的是，日语没有四字母字这个概念。在他们的语言里，所有的字都只有一个字母。翻译者解释概念时，可以尽最大可能地解释清楚，但是要让读者弄清这个问题时，他就必须先解释英语中所忌讳使用的字。这里我省掉了相应的例子。

答 疑

问题25.1：问题出在C++使用反斜杠（\）作为消除字符。字符\r是回车键，\n是换行符，而\t是水平制表符。真正的名字为：

```
<return>oot<new line>ew<tab>able
```

名字应该写为:

```
#define NAME "\\root\\new\\table"
```

注意

`#include`使用文件名, 而不是C++串。你在一个`#include`中可以使用一个反斜杠, 而在C++串中就必须使用双反斜杠 (`\\`)。下面两行代码均是正确的:

```
#define NAME "\\root\\new\\table"
```

```
#include "\\root\\new\\defs.h"
```



本章内容:

- 需求
- 代码设计
- 编码
- 功能描述
- 测试
- 修改
- 最后的警告
- 程序文件
- 编程练习

第26章

组合在一起

本章中，我们创建一个完整的程序，从提出需求到测试结果，涉及到编程的每一步。

需求

在开始之前，我们必须决定要干什么。这是非常重要的一步，并且可以缩短许多编程周期。

本章的程序必须满足若干要求。首先，它必须足够长，从而能够说明模块化程序设计。但同时，又必须足够短，能放在一章的内容里。第二，它必须足够复杂，以说明C++的众多特性，但又必须足够简单，让C++编程的入门者能够明白。

最后，它必须是有用的。定义是否有用很不简单。对一个人有用的程序可能对另一个人就没有用。我们精炼该要求，把它表述为“对C++程序员来说必须是有用的。”我们选择的这个程序读取C++源文件，并统计出嵌套括号的层数及代码行中注释行的比率。

统计程序的规格说明为：

```
Preliminary Specification for a C++ Statistics Gathering Program
Steve Oualline
February 10, 1995
```

程序stat采集C++源文件的统计数据，并输出出来。命令行为：

```
stat <files..>
```

这里，<files..>是源文件列表。下面是程序作用于一个小测试文件时的结果。

```
1 {0 {0 #include <iostream.h>
2 {0 {0 /*****
3 {0 {0 * calc -- a simple 4-function calculator *
4 {0 {0 *****/
```



```

5 (0 (0 int result; // The result of the calculations
6 (0 (0 char oper_char; // User-specified operator
7 (0 (0 int value; // Value specified after the operator
8 (0 (0 main()
9 (0 (1 {
10 (0 (1 result = 0; // Initialize the result
11 (0 (1
12 (0 (1 // Le (1) {
14 (0 (2 cout << "Result: " << result << '\n';
15 (0 (2 cout << "Enter operator and number: ";
16 (0 (2
17 (0 (2 cin >> oper_char >> value;
18 (0 (2
19 (1 (2 if ((oper_char == 'q') ||
20 (0 (2 (oper_char == 'Q'))
21 (0 (2 break;
22 (0 (2
23 (0 (3 switch (oper_char) {
24 (0 (3 case '+':
25 (0 (3 result += value;
26 (0 (3 break;
27 (0 (3 case '-':
28 (0 (3 result -= value;
29 (0 (3 break;
30 (0 (3 case '*':
31 (0 (3 result *= value;
32 (0 (3 break;
33 (0 (3 case '/':
34 (0 (4 if (value == 0) {
35 (0 (4 cout << "Error: Divide by zero\n";
36 (0 (4 cout << " operation ignored\n";
37 (0 (3 } else
38 (0 (3 result /= value;
39 (0 (3 break;
40 (0 (3 default:
41 (0 (3 cout << "Unknown operator " <<
42 (0 (3 oper_char << '\n';
43 (0 (3 break;
44 (0 (2 )
45 (0 (1 }
46 (0 (1 return (0);
47 (0 (0 )

```

Total number of lines: 47

Maximum nesting of () : 2

Maximum nesting of { } : 4

```

Number of blank lines .....4
Number of comment only lines .....4
Number of code only lines .....35
Number of lines with code and comments 4
Comment to code ratio 20.5%

```

代码设计

有几种代码设计流派。在结构化设计中，你把代码划分为模块，然后再把模块划分为子模块，再把子模块划分子子模块，依此类推。这也称作面向过程的程序设计。在面向对象的程序设计(*object-oriented programming*)中，你要试着把问题看作是成员函数要操纵的数据集合。

还有另外其它的方法，如状态表和转换图。所有这些方法基本核心都是一样的：“用尽可能最清楚最简单的方式整理程序的信息，然后再把它们转为C++代码。”

我们的程序分为几个逻辑模块。第一，是记号扫描模块，它读入原始的C++代码，并把它们转为记号。实际上，该函数被再划分为两个更小的模块。第一个模块读入输入流，并判定字符是什么类型的。第二个模块获取字符类型信息，并据此转为汇编记号。另一个模块包含采集的统计结果和一个主程序。

记号模块

我们的程序扫描C++源代码，并用记号产生统计结果。一个记号是组成一个单字、数或符号的一组字符。例如，代码行：

```
answer = (123 + 456) / 89; // Compute some sort of result
```

中包含下列记号：

T_ID	The word "answer"
T_OPERATOR	The character "="
T_L_PAREN	Left parenthesis
T_NUMBER	The number 123
T_OPERATOR	The character "+"
T_NUMBER	The number 456
T_R_PAREN	Right parenthesis
T_OPERATOR	The divide operator
T_NUMBER	The number 89
T_OPERATOR	The semicolon
T_COMMENT	The // comment
T_NEW_LINE	The end-of-line character

记号模块必须能识别出字符组。例如，一个标识符定义为一个字母或是下划线后跟任

何个字母或数字。这样记号识别程序要包含下面的伪代码：

```
If the current character is a letter then  
    scan until we get a character that's not a letter or digit
```

你从伪代码中可以看出，识别程序要使用大量的字符类型，所以我们需要一个模块帮助我们区分类型信息。

字符类型模块

字符类型模块的目的是读入字符，并判定它的类型。有些类型是重叠的。例如，C_ALPHA_NUMERIC包含了C_NUMERIC字符集。该模块在一个数组中存储了大部分的类型信息，在处理像C_ALPHA_NUMERIC这样的特殊类型时，再加上一些逻辑判定。

统计类

在这个程序中，一个统计是一个对象，它扫过记号并输出统计结果。我们从为其定义抽象类开始。该类是我们将采集的统计结果的基础。类的图示如图26.1所示。

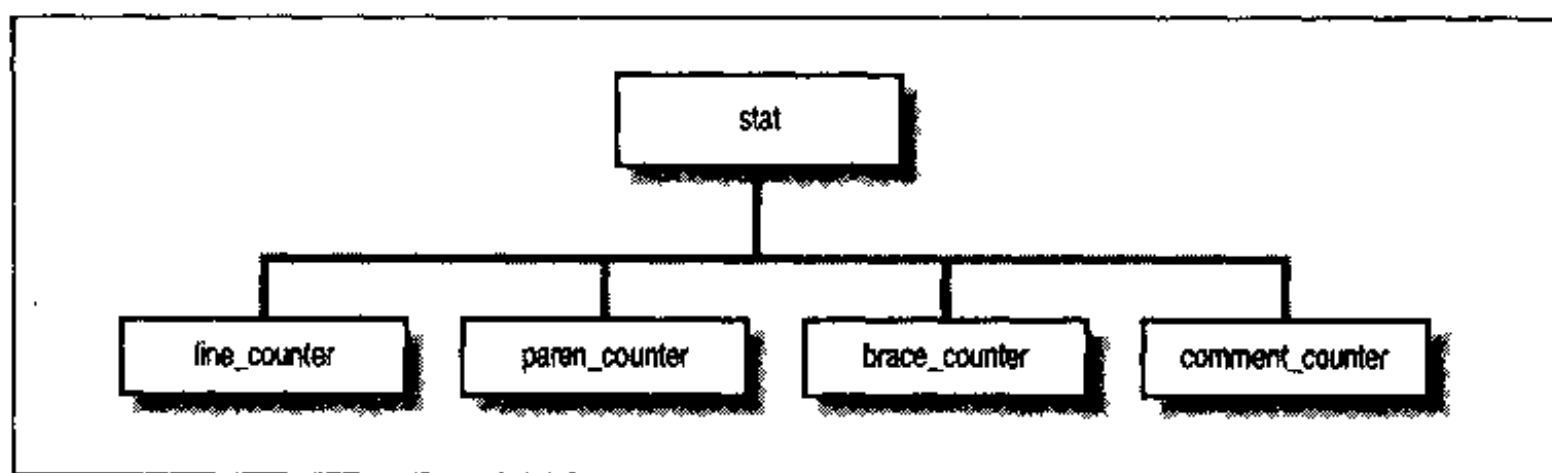


图 26.1 统计类层

统计的定义是：“使用记号来集中统计结果。”这些统计结果可以在每行的开头或是文件尾输出。

四个统计更具体一些。例如，类paren_counter统计括号的嵌套数及最大嵌套数。当前嵌套的括号打印在每行的开头（“(”数）。最大嵌套层数输出在文件尾。

其它类的定义与此类似。这里使用的唯一技巧是，我们让行号也作为一个统计结果。它统计T_NEW_LINE记号的数目，并在每行的开头输出出来。

编 码

编码过程相当简单。遇到的唯一问题是正确读入一整行。

功能描述

本节描述程序中的所有类和主要函数。如想更完整详细地了解这些情况，请参考本章最后列出的程序代码。

char_type类

char_type类设置字符类型。它的大部分工作通过一个type_info表来完成。有些类型，如C_ALPHA_NUMERIC，包括两个不同字符类型：C_ALPHA和C_DIGIT。所以，除了表之外，还需要一些少量的代码来处理特殊的情况。

input_file类

该类从输入文件中读入数据，一次读入一个字符。它缓存一整行内容，并根据控制输出该行。

token类

我们想得到记号的输入流。我们有一个由字符组成的输入流。该类的主要函数next_token把字符转换为记号。实际上，字符识别模块相当简单，因为我们不必处理一个完备的C++记号程序必须处理的大部分细节。

该函数的代码相当简单，但有一件事有点复杂：它要把多个注释行分解为一系列的T_COMMENT和T_NEW_LINE记号。

这一部分我们使用了个小技巧。TOKEN_LIST宏用来产生记号类型的枚举表及一个包含每个记号名的串数组。让我们详细地看看它是如何做的。

TOKEN_LIST类的定义如下：

```
#define TOKEN_LIST \
    T(T_NUMBER),      /* Simple number (floating point or integer) */ \
    T(T_STRING),      /* String or character constant */ \
    T(T_COMMENT),     /* Comment */ \
    T(T_NEWLINE),     /* Newline character */ \
    T(T_OPERATOR),    /* Arithmetic operator */ \
    T(T_L_PAREN),     /* Character "(" */ \
    T(T_R_PAREN),     /* Character ")" */ \
    T(T_L_CURLY),     /* Character "{" */ \
    T(T_R_CURLY),     /* Character "}" */ \
    T(T_ID),          /* Identifier */ \
    T(T_EOF)          /* End of File */
```

当运行时，该宏生成下面的代码：

```
T(T_NUMBER),
T(T_STRING),
// .. and so on
```

如果我们定义一个T宏，则当展开TOKEN_LIST宏时，它将被展开。我们愿意使用TOKEN_LIST宏来产生名字列表，所以我们这样定义T宏：

```
#define T(x) x // Defien T() as the name
```

现在，TOKEN_LIST宏将生成：

```
T_NUMBER,
T_STRING,
// .. and so on
```

把这些放到一起，得到一个生成TOKEN_TYPE枚举列表的方法：

```
#define T(x) x // Define T() as the name
enum TOKEN_TYPE {
    TOKEN_LIST
};
#undef T // Remove old temporary macro
```

然后，我们这样定义T宏，让其生成一个串：

```
#define T(x) #x // Define x as a string
```

这允许我们用TOKEN_LIST来生成一个包含记号名的串列表：

```
#define T(x) #x // Define T() as the name
const char *const TOKEN_NAMES[] = {
    TOKEN_LIST
};
#undef T // Remove old temporary macro
```

当我们展开时，该宏产生：

```
const char *const TOKEN_NAMES[] = {
    "T_NUMBER",
    "T_STRING",
    // ....
```

在有限的几种情况下，可以使用这样的技巧。但是，这样的技巧应该配有大量的注释，这样修改程序的维护程序员才可能读懂你的代码。

stat类

stat类是一个抽象类，它是我们采集的四个真正统计的基础。它的开始是读记号的成员



函数。该函数是个纯虚函数，这意味着任何派生类必须定义take_token函数。

```
class stat {
public:
    virtual void take_token(TOKEN_TYPE token) = 0;
```

take_token函数从记号得出统计结果。我们要有一些办法来输出结果。我们在两个地方输出统计结果。第一个地方是每行的开头，第二个在文件尾。我们的抽象类包含两个虚函数，分别处理两种情况：

```
    virtual void line_start(void) {};
    virtual void eof(void) {};
};
```

和take_token不一样，这些函数的函数体都是缺省的空，但是函数体是一样的。这是什么意思？我们的派生类必须定义take_token。它们不必定义line_start或是eof函数。

line_counter类

我们采集的最简单的统计是目前已处理的行数。这个统计由line_counter类来完成。它只注意一个记号，即：T_NEW_LINE。在每行的开头，它输出行号（T_NEW_LINE记号的当前值）。在文件尾，该类不输出信息。事实上，line_counter类甚至没有定义eof函数。相反，我们让基类（stat）中定义的缺省函数来完成相应的任务。

brace_counter类

该类跟踪花括号{}的嵌套层数。该类中的take_token成员函数读入记号流。该函数只记录左右花括号数，略过其它符号。

```
// Consume tokens, count the nesting of {}
void brace_counter::take_token(TOKEN_TYPE token) {
    switch (token) {
        case T_L_CURLY:
            ++cur_level;
            if (cur_level > max_level)
                max_level = cur_level;
            break;
        case T_R_CURLY:
            --cur_level;
            break;
        default:
            // Ignore
            break;
    }
}
```


在两个地方输出该统计结果。第一个是在每行的开始。第二个是在文件尾。我们定义两个成员函数来输出这些统计结果：

```
// Output start of line statistics
// namely the current line number
void brace_counter::line_start(void) {
    cout.setf(ios::left);
    cout.width(2);

    cout << '(' << cur_level << ' ';

    cout.unsetf(ios::left);
    cout.width();
}

// Output eof statistics
// namely the total number of lines
void brace_counter::eof(void) {
    cout << "Maximum nesting of {} : " << max_level << '\n';
}
```

paren_counter类

该类非常类似于brace_counter类。事实上，它就是先拷贝brace_counter类，然后再做一些小的修改而得到的。

我们可以把paren_counter类和brace_counter类合并为一个类，使用一个参数可以告诉它遇到的是什么字符。好，这是下一版本要做的事情。

comment_counter类

在这个类中，我们记录下注释行、代码行、既有注释又有代码的行数及空行。结果输出在文件尾。

do_file过程

do_file过程读入文件，一次读入一个记号，并把它们送到每个统计类的take_token程序。但是它如何知道要使用哪个统计类？见下面的代码：

```
static line_counter line_count;           // Counter of lines
static paren_counter paren_count;         // Counter of () levels
static brace_counter brace_count;         // Counter of {} levels
static comment_counter comment_count;     // Counter of comment info
```

```

// A list of the statistics we are collecting
static stat *stat_list[] = {
    &line_count,
    &paren_count,
    &brace_count,
    &comment_count,
    NULL
};

```

这段代码中要注意两件事。虽然line_count、paren_count、brace_count和comment_count是完全不同的类型，但它们都是基于stat类型的。这意味着，我们可以把它们放到一个叫stat_list的数组中。这样的设计可以容易地增加另外的统计类。所要做的就是定义一个新类，并把新输入项放到stat_list中。

测试

我们使用一个小的包含各种可能遇到的不同类型符号的C++程序来测试这个程序。结果显示在示例26.2中。

示例26.2 stat/test.cc

```

/*****
 * This is a mult-line comment
 *      T_COMMENT, T_NEWLINE
 *****/
const int LINE_MAX = 500;      // T_ID, T_OPERATOR, T_NUMBER

// T_L_PAREN, T_R_PAREN
static void do_file( const char *const name)
{
    // T_L_CURLY
    char *name = "Test"        // T_STRING

    // T_R_CURLY
}
// T_EOF

```

修改

正如程序表现出来的，它只统计了很有限的几种情况。增加平均标识符长度、每个过程的统计情况和每个类的统计情况等这样一些数据可能会更好。设计程序时，牢记在心的应该是扩充。

我们只有四种类型的统计结果，因为我们已经完成了证明C++先进特性之使命。我们没

有做得更多，因为这会使程序太复杂，作为一章的内容太多了。总之，程序运行得很好。

最后的警告

不要因为你可以得到统计结果，就认为它是有用的程序。

程序文件

ch_type.h文件

示例26.3 stat/ch_type.h

```
/* **** */
* char_type -- Character type class *
* *
* Member functions: *
*     type -- returns the type of a character. *
*             (Limited to simple types) *
*     is(ch, char_type) -- check to see if ch is *
*             a member of the given type. *
*             (Works for derived types as well.) *
* **** */
class char_type {
public:
    enum CHAR_TYPE {
        C_EOF,                // End of file character
        C_WHITE,              // Whitespace or control character
        C_NEWLINE,            // A newline character
        C_ALPHA,              // A letter (includes _)
        C_DIGIT,              // A number
        C_OPERATOR,           // Random operator
        C_SLASH,              // The character '/'
        C_L_PAREN,            // The character '('
        C_R_PAREN,            // The character ')'
        C_L_CURLY,            // The character '{'
        C_R_CURLY,            // The character '}'
        C_SINGLE,             // The character '\''
        C_DOUBLE,             // The character '"'
        // End of simple types, more complex, derived types follow
        C_HEX_DIGIT,          // Hexidecimal digit
        C_ALPHA_NUMERIC       // Alpha numeric
    };
private:
```

```

        static enum CHAR_TYPE type_

        // Fill in a range of type info stuff
        void fill_range(int start, int end, CHAR_TYPE type);
    public:
        char_type();    // Initialize the data
        // ~char_type    -- default destructor

        // Returns true if character is a given type
        int is(int ch, CHAR_TYPE kind);

        CHAR_TYPE type(int ch);
};

```

ch_type.cc文件

示例26.4 stat/ch_type.cc

```

/*****
 * ch_type package
 *
 * The class ch_type is used to tell the type of
 * various characters.
 *
 * The main member functions are:
 *     is -- True if the character is the indicated
 *           type.
 *     type -- Return type of character.
 *****/
#include <iostream.h>

#include "ch_type.h"

// Define the type information array
char_type::CHAR_TYPE char_type::type_info[256];
/*****
 * fill_range -- fill in a range of types for the
 *               character type class
 *
 * Parameters
 *     start, end -- range of items to fill in
 *     type -- type to use for filling
 *****/
void char_type::fill_range(int start, int end, CHAR_TYPE type)
{

```

```

    int cur_ch;

    for (cur_ch = start; cur_ch <= end; ++cur_ch) {
        type_info[cur_ch] = type;
    }
}

/*****
 * char_type::char_type -- initialize the char type table*
 *****/
char_type::char_type()
{
    fill_range(0, 255, C_WHITE);

    fill_range('A', 'Z', C_ALPHA);
    fill_range('a', 'z', C_ALPHA);
    type_info['_'] = C_ALPHA;

    fill_range('0', '9', C_DIGIT);

    type_info['!'] = C_OPERATOR;
    type_info['#'] = C_OPERATOR;
    type_info['$'] = C_OPERATOR;
    type_info['%'] = C_OPERATOR;
    type_info['^'] = C_OPERATOR;
    type_info['&'] = C_OPERATOR;
    type_info['*'] = C_OPERATOR;
    type_info['~'] = C_OPERATOR;
    type_info['+'] = C_OPERATOR;
    type_info['='] = C_OPERATOR;
    type_info['|'] = C_OPERATOR;
    type_info['~'] = C_OPERATOR;
    type_info[','] = C_OPERATOR;
    type_info[':'] = C_OPERATOR;
    type_info['?'] = C_OPERATOR;
    type_info['.'] = C_OPERATOR;
    type_info['<'] = C_OPERATOR;
    type_info['>'] = C_OPERATOR;

    type_info['/'] = C_SLASH;
    type_info['\n'] = C_NEWLINE;

    type_info['('] = C_L_PAREN;
    type_info[')'] = C_R_PAREN;

```

```

    type_info['{'] = C_L_CURLY;
    type_info['}'] = C_R_CURLY;

    type_info['"'] = C_DOUBLE;
    type_info['\'''] = C_SINGLE;
}

int char_type::is(int ch, CHAR_TYPE kind)
{
    if (ch == EOF) return (kind == C_EOF);

    switch (kind) {
        case C_HEX_DIGIT:
            if (type_info[ch] == C_DIGIT)
                return (1);
            if ((ch >= 'A') && (ch <= 'F'))
                return (1);
            if ((ch >= 'a') && (ch <= 'f'))
                return (1);
            return (0);
        case C_ALPHA_NUMERIC:
            return ((type_info[ch] == C_ALPHA) ||
                    (type_info[ch] == C_DIGIT));
        default:
            return (type_info[ch] == kind);
    }
};

char_type::CHAR_TYPE char_type::type(int ch) {
    if (ch == EOF) return (C_EOF);
    return (type_info[ch]);
}

```

token.h文件

示例26.5 stat/token.h

```

/*****
 * token -- token handling module
 *
 * Functions:
 *     next_token -- get the next token from the input
 *****/

/*

```

```

* A list of tokens
*     Note, how this list is used depends on defining the macro T.
*     This macro is used for defining the tokens types themselves
*     as well as the string version of the tokens.
*/
#define TOKEN_LIST \
    T(T_NUMBER),          /* Simple number (floating point or integer) */ \
    T(T_STRING),          /* String-or character constant */ \
    T(T_COMMENT),         /* Comment */ \
    T(T_NEWLINE),         /* Newline character */ \
    T(T_OPERATOR),        /* Arithmetic operator */ \
    T(T_L_PAREN),         /* Character "(" */ \
    T(T_R_PAREN),         /* Character ")" */ \
    T(T_L_CURLY),         /* Character "{" */ \
    T(T_R_CURLY),         /* Character "}" */ \
    T(T_ID),              /* Identifier */ \
    T(T_EOF)              /* End of File */

/*
* Define the enumerated list of tokens.
*     This makes use of a trick using the T macro
*     and our TOKEN_LIST
*/
#define T(x) x            // Define T() as the name
enum TOKEN_TYPE {
    TOKEN_LIST
};
#undef T                  // Remove old temporary macro

// A list of the names of the tokens
extern const char *const TOKEN_NAMES[];

const int LINE_MAX = 500;    // Longest possible line

/*****
* input_file -- data from the input file
*
* The current two characters are store in
*     cur_char and next_char
*
* The member function read_char moves everyone up
* one character.
*
* The line is buffered and output everytime a newline
* is passed.
*****/

```

```

*****/
class input_file: public ifstream {
private:
    char line[LINE_MAX];    // Current line
    char *char_ptr;         // Current character on the line
public:
    int cur_char;    // Current character (can be EOF)
    int next_char;   // Next character (can be EOF)

    /*
     * Initialize the input file and read the first 2
     * characters.
     */
    input_file(const char *const name) : ifstream(name) {
        if (bad())
            return;
        cur_char = get();
        next_char = get();
        char_ptr = line;
    }

    /*
     * Write the line to the screen
     */
    void flush_line() {
        *char_ptr = '\0';
        cout << line;
        cout.flush();
        char_ptr = line;
    }

    /*
     * Advance one character
     */
    void read_char(void) {
        *char_ptr = cur_char;
        ++char_ptr;

        cur_char = next_char;
        next_char = get();
    }
};

#ifndef TRUE
#define TRUE 1           // Define a simple TRUE/FALSE value
#define FALSE 0

```



```

#endif /* TRUE */

/*****
 * token class
 *
 * Reads the next token in the input stream
 * and returns its type.
 *****/
class token {
private:
    // True if we are in the middle of a comment
    int in_comment;

    // True if we need to read a character
    // (This hack is designed to get the new lines right)
    int need_to_read_one;

    // Read a /* */ style comment
    TOKEN_TYPE read_comment(input_file &in_file);
public:
    token(void) {
        in_comment = FALSE;
        need_to_read_one = 0;
    }

    // Return the next token in the stream
    TOKEN_TYPE next_token(input_file &in_file);
};

```

token.cc文件

示例26.6 stat/token.cc

```

/*****
 * token -- token handling module
 *
 * Functions:
 * next_token -- get the next token from the input *
 *****/
#include <fstream.h>
#include <stdlib.h>

#include "ch_type.h"
#include "token.h"

```

```

/*
 * Define the token name list
 *     This makes use of a trick using the T macro
 *     and our TOKEN_LIST
 */
#define T(x) #x          // Define x as a string
const char *const TOKEN_NAMES[] = {
    TOKEN_LIST
};
#undef T                  // Remove old temporary macro

static char_type char_type;    // Character type information
/*****
 * read_comment -- read in a comment
 *
 * Parameters
 *     in_file -- file to read
 *
 * Returns
 *     Token read. Can be a T_COMMENT or T_NEW_LINE
 *     depending on what we read.
 *
 * Multi-line comments are split into multiple
 * tokens.
 *****/
TOKEN_TYPE token::read_comment(input_file &in_file)
{
    if (in_file.cur_char == '\n') {
        in_file.read_char();
        return (T_NEWLINE);
    }
    while (1) {
        in_comment = TRUE;
        if (in_file.cur_char == EOF) {
            cerr << "Error: EOF inside comment\n";
            return (T_EOF);
        }
        if (in_file.cur_char == '\n')
            return (T_COMMENT);
        if ((in_file.cur_char == '*') &&
            (in_file.next_char == '/')) {
            in_comment = FALSE;
            // Skip past the ending */
            in_file.read_char();
            in_file.read_char();

```



```

        return (T_COMMENT);
    }
    in_file.read_char();
}
}

/*****
 * next_token -- read the next token in an input stream *
 *
 * Parameters
 *   in_file -- file to read
 *
 * Returns
 *   next token
 *****/
TOKEN_TYPE token::next_token(input_file &in_file)
{
    if (need_to_read_one)
        in_file.read_char();

    need_to_read_one = 0;

    if (in_comment)
        return (read_comment(in_file));

    while (char_type.is(in_file.cur_char, char_type::C_WHITE)) {
        in_file.read_char();
    }
    if (in_file.cur_char == EOF)
        return (T_EOF);

    switch (char_type.type(in_file.cur_char)) {
        case char_type::C_NEWLINE:
            in_file.read_char();
            return (T_NEWLINE);
        case char_type::C_ALPHA:
            while (char_type.is(in_file.cur_char,
                                char_type::C_ALPHA_NUMERIC))
                in_file.read_char();
            return (T_ID);
        case char_type::C_DIGIT:
            in_file.read_char();
            if ((in_file.cur_char == 'X') || (in_file.cur_char == 'x')) {
                in_file.read_char();
                while (char_type.is(in_file.cur_char, char_type::C_HEX_

```

```

        in_file.read_char();
        return (T_NUMBER);
    }
    while (char_type.is(in_file.cur_char, char_type::C_DIGIT))
        in_file.read_char();
    return (T_NUMBER);
case char_type::C_SLASH:
    // Check for /* characters
    if (in_file.next_char == '*') {
        return (read_comment(in_file));
    }
    // Now check for double slash comments
    if (in_file.next_char == '/') {
        while (1) {
            // Comment starting with // and ending with EOF is legal
            if (in_file.cur_char == EOF)
                return (T_COMMENT);
            if (in_file.cur_char == '\n')
                return (T_COMMENT);
            in_file.read_char();
        }
    }
    // Fall through
case char_type::C_OPERATOR:
    in_file.read_char();
    return (T_OPERATOR);
case char_type::C_L_PAREN:
    in_file.read_char();
    return (T_L_PAREN);
case char_type::C_R_PAREN:
    in_file.read_char();
    return (T_R_PAREN);
case char_type::C_L_CURLY:
    in_file.read_char();
    return (T_L_CURLY);
case char_type::C_R_CURLY:
    in_file.read_char();
    return (T_R_CURLY);
case char_type::C_DOUBLE:
    while (1) {
        in_file.read_char();
        // Check for end of string
        if (in_file.cur_char == '"')
            break;
    }

```

```

        // Escape character, then skip the next character
        if (in_file.cur_char == '\\')
            in_file.read_char();
    }
    in_file.read_char();
    return (T_STRING);
case char_type::C_SINGLE:
    while (1) {
        in_file.read_char();
        // Check for end of character
        if (in_file.cur_char == '\\')
            break;

        // Escape character, then skip the next character
        if (in_file.cur_char == '\\')
            in_file.read_char();
    }
    in_file.read_char();
    return (T_STRING);
default:
    cerr << "Internal error: Very strange character\n";
    abort();
}
cerr << "Internal error: We should never get here\n";
abort();
return (T_EOF);    // Should never get here either
}

```

stat.cc文件

示例26.7 stat/stat.cc

```

/*****
 * stat
 *
 * Produce statistics about a program
 *
 * Usage:
 * stat [options] <file-list>
 *
 *****/
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <memory.h>

```

```

#include "ch_type.h"
#include "token.h"

/*****
 * stat -- general purpose statistic
 *
 * Member functions:
 *   take_token -- receives token and uses it to
 *                  compute statistic
 *   line_start -- output stat at the beginning of
 *                  a line
 *   eof -- output stat at the end of the file
 *****/
class stat {
public:
    virtual void take_token(TOKEN_TYPE token) = 0;
    virtual void line_start(void) {};
    virtual void eof(void) {};
    // Default constructor
    // Default destructor
    // Copy constructor defaults as well (probably not used)
};

/*****
 * line_counter -- handle line number / line count
 *                  stat.
 *
 * Counts the number of T_NEW_LINE tokens seen and
 * outputs the current line number at the beginning
 * of the line.
 *
 * At EOF it will output the total number of lines
 *****/
class line_counter: public stat {
private:
    int cur_line; // Line number for the current line
public:
    // Initialize the line counter -- to zero
    line_counter(void) {
        cur_line = 0;
    };
    // Default destructor
    // Default copy constructor (probably never called)
};

```

```

// Consume tokens, count the number of new line tokens
void take_token(TOKEN_TYPE token) {
    if (token == T_NEWLINE)
        ++cur_line;
}

// Output start of line statistics
// namely the current line number
void line_start(void) {
    cout << setw(4) << cur_line << ' ' << setw(0);
}

// Output eof statistics
// namely the total number of lines
void eof(void) {
    cout << "Total number of lines: " << cur_line << '\n';
}
};

/*****
 * paren_count -- count the nesting level of ()
 *
 * Counts the number of T_L_PAREN vs T_R_PAREN tokens
 * and writes the current nesting level at the beginning
 * of each line.
 *
 * Also keeps track of the maximum nesting level.
 *****/
class paren_counter: public stat {
private:
    int cur_level;          // Current nesting level
    int max_level;          // Maximum nesting level
public:
    // Initialize the counter
    paren_counter(void) {
        cur_level = 0;
        max_level = 0;
    };
    // Default destructor
    // Default copy constructor (probably never called)

    // Consume tokens, count the nesting of ()
    void take_token(TOKEN_TYPE token) {
        switch (token) {
            case T_L_PAREN:

```

```

        ++cur_level;
        if (cur_level > max_level)
            max_level = cur_level;
        break;
    case T_R_PAREN:
        --cur_level;
        break;
    default:
        // Ignore
        break;
    }
}

// Output start of line statistics
// namely the current line number
void line_start(void) {
    cout.setf(ios::left);
    cout.width(2);

    cout << '(' << cur_level << ' ';

    cout.unsetf(ios::left);
    cout.width();
}

// Output eof statistics
// namely the total number of lines
void eof(void) {
    cout << "Maximum nesting of () : " << max_level << '\n';
}

};

/*****
 * brace_counter -- count the nesting level of {}      *
 *                                                       *
 * Counts the number of T_L_CURLY vs T_R_CURLY tokens  *
 * and writes the current nesting level at the beginning *
 * of each line.                                       *
 *                                                       *
 * Also keeps track of the maximum nesting level.      *
 *                                                       *
 * Note: brace_counter and paren_counter should        *
 * probably be combined.                               *
 *****/
class brace_counter: public stat {

```



```

private:
    int cur_level;           // Current nesting level
    int max_level;          // Maximum nesting level
public:
    // Initialize the counter
    brace_counter(void) {
        cur_level = 0;
        max_level = 0;
    };
    // Default destructor
    // Default copy constructor (probably never called)

    // Consume tokens, count the nesting of ()
    void take_token(TOKEN_TYPE token) {
        switch (token) {
            case T_L_CURLY:
                ++cur_level;
                if (cur_level > max_level)
                    max_level = cur_level;
                break;
            case T_R_CURLY:
                --cur_level;
                break;
            default:
                // Ignore
                break;
        }
    }

    // Output start of line statistics
    // namely the current line number
    void line_start(void) {
        cout.setf(ios::left);
        cout.width(2);

        cout << '(' << cur_level << ' ';

        cout.unsetf(ios::left);
        cout.width();
    }

    // Output eof statistics
    // namely the total number of lines
    void eof(void) {
        cout << "Maximum nesting of {} : " << max_level << '\n';
    }

```

```

    }
};

/*****
 * comment_counter -- counts the number of lines
 * with and without comments.
 *
 * Outputs nothing at the beginning of each line, but
 * will output a ratio at the end of file
 *
 * Note: This class makes use of two bits:
 * CF_COMMENT -- a comment was seen
 * CF_CODE    -- code was seen
 * to collect statistics.
 *
 * These are combined to form an index into the counter
 * array, so the value of these two bits is very
 * important.
 *****/
static const int CF_COMMENT = (1<<0); // Line contains comment
static const int CF_CODE    = (1<<1); // Line contains code
// These bits are combined to form the statistics
//
// 0          -- [0] Blank line
// CF_COMMENT -- [1] Comment only line
// CF_CODE    -- [2] Code only line
// CF_COMMENT|CF_CODE -- [3] Comments and code on this line

class comment_counter: public stat {
private:
    int counters[4]; // Count of various types of stats
    int flags;       // Flags for the current line
public:
    // Initialize the counters
    comment_counter(void) {
        memset(counters, '\0', sizeof(counters));
        flags = 0;
    };
    // Default destructor
    // Default copy constructor (probably never called)

    // Consume tokens, count the nesting of ()
    void take_token(TOKEN_TYPE token) {
        switch(token) {
            case T_COMMENT:

```



```

        flags |= CF_COMMENT;
        break;
    default:
        flags |= CF_CODE;
        break;
    case T_NEWLINE:
        ++counters[flags];
        flags = 0;
        break;
    }
}

// void line_start(void) -- defaults to base

// Output eof statistics
// namely the total number of lines
void eof(void) {
    cout << "Number of blank lines ....." <<
        counters[0] << '\n';
    cout << "Number of comment only lines ....." <<
        counters[1] << '\n';
    cout << "Number of code only lines ....." <<
        counters[2] << '\n';
    cout << "Number of lines with code and comments " <<
        counters[3] << '\n';
    cout.setf(ios::fixed);
    cout.precision(1);
    cout << "Comment to code ratio " <<
        float(counters[1] + counters[3]) /
        float(counters[2] + counters[3]) * 100.0 << "%\n";
}
};

```

```

static line_counter line_count;        // Counter of lines
static paren_counter paren_count;      // Counter of () levels
static brace_counter brace_count;      // Counter of {} levels
static comment_counter comment_count;  // Counter of comment info

```

```

// A list of the statistics we are collecting
static stat *stat_list[] = {
    &line_count,
    &paren_count,
    &brace_count,
    &comment_count,
}

```

```

        NULL
    };

/*****
 * do_file -- process a single file
 *
 * Parameters
 *     name -- the name of the file to process
 *****/
static void do_file(const char *const name)
{
    input_file in_file(name);    // File to read
    token token;                 // Token reader/parser
    TOKEN_TYPE cur_token;        // Current token type
    class stat **cur_stat;       // Pointer to stat for collection/writing

    if (in_file.bad()) {
        cerr << "Error: Could not open file " << name << " for reading\n";
        return;
    }
    while (1) {
        cur_token = token.next_token(in_file);
        for (cur_stat = stat_list; *cur_stat != NULL; ++cur_stat)
            (*cur_stat)->take_token(cur_token);
#ifdef DEBUG
        cout << "    " << TOKEN_NAMES[cur_token] << '\n';
#endif /* DEBUG */

        switch (cur_token) {
            case T_NEWLINE:
                for (cur_stat = stat_list; *cur_stat != NULL; ++cur_stat)
                    (*cur_stat)->line_start();
                in_file.flush_line();
                break;
            case T_EOF:
                for (cur_stat = stat_list; *cur_stat != NULL; ++cur_stat)
                    (*cur_stat)->eof();
                return;
            default:
                // Do nothing
                break;
        }
    }
}

```

```

main(int argc, char *argv[])
{
    char *prog_name = argv[0]; // Name of the program

    if (argc == 1) {
        cerr << "Usage is " << prog_name << "[options] <file-list>\n";
        exit (8);
    }

    for (/* argc set */; argc > 1; --argc) {
        do_file(argv[1]);
        ++argv;
    }
    return (0);
}

```

用于CC（一般UNIX系统）的UNIX Makefile

示例26.8 *stat/makefile.unx*

```

#
# Makefile for many UNIX compilers using the
# "standard" command name CC
#
CC=CC
CFLAGS=-g
OBJS= stat.o ch_type.o token.o

all: stat.out stat

stat.out: stat
    stat ../calc3/calc3.cc >stat.out

stat: $(OBJS)
    $(CC) $(CFLAGS) -o stat $(OBJS)

stat.o: stat.cc token.h
    $(CC) $(CFLAGS) -c stat.cc

ch_type.o: ch_type.cc ch_type.h
    $(CC) $(CFLAGS) -c ch_type.cc

token.o: token.cc token.h ch_type.h
    $(CC) $(CFLAGS) -c token.cc

```

```
clean:
    rm stat stat.o ch_type.o token.o
```

用于g++的UNIX Makefile

示例26.9 *stat/makefile.gnu*

```
#
# Makefile for the Free Software Foundations g++ compiler
#
CC=g++
CFLAGS=-g -Wall
OBJS= stat.o ch_type.o token.o

all: stat.out stat

stat.out: stat
    stat ../calc3/calc3.cc >stat.out

stat: $(OBJS)
    $(CC) $(CFLAGS) -o stat $(OBJS)

stat.o: stat.cc token.h
    $(CC) $(CFLAGS) -c stat.cc

ch_type.o: ch_type.cc ch_type.h
    $(CC) $(CFLAGS) -c ch_type.cc

token.o: token.cc token.h ch_type.h
    $(CC) $(CFLAGS) -c token.cc

clean:
    rm stat stat.o ch_type.o token.o
```

Turbo C++的UNIX Makefile

示例26.10 *stat/makefile.tcc*

```
#
# Makefile for Borland's Turbo-C++ compiler
#
CC=tcc
#
```

```

# Flags
#      -N  -- Check for stack overflow
#      -v  -- Enable debugging
#      -w  -- Turn on all warnings
#      -ml -- Large model
#
CFLAGS=-N -v -w -ml
OBJS= stat.obj ch_type.obj token.obj

all: stat.out stat.exe

stat.out: stat.exe
        stat ../calc3/calc3.cpp >stat.out

stat.exe: $(OBJS)
        $(CC) $(CFLAGS) -o stat.exe $(OBJS)

stat.obj: stat.cpp token.h
        $(CC) $(CFLAGS) -c stat.cpp

ch_type.obj: ch_type.cpp ch_type.h
        $(CC) $(CFLAGS) -c ch_type.cpp

token.obj: token.cpp token.h ch_type.h
        $(CC) $(CFLAGS) -c token.cpp

clean:
        erase stat.exe stat.obj ch_type.obj token.obj

```

Borland-C++的UNIX Makefile

示例26.11 stat/makefile.bcc

```

#
# Makefile for Borland's Borland-C++ compiler
#
CC=bcc
#
# Flags
#      -N  -- Check for stack overflow
#      -v  -- Enable debugging
#      -w  -- Turn on all warnings
#      -ml -- Large model

```

```

#
CFLAGS=-N -v -w -ml
OBJS= stat.obj ch_type.obj token.obj

all: stat.out stat.exe

stat.out: stat.exe
    stat ../calc3/calc3.cpp >stat.out

stat.exe: $(OBJS)
    $(CC) $(CFLAGS) -estat $(OBJS)

stat.obj: stat.cpp token.h
    $(CC) $(CFLAGS) -c stat.cpp

ch_type.obj: ch_type.cpp ch_type.h
    $(CC) $(CFLAGS) -c ch_type.cpp

token.obj: token.cpp token.h ch_type.h
    $(CC) $(CFLAGS) -c token.cpp

clean:
    erase stat.exe stat.obj ch_type.obj token.obj

```

Microsoft Visual C++的UNIX Makefile

示例26.12 *stat/makefile.msc*

```

#
# Makefile for Microsoft Visual C++
#
CC=cl
#
# Flags
#      AL -- Compile for large model
#      Zi -- Enable debugging
#      Wl -- Turn on warnings
#
CFLAGS=/AL /Zi /Wl
OBJS= stat.obj ch_type.obj token.obj

all: stat.out stat.exe

stat.out: stat.exe

```



```

stat ../calc3/calc3.cpp >stat.out

stat.exe: $(OBSJ)
$(CC) $(CCFLAGS) $(OBSJ)

stat.obj: stat.cpp token.h
$(CC) $(CCFLAGS) -c stat.cpp

ch_type.obj: ch_type.cpp ch_type.h
$(CC) $(CCFLAGS) -c ch_type.cpp

token.obj: token.cpp token.h ch_type.h
$(CC) $(CCFLAGS) -c token.cpp

clean:
erase stat.exe stat.obj ch_type.obj token.obj

```

编程练习

练习26.1: 编写一个程序，检查文本文件中的双字。

练习26.2: 编写一个程序，从一个文件中删除不文雅字，并把它们替换为可以接受的字眼。

练习26.3: 编写一个邮件列表程序。该程序读、写、排序并输出邮件标签。

练习26.4: 修改本章介绍的统计程序，增加一个交叉引用功能。

练习26.5: 编写一个程序，它得到一个文本文件，并把每个长行划分为两个短一些的行。如果可能的话，分隔点应该在句子的结束处，如果句子太长，分隔点在字与字之间。

本章内容:

- 总的看法
- K&R风格的函数
- struct
- malloc和free
- 把结构转化为类
- ssetjmp和longjmp
- 小结
- 编程练习

第27章

从C到C++

总的看法

C++是建筑在原有的C语言之上的, 所以现在仍有许多的C代码。这既是好事又是坏事。说它不好, 是因为这将意味着你可能要处理大量的旧代码。另一方面, 它也总是能为你服务。本章讨论C和C++程序之间的若干不同点, 并讨论如何从一种转换为另一种。

K&R风格的函数

经典的C (因其创始人为Brian Kernighan和Dennis Ritchie, 又称为K&R C) 使用的函数头, 与C++中使用的不一样。在C++中, 函数定义的()内既有参数类型, 又有参数名字。在经典的C中, 只出现参数名字, 到后面再写类型信息:

```
int do_it(char *name, int function)    // C++ function definition
{
    // Body of the function

int do_it(name, function)              // Classic C definition
char *name;
int function;
{
    // Body of the function
```

当C++语言出现时, ANSI C委员会认为, C使用新的函数定义方法是个好主意。但是因为还有使用原来的方式写就的大量代码, 所以C++接受两种类型的函数。

经典C不需要原型。多数情况下, 在C程序中没有原型。没有原型的函数有一个隐含的原型:

```
int funct(...);    // Default prototype for Classic C functions
```


而且，经典C原型没有参数列表。它们只包含“()”，如：

```
int do_it();    // Classic C function prototype
```

它告诉C，do_it返回一个整数，它可以带有任意个参数。C不进行参数的类型检查，所以下面调用do_it的代码是合法的：

```
i = do_it();
i = do_it(1, 2, 3);
i = do_it("Test", 'a');
```

C++需要函数原型，所以你必须写上。有像GNU prototize这样的实用程序帮助你读程序代码，并生成函数原型。否则，你只能手工处理。

struct

在C++中，当你说明一个struct时，你可以把一个结构用作类型名。例如：

```
struct sample {
    int i, j;    // Data for the sample
};
sample sample_var;    // Last sample seen
```

C更严格一些。你必须在每个变量定义之前写上关键字struct：

```
struct sample sample_var;    // Legal in C
sample sample_var;          // illegal in C
```

malloc和free

在C++中，你使用new运算符从堆中得到内存空间，使用delete送回内存。C没有内置的内存处理运算。相反，它使用两个库函数：malloc和free。

函数malloc带有一个参数——要分配的字节数——并返回指向它们的一个指针（如一个char *或是void *）。但是我们怎么能知道一个结构有多大呢？所以就有了sizeof运算符。它返回结构中的字节数。我们可以使用下面的代码分配struct foo类型的一个新变量：

```
foo_var = (struct foo *)malloc(sizeof(struct foo));
```

注意，我们必须使用类型转换，把malloc返回的指针类型转换为有用的类型。在C++语法中，同样的操作更简洁一些：

```
foo_var = new foo;
```

假定我们想分配有三个结构的数组。我们必须让分配字节数乘上3，即：

```
foo_var = (struct foo *)malloc(sizeof(struct foo) * 3);
```

C++的写法为:

```
foo_var = new foo[3];
```

函数`calloc`类似于`malloc`, 只是它带有两个参数: 对象数组中的元素个数和一个元素的大小。使用有三个`foo`的数组例子, 我们有:

```
foo_var = (struct foo *)calloc(3, sizeof(foo));
```

还有一点不同: `calloc`把结构初始化为0。所以C++的写法为:

```
foo_var = new foo[3];  
memset(foo_var, '\0', sizeof(foo) * 3);
```

程序可以自由地混用C风格的`malloc`和C++的`new`调用。但是C的内存分配非常混乱, 所以可能的话应该转换为C++的方式。

C风格的内存分配还有一些陷阱。假定我们以结构`foo`为例, 把它转为一个类。虽然我们使用C的内存函数来为类分配空间, 但不应该使用:

```
class foo (...);  
foo_var = (struct foo *)malloc(sizeof(struct foo)); // Don't code like  
           this
```

因为C++把`struct`看作是类的特殊形式, 所以大多数编译器不会抱怨这行代码。问题是, `malloc`语句只为`foo`分配空间。不调用构造函数, 所以很有可能类没有正确生成。

C使用函数`free`把内存送回堆中。函数`free`带有一个字符指针参数 (所以大多数情况下需要进行类型转换):

```
free((char *)foo_var);  
foo_var = NULL;
```

在C++中, 对简单变量的写法为:

```
delete foo_var;  
foo_var = NULL;
```

而用于数组的写法为:

```
delete []foo_array;  
foo_array = NULL;
```

这里, `foo_array`指向一个数组。

另外, 当把`foo`转换为类时要特别小心。`free`函数只把内存返回堆中。它不调用于`foo`的析构函数。

C风格的内存分配非常混乱，而且有风险。当把它转换为C++时，应尽可能地去掉所有的malloc、calloc和free调用。

警告

根据ANSI C标准草案，由malloc分配的内存必须通过free来释放。同样，由new分配的内存必须由delete来释放。但是，我所见过的大多数编译器实现new时调用malloc，实现delete时调用free。换句话说，new/free或是malloc/free的混合调用通常是能工作的。为了避免出错，你应该遵从规则，避免C和C++运算的混合使用。

把结构转换为类

当我们检查C代码时，常常发现有许多的struct语句，看上去它们应该是class语句。实际上，一个结构就是一个只有数据的类，它的所有成员都是公用的。

C程序员常常利用结构只包含数据这一事实。一个例子就是，对二进制文件的读写是以结构为单位的。例如：

```
a_struct struct_var;    // A structure variable

// Perform a raw read to read in the structure
read_size = read(fd, (char *)&struct_var, sizeof(struct_var));

// Perform a raw write to send the data to a file
write_size = write(fd, (char *)&struct_var, sizeof(struct_var));
```

把这个结构转换为一个类可能会引出问题。C++保留类中的一些额外信息，如虚函数指针。当你使用单纯的写语句把类写向磁盘时，你将输出所有信息。更糟的是，当你读入类时，你会把这个簿记数据覆盖掉。

例如，假定我们有类：

```
class sample {
public:
    const int sample_size;    // Number of samples
    int cur_sample;           // Current sample number
    sample(void) : sample_size(100) {} // Set up class
    virtual void get_sample(); // Routine to get a sample
};
```

：这个类包含三个成员变量：一个常量sample_size（C++不允许你改变它的值），一个简单变量cur_sample和一个指针，它指向调用get_sample时要使用的真正函数。下面的调用将把所有这三个成员都写入磁盘：

```

sample a_sample;
// ...
write_size = write(fd, (char *)&a_sample, sizeof(a_sample));

```

当读入该类时，所有的三个成员都被改变了。这包含常量（不允许改变的量）和函数指针（现在可能指向了令人费解的值）。

C程序员还利用memset函数把结构中的所有成员设置为0。例如：

```

struct a_struct { ... }
a_struct struct_var;
// ...
memset(&struct_var, '\0', sizeof(struct_var));

```

再次强调，把结构转为类时千万要小心。如果我们用类a_sample代替结构struct_var，则常量sample_size将设为0，虚函数指针也设为0。结果，如果要调用get_sample的话，可能会使系统崩溃。

setjmp和longjmp

C通过使用setjmp和longjmp来处理例外。setjmp函数标记程序中的一个位置。longjmp函数跳过被setjmp标记过的地方。

一般地，setjmp返回一个0。这表明程序执行正常的代码。当发生例外时，longjmp调用将返回setjmp函数的位置。程序能够看到的，真正的setjmp调用和longjmp引起的伪setjmp调用之间唯一的区别是，正常的setjmp返回一个0。当longjmp“调用”setjmp时，返回值由longjmp的一个参数来控制。

setjmp函数的定义为：

```

#include <setjmp.h>

int setjmp(jmp_buf env);

```

这里：

env是setjmp保存当前环境信息的地方，这些信息留待以后longjmp来使用。

返回值是0。

正常调用非0值。

返回非0的代码是longjmp调用的结果。

longjmp调用的定义为：

```

void longjmp(jmp_buf env, int return_code);

```

这里：

env是前面setjmp调用所初始化的环境。

return_code是setjmp调用所返回的返回代码。

图27.1说明使用setjmp和longjmp函数时的控制流。

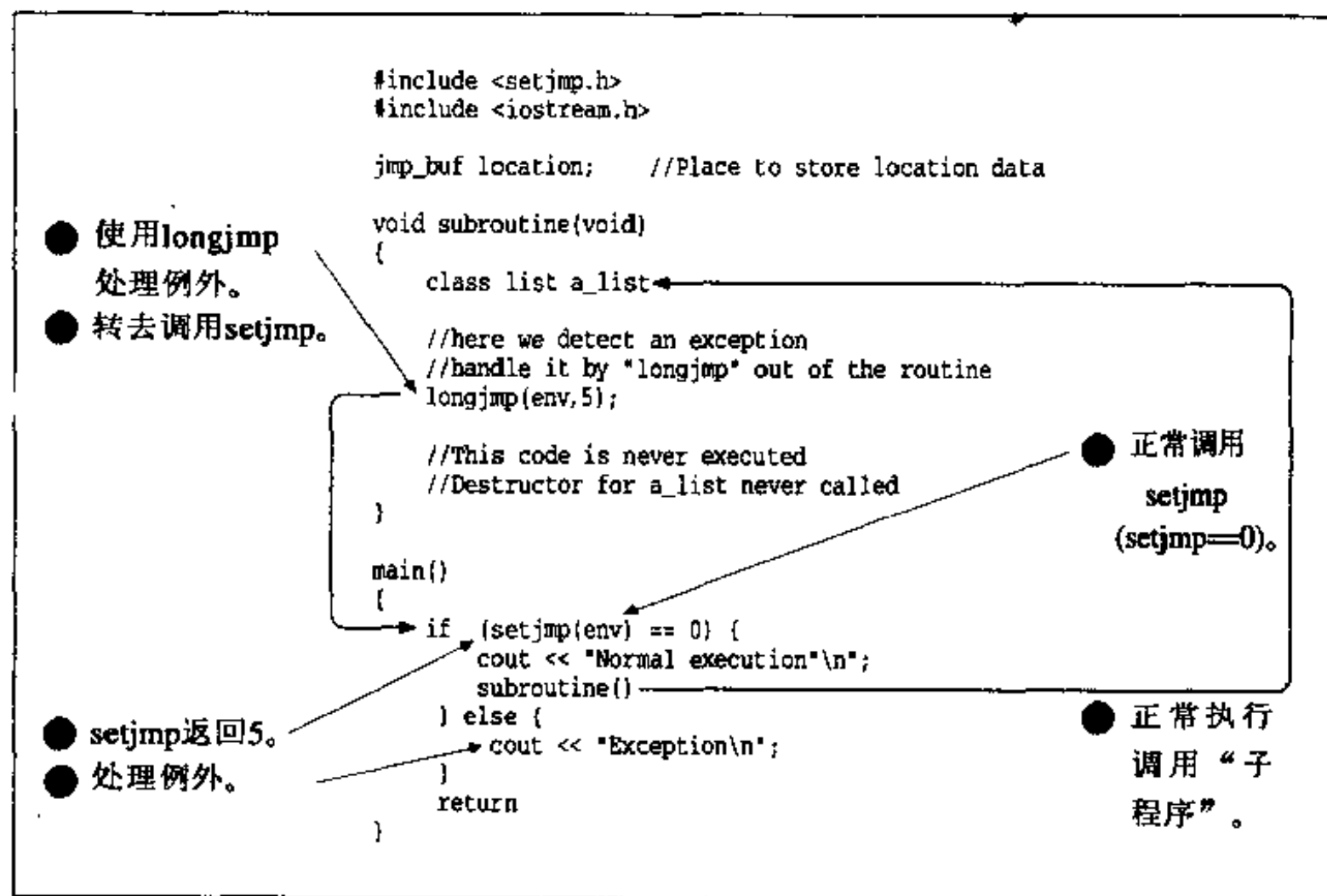


图 27.1 setjmp/longjmp控制流

这里有一个问题。longjmp调用把控制返回给对应的setjmp。它不调用其间要删除的任何类的析构函数。

在图27.1中，我们可以看到，在subroutine中，我们定义了类a_list。一般地，我们在函数尾或是遇到return语句时要调用s_list的析构函数。但是，本例中，我们使用longjmp函数退出函数。因为longjmp是一个不了解类和析构函数的C函数，所以它不调用a_list的析构函数。因此我们遇到了这样一种状况：一个变量已经消失了，但是析构函数却没有调用。技术上称之为“弄乱”。

当把C转为C++时，要把所有的setjmp/longjmp语句改为例外处理。

小 结

用C++编译器编译C程序时，你只需：

1. 把K&R风格的函数头改为标准的C++函数头。
2. 添加函数原型。

3. 把setjmp/longjmp调用改为catch/throw操作。

这两步完成后，你就有了一个C++程序了。它可能运行，但只是一个包装在C++外衣内的C程序。要把它转换为真正的C++程序，还需做下列工作：

4. 把malloc修改为new。

5. 把free修改为delete或delete[]调用。

6. 把printf和scanf调用转为cout和cin。

7. 在把结构说明转为类变量时，要注意使用整个结构或类的read、write和memset函数。

编程练习

练习27.1： 找一个C程序，把它转换为C++程序。



本章内容:

- do/while
- goto
- ?:命令
- 逗号运算符
- ()运算符的重载
- 指向成员的指针
- “吸血鬼”特性
- 答疑

第28章

C++内的“角落”

本章讨论在前面各章中从没有涉及的几个C++特性。本章的名称之所以称为“角落”，是因为在真正的程序中很少用到这些语句。

do/while

do/while语句的语法如下:

```
do {  
    语句  
    语句  
} while (表达式);
```

程序执行循环，然后检查表达式的值，如果表达式为假（0），则停止循环。

注意

该命令至少执行一次。

在C++程序中，do/while语句不常用，因为大多数程序员愿意使用while/break配对语句。

goto

本书中所有的示例程序都没有使用goto语句。实际上，我发现每隔一年，我使用一次goto语句。在极少数情况下，还必须有goto语句，它的语法为:

```
goto 标号;
```

这里，标号是一个语句标号。语句标号与变量名的命名方法一样。给语句加标号的形式如下:

```
标号: 语句
```


例如:

```
for (x = 0; x < X_LIMIT; ++x) {
    for (y = 0; y < Y_LIMIT; ++y) {
        if (data[x][y] == 0)
            goto found;
    }
}
cout << "Not found\n";
exit(8);

found:
    cout << "Found at (" << x << ", " << y << ")\n";
```

问题28.1: 当键入不正确的命令时, 示例28.1为什么没有打印出出错信息? 提示: 这也正是我把这个例子放到goto一节中的原因。

示例28.1 def/def.cc

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

main()
{
    char line[10];

    while (1) {
        cout << "Enter add(a), delete(d), quit(q): ";
        cin.getline(line, sizeof(line));

        switch (line[0]) {
            case 'a':
                cout << "Add\n";
                break;
            case 'd':
                cout << "Delete\n";
                break;
            case 'q':
                cout << "Quit\n";
                exit(0);
            default:
                cout << "Error: Bad command " << line[0] << '\n';
                break;
        }
    }
    return (0);
}
```

}

?:命令

?:运算符与if/then/else作用极相似。和if/then/else不一样的是，?:运算符只能用在表达式中。?:的一般形式为：

(表达式)?表达式1:表达式2

例如，下面的语句根据balance的值，将把balance的值或0赋给amount_owed:

```
amount_owed = (balance < 0) ? 0 : balance;
```

下面的宏返回两个自变量中最小者：

```
#define min(x, y) ((x) < (y) ? (x) : (y))
```

注意

把min定义为内联函数，而不是定义为带参宏会更好一些。详细内容请参见第10章，C++预处理器。

逗号运算符

逗号运算符(,)可以用在一组语句中。例如：

```
if (total < 0) {  
    cout << "You owe nothing\n";  
    total = 0;  
}
```

可以写为：

```
if (total < 0)  
    cout << "You owe nothing\n", total = 0;
```

在大多数情况下，用{}来代替逗号。大约只有一个地方使用逗号运算符，即在for语句中。下面的for循环将使两个计数变量two和three的值分别递增2和3：

```
for (two = 0, three = 0;  
     two < 10;  
     two += 2, three += 3)  
    cout << two << ' ' << three << '\n';
```

()运算符的重载

“()”运算符可以在类中被重载，给类提供一个“缺省”的函数。例如：

```
class example {
public:
    int operator () (int i) {
        return (i * 2);
    }
};
// ....
example examp; e_var;
j = example_var(j);    // j is assigned the value 6 (3 * 2)
```

重载()运算符很少使用。一般的成员函数很容易完成相同的功能，同时，还给用户提供一一个函数名。

指向成员的指针

::*是用于指向类成员的指针的运算符。例如，在下面的代码中，我们把data_ptr说明为一个“指向sample中一个整数的指针”：

```
class sample {
public:
    int i;    // A couple of member variables
    int j;
};
int sample::* data_ptr;
```

现在data_ptr既可以指向sample的i成员，也可以指向j成员。（毕竟，它们都只是sample的整数成员。）

我们设置data_ptr的值，让它指向成员i：

```
data_ptr = &sample::i;
```

普通指针只标识一个项。成员指针标识一个成员，但不能标识一个独立变量。我们只能让data_ptr指向sample的成员，它不能指向一个具体整数。

要使用data_ptr，你必须告诉它你想要的是哪个类：

```
sample a_sample; // A typical sample
sample b_sample;

cout << a_smaple.*data_ptr << '\n';
```

```
cout << b_smaple.*data_ptr << '\n';
```

其中:

```
cout << a_smaple.*data_ptr << '\n';
```

告诉C++我们想输出变量a_sample的元素。变量data_ptr指向sample的一个整数成员。(成员i和j是我们仅有的两个整数成员。)使用类指针还有一种简写方式:

```
sample *sample_ptr = &sample1;
```

```
cout << sample_ptr->*data_ptr << '\n';
```

指向成员指针的语法有一点复杂,并不是特别有用。我只见到它被一位极聪明的程序员使用过一次。(但是,第一位维护程序员立即把它去掉了。)

“吸血鬼”特性

本节讨论在C++标准草案*中已定义,但在已有的编译器†中还没有实现的一些特性。

“‘吸血鬼’特性”是指语言中已经问世的那些特性。

因为我不能实际使用这些特性,所以这里提供的信息是肤浅的。

bool 可为真或假的布尔类型

const_cast 新的类型转换,结果为常量

dynamic_cast 充分利用运行时信息的新的类型转换运算符,它能在两个类型之间进行安全转换

false 用于bool类型的新常量

mutable 指明类的常量实例成员可以被改变的修饰符

namespace 该关键字把程序划分到不同的名字空间或对模块排序,这样使程序更好

reinterpret_cast 帮助程序员安全地从基类转换为派生类的类型转换

static_cast 另一个新的类型转换运算符

true 用于bool类型的新常量

* Working Paper for Draft Proposed International Standard for Information Systems——
Programming Language C++(20 September 1994), American National Standards Institute.

† Turbo-C++, Borland C++ Version 4.5, Sunpro CC Version 4.0, g++ Version 2.5.8.

typeid 允许程序员得到运行时程序内部的类型信息

注意

许多头文件为“bool”、“true”和“false”定义了宏。这不会给还没有实现bool类型的编译器带来问题。但是，一旦编译器实现了标准，这样的宏将导致问题。

答疑

问题28.1：编译器没有看到缺省行，因为我们把“default”错拼为“defualt”。这并不是一个错误，因为“defualt”是一个合法的goto标号。这正是编译该程序时得到下列警告信息的原因：

```
def.c(26): warning: defualt unused in function main
```

警告信息的意思是：我们定义了一个goto标号，但却从没使用。

本章内容:

- 概述
- 设计
- 说明
- switch语句
- 预处理器
- 风格
- 编译
- C++程序员的10条戒律
- 最后的注解
- 答疑

第29章

程序设计格言

概 述

- 注释、注释、注释。在程序中写大量的注释。它们告诉其他程序员你做了什么。它们还告诉你你做了什么。
- 使用“KISS”原理（保持程序简单）。清楚和简单比复杂和绝妙更好。
- 避免副作用。++和--的使用要单独放在一行。
- 永远不要把赋值语句放在条件中。永远不要把赋值语句放在其它语句中。
- 了解=和==的区别。用=代替==是常见的错误，并且很难发现。
- 永远不要不声不响地不做任何事。

```
// Don't program like this
for (index = 0; data[index] < key; ++index);
// Did you see the semicolon at the end of the last line?
```

要放注释。

```
for (index = 0; data[index] < key; ++index)
    /* Do nothind */;
```

- 练习编码。练习是人们掌握一定水平的专业技能和创作技能（如艺术家、运动员）关键的一步。或者，帮助其他人学习编程。这可以给你一个好的练习机会，让你复习你所学到的知识，或思考你了解的内容。

设计

- 如果要你选择相对较快的方案或是比较灵活的方案的话，就选择更灵活的那一种。因为你更能重用它，或从中学到点什么。你还可能因为修改了需求并重新编写代码而受到后人的称赞。
- 永远不要希望每位使用者输入的内容都是你想要的。如果一只猫碰到了键盘，你的程序会怎么样呢？
- 小心有符号和无符号惯例及上溢/下溢条件。

说明

- 一行只放一个变量说明，并注释它们。
- 让变量名足够长，以便读者不费劲就可以明白，但是也不要太长了，免得键入时困难。（通常两三个字就足够了。）
- 永远不要使用缺省返回说明。如果一个函数返回一个整数，就把它的类型说明为int。

switch语句

- 在switch语句中一定放一个缺省情况。哪怕它不做任何事情，也要写上：

```
switch (表达式) {  
    default:  
        /* Do nothing */;  
        break;  
}
```

- switch中的任何一种情况都应该以break结束，或以/* fall through */语句结束。

预处理器

- 给在预处理器的#define命令所定义每个常量表达式加括号():

```
#define BOX_SIZE (3 * 10) /* Size of the box in pixels */
```

- 给带参宏中每个自变量加括号():

```
#define SQUARE(x) ((x) * (x))
```

- 宏中若有完整语句的话，要写花括号{}:

```
// A fatal error has occurred. Tell user and abort
#define DIE(msg) {(void)printf(msg);exit(8);}
```

- 在条件编译中使用#ifdef/#endif命令时，在程序的开头写上#define和#undef语句，并注释它们。
- 尽可能地用const代替#define。
- 宁愿使用内联函数，而不使用带参宏。

风格

- 包含在{}中的一个代码块不应长过两页纸。把大块划分为几个更小更简单的过程。
- 当你的代码接近右边界时，就是把过程划分为几个更小更简单过程的时候了。
- 永远为类定义构造函数、析构函数和拷贝构造函数。如果使用了C++的缺省函数，则用注释来“定义”这些程序，如：

```
class example {
public:
    // Example -- default constructor
```

编译

- 建立一个Makefile，这样其他人就知道如何编译你的程序了。
- 编译时，打开所有的警告标志。你永远不知道编译器可能发现什么错误。

C++程序员的10条戒律

1. 除最简单的类以外，你不应该依赖于编译器缺省的构造函数、析构函数、拷贝构造函数，或是赋值。你不应该忘记任何不平凡类的这四大方法。
2. 你应该把析构函数说明并定义为虚函数，这样，其他人可以借助它来“乘凉”。
3. 你不应该滥用继承机制而违背“is-a”规则。
4. 你不应该依赖与编译器、操作系统或是硬件环境相关的任何程序，把这些代码集中放到一个模块中。
5. 你不应该不经过深思熟虑就在最底层增大类的接口。当代码需要维护和扩展时，这种导致错误的做法会束缚用户使用你的类，并可能引起混乱。
6. 你应该与真正的同代人保持友谊。当心，经验不足的你不要和其他人脱离联系。
7. 除极少数情况外，你不应该过多地把数据说明为public或是static。数据应该是你自

己的，不要与其他人共享它们。

8. 你不应该容许在对象内隐藏有危险的指针或是引用。它们危害极大，是引发错误的祸根。
9. 你应该尽可能认真地使用类库。代码重用，不仅重用你自己的，也包括用户的，是OO神圣的圣杯。
10. 你不应该使用printf/scanf，而应使用streams。抛弃C中的不佳用法，代之以灵活可扩展的I/O。

最后的注解

当你认为你已经知道了C++为你做的所有事情的时候——再想一想。总还会有更惊奇的发现。

问题29.1：为什么示例29.1把所有的值都认为是2？（由此而有最后一条格言。）

示例29.1 not2/not2.cc

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

```
main()
```

```
{
```

```
    int number;
```

```
    cout << "Enter a number: ";
```

```
    cin >> number;
```

```
    if (number != 2)
```

```
        cout << "Number is not two\n";
```

```
    else
```

```
        cout << "Number is two\n";
```

```
    return (0);
```

```
}
```

答疑

问题29.1：语句(number != 2)不是关系等式，而是一个赋值语句。它等价于：

```
number = (!2);
```

（因为2非零，所以!2为零。）

程序员常常把不等于弄颠倒了，所以!=变成了=！。语句应该写为：

```
if (number != 2)
```

第六部分

附 录

附录A

ASCII码表

表 A.1 ASCII字符表

十进制	八进制	十六进制	字 符	十进制	八进制	十六进制	字 符
0	000	00	NUL	23	027	17	ETB
1	001	01	SOH	24	030	18	CAN
2	002	02	STX	25	031	19	EM
3	003	03	ETX	26	032	1A	SUB
4	004	04	EOT	27	033	1B	ESC
5	005	05	ENQ	28	034	1C	FS
6	006	06	ACK	29	035	1D	GS
7	007	07	BEL	30	036	1E	RS
8	010	08	BS	31	037	1F	US
9	011	09	HT	32	040	20	SP
10	012	0A	NL	33	041	21	!
11	013	0B	VT	34	042	22	"
12	014	0C	NP	35	043	23	#
13	015	0D	CR	36	044	24	\$
14	016	0E	SO	37	045	25	%
15	017	0F	SI	38	046	26	&
16	020	10	DLE	39	047	27	'
17	021	11	DC1	40	050	28	(
18	022	12	DC2	41	051	29	)
19	023	13	DC3	42	052	2A	*
20	024	14	DC4	43	053	2B	+
21	025	15	NAK	44	054	2C	,
22	026	16	SYN	45	055	2D	-

表 A.1 ASCII字符表 (续)

十进制	八进制	十六进制	字符	十进制	八进制	十六进制	字符
46	056	2E	.	82	122	52	R
47	057	2F	/	83	123	53	S
48	060	30	0	84	124	54	T
49	061	31	1	85	125	55	U
50	062	32	2	86	126	56	V
51	063	33	3	87	127	57	W
52	064	34	4	88	130	58	X
53	065	35	5	89	131	59	Y
54	066	36	6	90	132	5A	Z
55	067	37	7	91	133	5B	[
56	070	38	8	92	134	5C	\
57	071	39	9	93	135	5D	]
58	072	3A	:	94	136	5E	^
59	073	3B	;	95	137	5F	_
60	074	3C	<	96	140	60	`
61	075	3D	=	97	141	61	a
62	076	3E	>	98	142	62	b
63	077	3F	?	99	143	63	c
64	100	40	●	100	144	64	d
65	101	41	A	101	145	65	e
66	102	42	B	102	146	66	f
67	103	43	C	103	147	67	g
68	104	44	D	104	150	68	h
69	105	45	E	105	151	69	i
70	106	46	F	106	152	6A	j
71	107	47	G	107	153	6B	k
72	110	48	H	108	154	6C	l
73	111	49	I	109	155	6D	m
74	112	4A	J	110	156	6E	n
75	113	4B	K	111	157	6F	o
76	114	4C	L	112	160	70	p
77	115	4D	M	113	161	71	q
78	116	4E	N	114	162	72	r
79	117	4F	O	115	163	73	s
80	120	50	P	116	164	74	t
81	121	51	Q	117	165	75	u

表 A.1 ASCII字符表 (续)

十进制	八进制	十六进制	字 符
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

附录B

范 围

表B.1和B.2列出不同变量类型的范围。

表 B.1 32位UNIX机器

名 称	位 数	最低值	最高值	精 度
int	32	-2,147,483,648	2,147,483,647	
short int	16	-32,768	32,767	
long int	32	-2,147,483,648	2,147,483,647	
unsigned int	32	0	4,294,967,295	
unsigned short int	16	0	65,535	
unsigned long int	32	0	4,294,967,295	
char	8	System Dependent		
unsigned char	8	0	255	
float	32	-3.4E+38	3.4E+38	6 digits
double	64	-1.7E+308	1.7E+308	15 digits
long double	64	-1.7E+308	1.7E+308	15 digits

表 B.2 Turbo-C++, Borland C++和其它大多数16位系统

名 称	位 数	最低值	最高值	精 度
int	16	-32,768	32,767	
short int	16	-32,768	32,767	
long int	32	-2,147,483,648	2,147,483,647	
unsigned int	16	0	65,535	

表 B.2 Turbo-C++, Borland C++和其它大多数16位系统 (续)

名 称	位 数	最低值	最高值	精 度
unsigned short int	16	0	65,535	
unsigned long int	32	0	4,294,967,295	
char	8	-128	127	
unsigned char	8	0	255	
float	32	-3.4E+38	3.4E+38	6 digits
double	64	-1.7E+308	1.7E+308	15 digits
long double	80	-3.4E+4932	3.4E+4932	17 digits

附录C

运算符优先级

规则

实用的部分运算符优先级规则

表 C.1 实用的运算符优先级规则

优先级	运算符
1	* (multiply) / %
2	+ -

括号可以加在任何项外。

标准规则

表 C.2 标准C++优先级规则

优先级	运算符
1	() [] -> . :: * -> * (type)
2	! ~ ++ -- (type) - (unary) * (de-reference) & (address of) sizeof
3	* (multiply) / %
4	+ -
5	<< >>
6	< <= > >=
7	== !=
8	& (bitwise AND)

表 C.2 标准C++优先规则 (续)

优先级	运算符
9	$\wedge$
10	$!$
11	$\&\&$
12	$ $
13	$?:$
14	$=$ $++$ $--$ etc.
15	$,$

附录D

用幂级数计算 正弦函数

该程序使用幂级数计算正弦函数。它使用一种非常有限的浮点数格式来说明在使用浮点数时可能遇到的一些问题。

该程序计算幂级数中的每一项，并显示计算结果。直到最后一项的值太小了，对结果没有影响时停止计算。为了便于比较，在计算正弦函数的同时，还显示库函数sin的结果。

程序的调用方式为：

sine 值

这里，值是一个弧度数。例如，要计算sin(0)，则使用下列命令：

```
% sine 0
x**1      0.000E+00
1!         1.000E+00
x**1/1!    0.000E+00
1 term computed
sin(0.000E+00)=
  0.000E+00
Actual sin(0)=0
```

计算sin(π)时使用下列命令：

```
% sine 3.141
x**1      3.141E+00
1!         1.000E+00
x**1/1!    3.141E+00
total      3.141E+00

x**3      3.099E+01
```

3! 6.000E+00
x**3/3! 5.165E+00
total -2.024E+00

x**5 3.057E+02
5! 1.200E+02
x**5/5! 2.548E+00
total 5.239E-01

x**7 3.016E+03
7! 5.040E+03
x**7/7! 5.985E-01
total -7.457E-02

x**9 2.976E+04
9! 3.629E+05
x**9/9! 8.201E-02
total 7.438E-03

x**11 2.936E+05
11! 3.992E+07
x**11/11! 7.355E-03
total 8.300E-05

x**13 2.897E+06
13! 6.227E+09
x**13/13! 4.652E-04
total 5.482E-04

x**15 2.858E+07
15! 1.308E+12
x**15/15! 2.185E-05
total 5.263E-04

x**17 2.819E+08
17! 3.557E+14
x**17/17! 7.927E-07
total 5.271E-04

x**19 2.782E+09
19! 1.217E+17
x**19/19! 2.287E-08
total 5.271E-04

x**21 2.744E+10

```

21!      5.109E+19
x**21/21! 5.371E-10
11 term computed
sin(3.141E+00)=
    5.271E-04
Actual sin(3.141)=0.000592654

```

Makefile

用于UNIX的Makefile:

示例D.1 *sin/Makefile*

```

sine: sine.cc
    g++ -g -Wall -o sine sine.cc -lm

clean:
    rm sine
Makefile for Turbo-C++
[File: sin/Makefile.dos]
#
# Makefile for Borland's Turbo-C++ compiler
#
CC = tcc
#
# Flags
#      -N  -- Check for stack overflow
#      -v  -- Enable debugging
#      -w  -- Turn on all warnings
#      -ml -- Large model
#      -A  -- Force ANSI compliance
#
CFLAGS = -N -v -w -ml -A
#
sine.exe: sine.c
    $(CC) $(CFLAGS) -esine sine.c

```

程序: sine.cc

示例D.2 *sine/sine.cc*

```

/*****

```



```

* sine -- compute sine using very simple floating *
* arithmetic *
*
* Usage: *
*     sine <value> *
*
*     <value> is an angle in radians *
*
* Format used in f.ffffe+X *
*
* f.fff is a 4-digit fraction *
*     + is a sign (+ or -) *
*     X is a single-digit exponent *
*
*  $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$  *
*
* Warning: This program is intended to show some of *
* problems with floating point. It is not *
* intended to be used to produce exact values for *
* the sine function. *
*
* Note: Even though we specify only one digit for the *
* exponent, two are used for some calculations. *
* This is due to the fact that printf has no *
* format for a single-digit exponent. *
*****/
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    float total; // Total of series so far
    float new_total; // Newer version of total
    float term_top; // Top part of term
    float term_bottom; // Bottom of current term
    float term; // Current term
    float exp; // Exponent of current term
    float sign; // +1 or -1 (changes on each term)
    float value; // Value of the argument to sin
    int index; // Index for counting terms -

```

```

char    *float_2_ascii(float number); // Turn floating point to ASCII
float    fix_float(float number);      // Round to correct digits
float    factorial(float number);      // Compute n!

if (argc != 2) {
    cerr << "Usage is:\n";
    cerr << "  sine <value>\n";
    exit (8);
}

value = fix_float(atof(&argv[1][0]));

total = 0.0;
exp = 1.0;
sign = 1.0;

for (index = 0; /* Take care of below */ ; ++index) {
    term_top = fix_float(pow(value, exp));
    term_bottom = fix_float(factorial(exp));
    term = fix_float(term_top / term_bottom);
    cout << "x**" << int(exp) << " " <<
        float_2_ascii(term_top) << '\n';
    cout << exp << "! " << float_2_ascii(term_bottom) << '\n';
    cout << "x**" << int(exp) << "/" << int(exp) << "! " <<
        float_2_ascii(term) << "\n";

    new_total = fix_float(total + sign * term);
    if (new_total == total)
        break;

    total = new_total;
    sign = -sign;
    exp = exp + 2.0;
    cout << "  total  " << float_2_ascii(total) << '\n';
    cout << '\n';
}

cout << index + 1 << " term computed\n";
cout << "sin(" << float_2_ascii(value) << ")=\n";
cout << " " << float_2_ascii(total) << '\n';
cout << "Actual sin(" << atof(&argv[1][0]) << ")=" <<
    sin(atof(&argv[1][0])) << '\n';
return (0);
}

/*****
 * float_2_ascii -- turn a floating point string      *

```



```

*      into ASCII
*
* Parameters
*      number -- number to turn into ASCII
*
* Returns
*      pointer to the string containing the number
*
* Warning: Uses static storage, so later calls
*           overwrite earlier entries
*****/
char *float_2_ascii(float number)
{
    static char result[10];    // Place to put the number

    sprintf(result, "%8.3E", number);
    return (result);
}
/*****
* fix_float -- turn high-precision numbers into
*              low-precision numbers to simulate a
*              very dumb floating-point structure
*
* Parameters
*      number -- number to take care of
*
* Returns
*      number accurate to 5 places only
*
* Note: This works by changing a number into ASCII and
*       back. Very slow, but it works.
*****/
float fix_float(float number)
{
    float  result;    // Result of the conversion
    char   ascii[10]; // ASCII version of number

    (void)sprintf(ascii, "%8.4e", number);
    (void)sscanf(ascii, "%e", &result);
    return (result);
}
/*****
* factorial -- compute the factorial of a number
*
* Parameters

```

```

*      number -- number to use for factorial      *
*
* Returns
*      factorial(number) or number!
*
* Note: Even though this is a floating-point routine,
*      using numbers that are not whole numbers
*      does not make sense
*****/
float factorial(float number)
{
    if (number <= 1.0)
        return (number);
    else
        return (number *factorial(number - 1.0));
}

```


词汇表

- ^ 按位异或运算符。
- ~ 按位补运算符，即把所有位取反。
- ! 逻辑非运算符。
- != 不等于关系运算符。
- { } 见花括号。
- | 按位或运算符。
- || 逻辑或运算符。
- % 取模运算符。
- & 1. 按位与运算符。
2. 放在变量名前的符号（如&x）。意思是该变量的地址（x的地址）。用来把一个值赋给一个指针。
用来说明一个引用变量。
- && 逻辑与运算符（用于比较运算）。
- * 1. 乘法运算符。
2. 放在指针变量名前的符号，意思是取指针变量指向地址中存储的值。（*x是指取x中存储的值。）有时也叫逆引用运算或间接运算。
- ++ 加法运算符。
- ++ 增量运算符。
- 减法运算符。
- 减量运算符。
- > 用来得到类或结构指针中的成员。
- >* 表示由指向成员的指针所指向的项。
- / 除法运算符。
- < 小于关系运算符。
- << 1. 左移运算符。

2. `iostream`包输出符。

:: 作用域运算符。表示一个具体标识符属于哪个类。

::* 说明指向类成员的指针。

<= 小于等于关系运算符。

== 等于关系运算符。

> 大于关系运算符。

>= 大于等于关系运算符。

>> 1. 右移运算符。

2. `iostream`包输入符。

'\0'

串尾符 (NULL字符)。

#define

C++预处理器命令，它定义一个名字的替代文本。

#endif

预处理器宏的结束括号，相应的开始括号是**#ifdef**命令。

#ifdef

预处理器命令，检查是否定义了一个宏名。如果已定义了，则后续代码将加到源代码中。

#ifndef

预处理器命令，检查是否没定义一个宏名。如果当前它没有定义，则后续代码将加到源代码中。

#include

预处理器命令，它使所指文件插入到**#include**的位置。

#undef

预处理器命令，它删除**#define**。

_ptr

本书中的体例。所有指针变量名以**-ptr**为后缀。

抽象类

含有一个或多个纯虚函数的类。

精度

衡量实数表示中内在误差的一个量。

地址

表示内存中存储位置的值。

与

布尔运算，如果两个操作数中有一个为0，则结果为0；如果两个操作数均为1，则结果为1。

ANSI C

符合American National Standards Institute Committee X3J规范的任何一个C版本。

ANSI C++

符合American National Standards Institute规范的任何一个C++版本。到本书写作时，标准只有草案形式，还有许多细节仍在探讨中。

数组

按一维或多维下标排定的数据元素集。在C++中数组存储在连续内存中。

ASCII

American Standard Code for Information Interchange（美国信息交换用标准代码）。一种表示字符的编码。

赋值语句

把值存储在变量中的操作。

auto

C++关键字，用来建立一个临时变量。

自动变量

见临时变量。

基类

用作派生类基础的类。

位

二进制数字，可以为数字0或1。

位域

作为一个整体的一组邻近的位。C++语言的这个特征允许访问独立的位。

位反向

反转一个操作数中的所有位。又见补。

位运算符

见按位运算符。

位图

计算机图形，图形输出设备中的每个象素都由一个位或一组位来控制。

按位运算符

对两个操作数执行布尔操作的运算符。它把操作数中的每一位看作是独立的位，一位一位地执行相应的运算。

块

包含在花括号中的一段代码。

Borland C++

由Borland公司开发的用于个人计算机的一个C++语言版本。它是Borland公司的Turbo-C++产品的更高级版本。

（注释）框

使用垂直和水平星号及其它印刷字符在注释周围所画的框，为的是把它和代码分开。

break

中断for、while、switch和do/while语句最内层执行的一种语句。

断点

程序中的一个位置。程序执行到此时，挂起正常的执行，控制回到调试程序中。

缓冲I/O

在I/O流的源和目标之间使用中间存储（一个缓冲区）的输入/输出。

字节

八个位组成的组。

C

由Dennis Ritchie于1974年在贝尔实验室开发的通用计算机程序设计语言。C被认为是一种中级到高级之间的语言。

C++

基于C的一种语言，它是Bjarne Stroustrup于1980年开发的。最初叫“带类的C”，它已经发展为自己的语言。

C++代码

用C++语言编写的计算机指令。

C++编译器

把C++源代码翻译为机器代码的软件。

C++语法

见语法。

引用调用

一种参数传递机制，其中真正的参数并不传递给函数，而是用一个指针指向它。（又见值调用。）

值调用

一种过程调用。其中，通过传递参数的值来传递参数。（又见引用调用。）

case

当作switch语句中可选项之一的标号。

替换

通过明确表示类型转换符，把一个变量从一种类型转换为另一种类型。

cerr

用于C++的标准错误流。（对应于C的stderr。）

CFront

把C++代码翻译为C代码的一个程序。该程序是第一个C++编译器的基础。目前大多数编译器中都不使用它了，因为现在已经有了许多的原生的C++编译器。

CGA

彩色图形适配器。用于IBM PC的一种常见彩色图形卡。

char

C++关键字，用来说明表示字符或小整数的变量。

cin

字符输入。用于C++的标准输入流。（对应于C的stdin。）

类

包含不同数据类型的一种数据结构，保护成员及操纵成员的函数。

(变量的) 类

见存储类。

清除一个位

把一个独立的位设置为0的操作。在C++中没有定义这个操作。

clog

用于C++的标准注册文件。

代码设计

用一般术语描述程序如何完成它的功能的一个文档。

编码

用一种计算机语言编写程序的行为。

命令行选项

控制像编译器这样的程序运行的选项，它从计算机控制台键入。

注释

包含在计算机程序中的文本，它仅仅是为了提供程序的有关信息。注释是一个程序员对自己及未来程序员的注解。这些文本被编译器所忽略。

注释块

一组相关的注释，它表达一个程序或一段程序的一般信息。

编译

把源代码翻译为机器代码。

编译器

完成编译功能的一个系统程序。

编译过程

见编译。

补

算术或逻辑运算。逻辑补等于取反运算，或非运算。

计算机语言

见程序设计语言。

条件编译

通过测试代码中的条件命令，根据条件的真假，有选择地编译程序中一部分代码的能力。

continue

一种流控制语句，导致循环下一次执行的开始。

控制语句

根据条件测试结果，判定将执行哪个语句的一种语句。

控制变量

在循环执行期间由系统修改的一种变量。当变量达到一个预定值时，循环结束。

转换规格说明

C语言中用于printf系列函数的一个串，它规定一个变量的打印格式。

cout

用于C++程序的标准输出。（对应于C的stdout。）

花括号

字符{或}之一。它们用在C++中，标定作为一个整体的元素组。

调试

找出并删除程序中错误的过程。

判定语句

测试由程序建立的条件，根据这个判定来改变程序流的一个语句。

说明

程序中使用的变量类型和名字的规格说明。

default

在switch语句中，如果没有匹配的case值，就把它看作是一个case标号。

define语句

见#define。

delete

把由new建立的一个类或变量返回给堆的一个指令。

逆引用运算符

表示访问由一个指针变量或是一个地址表达式所指向的值的运算符。又见*。

派生类

在另一个类，即基类之上建立的一个类。

伪指令

给预处理器的一个命令（不同于产生机器代码的一条语句）。

double

C++语言的一个关键字，说明一个能存储实数的变量。该数通常需要两倍于float类型的存储空间。

双向链表

一种带有向前和向后指针的链表。又见链表。

双引号

ASCII字符34。在C++中用来标定字符串。

EGA

增强图形适配器。IBM PC中常见的图形卡。

else

if语句中的一个子句，用来指明当if条件失败时将执行什么语句。

enum

C++的一个关键字，定义一个枚举数据类型。

枚举数据类型

含有一组有名值的一种数据类型。C++编译器给其中的每个元素都指定一个整数值。

EOF

定义在stdio.h中的文件结束字符。

escape字符

一个特殊的字符，它改变了其后所跟字符的意思。在C++中表示为反斜杠\。

异或

布尔运算。如果两个操作数相同，结果为0；如果两个操作数不同，结果为1。

可执行文件

含有机码的一个文件，它已经被连接了，而且可以在一台计算机上运行。

指数

浮点数的一部分，表示整数幂，基数乘上幂次才能表示数本身。

指数上溢

由浮点数运算引起的一种情况。此时，得到了一个很大的指数，以至于在分配给指数的位域中装不下。

指数下溢

由浮点数运算引起的一种情况，此时，得到了一个很大的负值指数，以至于在分配给指数的位域中装不下。

extern

C++关键字，用来说明定义在当前文件之外的变量或是函数。

快速原型

自顶向下的程序设计方法。先编写规格中的最小部分，让其可以运行，然后再扩大之。

fclose

关闭文件函数。来自于原来C风格的I/O包stdio。

fflush

强制输出缓冲区函数。来自于原来C风格的I/O包stdio。

fgetc

读单个字符函数。来自于原来C风格的I/O包stdio。

FILE

定义在stdio中的一个宏，它说明一个文件变量。来自于原来C风格的I/O包stdio。

文件

看作一个整体的一组相关记录。

float

C++关键字，说明可以存储一个实数的一个变量。

浮点数

一种记数系统，表示为一个小数部分和一个指数。这种系统可以处理非常大和非常小的数。

浮点数例外（核心崩溃）

因除0或其它非法算术操作而导致的一个错误。因为它是由整数和浮点数错误引起的，所以有时很令人费解。

浮点数硬件

不必借助于软件，可以直接完成浮点数运算的电路。在个人计算机中，它就是数学协处理器。像80486这样的较先进的处理器中都有内置的浮点运算单元。

fopen

打开文件用于I/O流的函数。来自于原来C风格的I/O包stdio。

fprintf

把二进制数据转换为字符数据并写入一个文件中的函数。来自于原来C风格的I/O包stdio。

fputc

写一单个字符的函数。来自于原来C风格的I/O包stdio。

fputs

写一单行的函数。来自于原来C风格的I/O包stdio。

fread

二进制I/O输入函数。来自于原来C风格的I/O包stdio。

free

把数据返回给内存池的C函数。在C++中已经不使用，并被C++的delete运算符所替代。又见malloc。

Free Software Foundation

一个程序员小组，他们编制并奉献了高质量的免费软件。在他们的产品中，包括有编辑器**emacs**和C++编译器**g++**。他们的地址是：Free Software Foundation, Inc, 675 Massachusetts Ave., Cambridge, MA 02139, 电话是：(617)876-3296。

friend

虽然不是类中的一个成员，但是可以访问该类中私有成员的一个函数。

fscanf

类似于**scanf**的一个输入程序。来自于原来C风格的I/O包**stdio**。

fstream.h

用于文件I/O的C++包。

函数

返回一个值的过程。

fwrite

一个二进制I/O输出函数。来自于原来C风格的I/O包**stdio**。

通用指针

一个可以指向任何变量、不用强制转换为变量类型的指针。不用考虑内容而指向存储的一个指针。

Ghostscript

类似于**Postscript**的解释器，它可以从Free Software Foundation公司免费得到。

全局变量

在整个程序中都知道的变量。

监视位

浮点数计算中使用的一个额外精度数字，它确保不丢失精度。

头文件

见包含文件。

堆

new使用的内存中的一部分，通过**new**返回结构和类使用的空间。使用**delete**运算符可把空间返回给这个池。

十六进制数

基为16的数。

高级语言

计算机语言的一个级别，界于机器语言和自然（人类）语言之间。

I/O操作符

一组函数。虽然它的“输出”或“输入”没有引起I/O操作，但设置了各种转换标志或参数。

IEEE浮点标准

IEEE标准754，它规定了浮点数格式、精度和一些非分数值的标准。

if

根据一个条件的真假值来选择执行程序的一部分的语句。

实现依赖

是指因为计算机系统的变化性，使得从计算机操作或是软件得到的结果不统一这样一种情形。

包含文件

一个文件，执行预处理器命令`#include`后，该文件可以与源代码合并在一起。又叫头文件。

相容析取

见或。

下标

一个值、变量或是表达式，选取数组中的一个具体元素。

间接运算符

见逆引用运算符。

信息隐藏

一种代码设计系统，它使得模块之间传递的信息量达到最小。这种思想是尽可能地把信息隐藏在模块内部，只有当绝对需要时，才共享信息。

指令

定义由计算机完成的一个操作的一组位或是字符。

int

C++中说明一个整数的关键字。

整型

一个整数。

交互调试器

用来调试程序的一个程序。

反运算符

执行非操作的逻辑运算符。

iostream.h

C++的标准I/O包。

左移

移位操作。在位域内向左移动指定的位数，空出的位填0。

库

文件集合。

链表

数据结点集合。每个结点含有一个值和一个指向链表中下一项的指针。

局部包含文件

私用库中的文件。使用预处理器命令`#include "文件名"`，可以把它插入到源代码中。

局部变量

其作用域局限于定义它的块中的变量。

逻辑与

布尔运算。如果两个白变量都为真，则结果为真。当用于整数时，每个位单独运算。

逻辑运算符

C++运算符，它对两个操作数进行逻辑运算，返回真值或假值。

逻辑或

布尔运算。如果两个白变量中有一个为真，则结果为真。当用于整数时，每个位单独运算。

long

说明长于正常精度的一种数据类型的修饰符。

机器代码

二进制格式的机器指令，它不需要转换就可由机器直接识别。

机器语言

见机器代码。

宏

一小段文本或文本模板，它可以被扩充到更长的文本中。

宏处理器

产生代码的一个程序，它把值放到已定义模板中的相应位置。

(数) 量

不考虑符号时一个数的值。

(程序) 维护

因为计算机系统外部条件的改变而进行的程序修改。

make

UNIX和MS-DOS/Windows系统下的一个实用程序，它管理程序的编译过程。

Makefile

含有用于make实用程序命令的文件。

malloc

管理内存堆的C过程。该函数现在已经不用了。C++的new运算符已经取代了这个函数。

屏蔽

位的一种模式，用来控制另一组位的留与去。

成员

类或结构的一个元素。

模块

程序的一个逻辑部分。

MS-DOS

微软公司开发的用于IBM个人计算机的一个操作系统。

new

C++运算符，它从堆中得到一个新变量。

换行符

一个字符，它使得输出设备定位到下一行的开始位置。

无意义数字

前导数字，对一个数的值不起作用（在补码格式下，0用于正数，1用于负数）。

标准化

移动浮点数的小数部分（并调整幂次），这样，在小数部分中就没有前导0了。

非

布尔运算，它得到操作数的逻辑反。NOT 1得0，NOT 0得1。

不是一个数

IEEE 754中定义的一个特殊值，它表示浮点数运算得到了一个无效结果。

NULL

一个常量，其值为0，表示指向空。

空指针

其位全为0的一个指针。它表示该指针不指向任何有效数据。

面向对象的设计

一种设计方法，程序员把他或她的设计建立在数据对象（类）和它们之间的连接上。

八进制数

基数为8的数。

补

把一个整数中的所有位反转的操作。1变为0，0变为1。

运算符

表示要执行的一个动作的符号。

或

布尔运算。如果两个操作数中有一个为1，则结果为1；如果两个操作数均为0，则结果为0。

上溢错误

因算术运算的结果大于计算机为结果而提供的存储空间而导致的一种算术运算错误。

压缩结构

按需分配位域的大小而不考虑字边界的一种数据结构技术。

填充字节

加到一个结构中的一个字节，它的唯一目的是确保内存对齐。

参数

一个数据项，可以给它赋一个值。常常是指调用函数和被调用函数之间传递的自变量。

参数宏

含有一个模板的宏，模板中带有参数的插入点。

宏的参数

要插入到宏中定义参数位置的值。在扩展宏时才实施插入操作。

永久变量

在程序启动之前建立的一个变量。它在程序启动之前进行初始化，在整个程序的运行期间始终保留它的内存。

像素

最小的显示元素，它可以单独地被赋以亮度和颜色。本词来自于Picture Element（图元）。

指针

一种可以存储内存地址的数据类型。

指针算术运算

C++允许对指针进行三种算术运算：

1. 指针加上一个数。
2. 指针减去一个数。
3. 一个指针可以减去另一个指针。

指针变量

见指针。

可移植的C编译器

Stephen Johnson编写的一个C编译器，它改为不同计算机体系结构下的编译器相对要容易一些。

精度

衡量差不多相等的两个值之间差别的一种机制。

预处理器

执行预处理的一个程序，它的目的是把宏代码模板扩展为C++代码。

预处理器命令

预处理器的一個命令。

printf

产生格式输出的一个C库程序。来自于原来C风格的I/O包stdio。

private

C++关键字，表示其后的成员只能被类中的成员或类的友元来访问。

过程

可从不同程序或程序的不同部分调用的一个程序段。它不返回值（void类型函数）。

程序

让计算机执行一系列操作的一组指令。

程序头

程序开头的注释块。

程序规格说明

说明程序要做什么的一段手写文档。

程序员

为计算机写程序的人。

(计算机) 程序设计

用代表计算机指令的语言来描述一个问题的解决方案的过程。

程序设计语言

用于表示计算机程序的一种形式记号体系。

protected

C++关键字，表示其后的成员可被类中的成员、类的友元函数或派生类的成员访问，但不能被这些以外的函数来访问。

伪代码

一种编码技术。过程的明确描述由易读的语言指令来书写，而不必考虑计算机语言的语法规则。

public

C++关键字，表示其后的成员可被类外的成员来访问。

纯虚函数

没有缺省函数体的虚函数。含有一个纯虚函数的类不能直接使用，必须作为另一个类的基类。（又见派生类和抽象类。）

修饰符

修改数据说明含义的一个字。

基数

正整数，数位的权乘上基数得到下一个更高数位的权。

实数

在固定基数计数系统中，由有穷或无穷数表示的一个数。

递归

当一个函数直接或间接地调用它本身时，发生递归。

改向

命令行选项>file允许用户把程序的结果输出到一个文件中，而不是输出到屏幕上。类似的选项<file将从文件中取输入信息，而不是从键盘上。

约简

用简单操作替代昂贵操作的过程。

关系运算符

比较两个操作数，根据其关系的真或假得出真假值的运算符。

交付

编程工程完成的这一时刻，即程序备好可以使用了。

重放文件

用来替代键盘数据标准输入的文件。

return语句

标志函数结束，并使控制返回调用者的语句。

修正

对程序的明显改动。

右移

移位操作，在位域内向右移动指定的位数。

四舍五入

在位置表示中，删去或忽略一个或多个最小的有效数字，然后用某些特定的规则来调整剩余的部分，如使误差最小。

截断误差

截断时产生的误差。

保存文件

一个调试工具，用户键入的所有内容都保存到一个文件中以备。又见重放文件。

scanf

直接从键盘读数据的库输入函数。很难使用。在大多数情况下，使用fgets/sscanf两个函数。来自于原来C风格的I/O包stdio。

作用域

变量的作用域是程序中知道变量名的那一部分。

段违例

程序试图访问其地址空间之外的内容而引起的一个错误。因逆引用一个坏指针而引起。

设置一个位

把一个指定位设置为1的操作。这不是C++中定义的操作。

setw

设置下一个输入宽度的I/O操作符。

移位

在位域内向左或向右移动位的操作。

short

与整数相同或更小的一种算术数据类型。

副作用

一个语句中主操作之外执行的一个操作，如赋值语句中的变量增量操作：

```
result =begin++ - end;
```

有效数

浮点数中最有效数字，不考虑其基点位置。

有效数字

保持给定精度的一个数字。

单引号

ASCII字符39。在C++中用在单字符两侧。

sizeof

返回一种数据类型变量大小（字节数）的运算符。

源代码

在被计算机翻译之前的原码形式。

源文件

包含源代码的文件。

规格说明

描述程序作什么的文档。

sprintf

类似于fprintf，但它使用一个串输出。来自于原来C风格的I/O包stdio。

sscanf

库输入程序。来自于原来C风格的I/O包stdio。

栈

内存的一个区域，用来临时保存一串数据和指令。

栈上溢

因程序中的变量使用了太多的临时空间（栈空间）而引起的一个错误。

栈变量

见临时变量。

static

一种存储类属性。在一对花括号内时，它表示一个永久变量。在一对花括号外时，它表示一个局部文件变量。对于类成员来说，它表示一个变量或函数是独立的实例。它的完整的用法请见表14.1。

stderr

预定义的标准错误文件。来自于原来C风格的I/O包stdio。

stdin

预定义的输入源。来自于原来C风格的I/O包stdio。

stdio.h

原来C风格的I/O包stdio。

stdout

预定义的标准输出。来自于原来C风格的I/O包stdio。

存储类

一种变量定义属性，它控制变量在内存中是如何存储的。

串

字符序列或是字符数组。

struct

C++关键字，表示结构数据类型。

结构

一组名字，表示可能有不同属性的一组数据项。

风格单

描述编程风格的文档，它由具体的公司或团体来使用。

Sunview

用在SUN工作站上的图形和窗口系统。

switch

多路分支。根据标志表达式的值，把控制转到几个case语句之一。

语法

构造语句所遵从的规则。

语法错误

构造C++表达式时的错误。

临时变量

从栈中分配存储的变量。每次进入定义它的块时，初始化临时变量。

测试一个位

判定一个具体的位是否被设置的操作。它不是C++中定义的操作。

测试计划

测试的详细说明，程序必须按照它的说明来运行。

文本编辑器

用来建立或修改文本文件的软件。

翻译

用一种替代语言来建立新程序，保证在逻辑上等价于用源语言写的程序。

树

一种层次数据结构。

截断

丢弃实数中小数部分的一种操作。

Turbo-C++

由Borland公司开发的用于个人计算机的一个C++语言版本。

类型转换

见转换。

typedef

从已有类型建立新类型的运算符。

类型语句

建立变量特征的语句。

非缓冲I/O

每一次的读写都导致一次系统调用的I/O操作。

union

允许对同一存储位置指定不同数据名和数据类型的数据类型。

UNIX

通用多用户操作系统，最初由贝尔实验室的Ken Thompson和Dennis Ritchie开发成功。

unsigned

指定int和char变量不包含负号的修饰符。

(程序的) 升级

程序的修改，以改进其性能或提供新特征。

值

赋给一个常量的量。

变量

给一个值取的名字。由变量名表示的数据在程序运行的不同时刻，可以取不同的值。

变量名

给用来存储一个变量的一段内存取的符号名。

版本

用来标识软件特定版的一个术语。通常包括一个版本号。整数部分表示主要的修改。分数部分表示较小的修改或是问题的修正。

virtual

C++关键字，表示一个成员函数可以被派生类中的一个函数所覆盖。

void

C++中的一种数据类型。当用作函数调用中的一个参数时，它表示没有返回值。void *表示返回一个通用指针值。当用在类型转换时，它表示丢弃一个给定值。

volatile

C++关键字，表示一个变量或常量的值可随时改变。该属性用在内存映射I/O、共享内存应用程序和其它高级程序设计中。

while

重复语句。当给定的条件为真时，它就重复地执行一条语句。

X Window系统

一个图形和窗口系统，可从X Consortium公司得到，目前它运行在许多计算机系统上。

基于0的计数

一种计数系统，第一个对象计为0而不是计为1。

[G e n e r a l I n f o r m a t i o n]

书名=实用C++编程大全

作者=

页数=521

SS号=10460666

出版日期=

封面
书名
版权
前言
目录
正文