THE EXPERT'S VOICE® IN C

Beginning C, Fifth Edition

C语言入门经典

（第5版）

［美］Ivor Horton 著
杨浩 译

Apress®
清华大学出版社

THE EXPERT'S VOICE® IN C

C语言入门经典

(第5版)

[美] Ivor Horton 著

杨 浩 译

清华大学出版社

北 京

Ivor Horton

Beginning C, Fifth Edition

EISBN：978-1-4302-4881-1

图书在版编目(CIP)数据

C 语言入门经典(第 5 版) / (美) 霍尔顿(Horton, I.) 著；杨浩 译. —北京：清华大学出版社，2013
书名原文：Beginning C, Fifth Edition
ISBN 978-7-302-34341-7

Ⅰ. ①C…　Ⅱ. ①霍… ②杨…　Ⅲ. ①C 语言—程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 255156 号

责任编辑：王　军　于　平
装帧设计：牛艳敏
责任校对：邱晓玉
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：37.75　　**字　　数**：872 千字
版　　次：2013 年 11 月第 1 版　　**印　　次**：2013 年 11 月第 1 次印刷
印　　数：1~4000
定　　价：69.80 元

产品编号：

作 者 简 介

Ivor Horton 原本是一名数学家，因听闻信息技术工作轻松且收入丰厚而踏足其中。尽管现实情况常常是工作辛苦且收入相对一般，但他至今依然坚持从事计算机工作。Ivor 在不同的时期从事过各种类型的工作，包括程序设计、系统设计、咨询顾问以及管理和实现一些颇为复杂的项目。 Ivor 对于将计算机系统设计和实现应用在各种行业工程设计和运营管理方面有着十分丰富的经验。他能够运用多种编程语言开发特定用途的应用程序，同时还为科研人员和工程人员提供教学，以帮助他们完成这类工作。多年来，他一直撰写编程方面的书籍，近期作品包括 C、C++和 Java 教程。在写书与指导他人之余，一般他会选择钓鱼、旅行和享受生活。

技术审稿人简介

Marc Gregoire 是一名来自比利时的软件工程师。他毕业于比利时天主教鲁汶大学(Catholic University of Leuven)，并拥有该校的计算机工程学硕士学位。一年后，他以优异成绩获得了同一所大学的人工智能专业硕士学位。毕业后，Marc 开始在一家名为 Ordina Belgium 的软件咨询公司工作。作为一名咨询师，他主要为西门子及诺基亚西门子网络公司提供服务，工作内容包括帮助电信运营商在 Solaris 上运行关键性的 2G 与 3G 软件。这项工作需要身处国际范围的团队(跨度从南美地区和美国到欧洲、中东地区、非洲以及亚洲)中完成。现在，Marc 在 Nikon Metrology 从事 3D 激光扫描软件开发工作。

Marc 在 C/C++方面经验丰富，尤其精通 Microsoft VC++和 MFC 框架。此外，Marc 也喜爱 C#，并使用 PHP 创建网页。除了主要对 Windows 开发感兴趣之外，Marc 还对开发 Linux 平台上 24/7 全天候运行的 C++程序颇有心得(例如，EIB 家庭自动化软件)。

自 2007 年 4 月起，Marc 就因他在 Visual C++方面的丰富经验而屡屡荣获每年的 Microsoft 的年度 MVP(Most Valuable Professional，最有价值专家)奖。

Marc 是比利时 C++用户群组(www.becpp.org)的创始人，并经常以 Marc G 的代号活跃在 CodeGuru 论坛中。他还创建了一些免费软件和共享软件，并发布在了他的网站 www.nuonsoft.com 上。此外，Marc 还在 www.nuonsoft.com/blog/上维护自己的博客。

致　　谢

作者只不过是一个大型团队中将书复印成册之人。我想感谢整个 Apress 编辑部与产品组自始至终的帮助与支持。我要感谢 Jonathan Gennick 对于启动本书新版本所做的努力，感谢 Jill Balzano 在整个编辑过程中耐心地帮助解决我遇到的各种难题。

我还要感谢我的技术编辑 Marc Gregoire，感谢他帮助审核文字和检查所有的代码片段与示例。他找错的本领实在了得，他的许多建设性的评论与深思熟虑的建议无疑使本书变成了一本更好的教程。

前　言

欢迎使用《C 语言入门经典(第 5 版)》。研读本书，你就可以成为一位称职的 C 语言程序员。从许多方面来说，C 语言都是学习程序设计的理想起步语言。C 语言很简洁，因此无须学习大量的语法便能够开始编写真正的应用程序。除了简明易学以外，它还是一门功能非常强大的语言，并被专业人士广泛应用在各种领域。C 语言的强大之处主要体现在，它能够应用于各类层次的开发中，从设备驱动程序和操作系统组件到大规模应用程序，它都能胜任。此外，C 语言还可以适用于相对较新的手机应用程序开发上。

几乎所有计算机都包含 C 语言编译器，因此，当你学会了 C 语言，就可以在任何环境下进行编程。最后一点，掌握 C 语言可以为理解面向对象的 C++语言奠定良好的基础。

在作者眼中，有抱负的程序员必将面对三重障碍，即掌握遍布程序设计语言中的各类术语、理解如何使用语言元素(而不仅仅只是知道它们的概念)以及领会如何在实际场景中应用该语言。本书的目的就是将这些障碍降到最低限度。

术语是专业人士及优秀业余爱好者之间的交流必不可少的，因此有必要掌握它们。本书将确保你理解这些术语，并自如地在各种环境下使用它们。这样才能更有效地使用大多数软件产品附带的文档，且能轻松地阅读和学习大部分程序设计语言相关的著作。

理解语言元素的语法和作用固然是学习 C 语言过程中的一个重要部分，但认识语言特性如何工作及应用也同等重要。本书不仅采用了代码片段，还在每个章节中使用一些实际应用示例展示语言特性如何应用于特定的问题。这些示例提供了实践的基础，读者可以通过改动代码观察修改后的结果。

理解特定背景下的程序设计不仅只是应用个别语言元素。为了帮助读者理解它们，本书大部分章节之后都给出了一个较为复杂的应用程序，以应用本章之前学到的知识。这些程序可以帮助你获得开发应用程序的能力与信心，了解如何联合以及更大范围地应用语言元素。最重要的是，它们能让你了解设计实际应用程序与管理实际代码会碰到的问题。

不管学习什么程序设计语言，有几件事情都要意识到。首先，虽然要学的东西很多，但是掌握它们之后，你就会有极大的成就感。其次，学习的过程很有趣，你会深深地体会到这点；第三，只有通过动手实践才能学会编程，这也是本书贯彻的思想。最后，在学习的过程中，肯定会时不时犯许多错误和感到沮丧。当觉得自己完全停滞时，你要做的就是坚持。最终你一定会体验到成功的喜悦，并且回头想想时，你会觉得它也并没有你想象中的那么难。

如何使用本书

作者认为动手实践是学习编程最好的方法，很快你就会编写第一个程序了。每一章都会有几个将理论应用于实践的程序，它们也是本书的核心所在。建议读者手工键入并运行书中的示例，因为手工键入可以极大地帮助记忆语言元素。此外，你还应当尝试解决每章末尾的所有练习题。当你第一次将一个程序运行成功，尤其是在解决自己的问题后，你会感觉到很大的成就感和惊人的进步速度，那时你一定会觉得一切都挺值得。

刚开始，学习的进展不会太快，不过随着逐渐深入，你的学习速度会越来越快。每一章都会涉及许多基础知识，因此在学习新的内容之前，需要花些时间确保理解前面学习过的所有知识。实践各部分的代码，并尝试实现自己的想法，这是学习程序设计语言的一个重要部分。尝试修改书中的程序，看看还能让它们做些什么，那才是有趣之处。不要害怕尝试，如果某些地方不太明白，尝试输入一些变体，看看会出现什么情况。出错并没什么大不了，你会从出错中学到很多知识。一个不错的方法是彻底通读每一章，了解各章的范围，然后回过头来过一遍所有的示例。

你可能会觉得某些章末尾的练习题非常难。如果第一次没有完全搞明白，不用担心。之所以第一次觉得困难是因为它们通常都是将你所学的知识应用到了相对复杂的问题中。如果你实在觉得困难的话，可以略过它们继续学习下一章，然后再回头研究这些程序。你甚至可以阅读完整本书再考虑它们。尽管如此，如果你能完成练习的话，说明你取得了真正的进步。

本书读者对象

《C 语言入门经典(第 5 版)》的目的是教会读者如何尽可能简单快速地编写有用的程序。在阅读完全书后，读者会彻底了解 C 语言编程。这本教程面向的是那些之前编过一些程序，了解背后的概念，并且希望通过学习 C 语言进一步扩展知识的读者。尽管如此，本书并未假设读者拥有先前的编程知识，因此如果你刚刚接触编程，本书依然是你的不错选择。

使用本书的条件

要使用本书，你需要一台安装 C 编译器和库的计算机以执行书中的示例，以及一个程序文本编译器用于创建源代码文件。你使用的编译器应支持目前 C 语言国际标准(ISO/IEC 9899:2011，也被称为 C11)。你还需要一个用于创建和修改代码的编辑器，可以采用纯文本编辑器(如记事本(Notepad)或 vi)创建源文件。不过，采用专为编辑 C 语言代码设计的编辑器会更有帮助。

以下是作者推荐的两款 C 语言编译器，均为免费软件：

- GNU C 编译器，GCC，可从 http://www.gnu.org 下载，它支持多种不同的操作系统环境。

- 面向 Microsoft Windows 的 Pelles C 编译器，可从 http://www.smorgasbordet.com/pellesc/下载，它提供了一个非常棒的集成开发环境(IDE)。

本书采用的约定

本书的文本和布局采用了许多不同的样式，以便区分各种不同的信息。大多数样式表达的含义都很明显。程序代码样式如下：

```
int main(void)
{ printf("Beginning C\n");
  return 0;
}
```

如果代码片段是从前面的实例修改而来，修改过的代码行就用粗体显示，如下所示：

```
i int main(void)
{
  printf("Beginning C by Ivor Horton\n");
   return 0;
}
```

当代码出现在文本中时，它的样式会有所不同，如：double。

程序代码中还是用了各种“括号”。它们之间的差别非常重要，不同称呼。本书中称()为圆括号，{}为大括号，[]为方括号。

目　　录

译 者 序

C 语言是经典的编程语言，凭借其自身简洁、灵活和功能强大等特点，自诞生以来就牢牢占据着流行编程语言的榜首。C 语言是强大的编程语言，可以进行从操作系统到设备驱动的各层次开发，可以应用在从大规模传统应用到新兴移动应用的各种领域。C 语言也是一门非常优秀的学习程序设计入门语言，其简洁性使得初学者无须学习太多语法就可以开始编写真正的应用程序，很多程序员的职业生涯就是从握手 C 语言开始的。

对于初学者来说，目前市面上介绍 C 语言入门的书籍太多了，可谓车载斗量，浩若繁星。《C 语言入门经典》则是众多 C 语言学习资源中的经典作品。本书的作者是世界著名的计算机图书大师 Ivor Horton。Ivor Horton 在 IBM 工作多年，具有丰富的实践经验，其著作曾帮助无数程序员步入编程的殿堂。《C 语言入门经典》作为 Ivor Horton 的经典之作，一版再版，培养了一代又一代的程序员，对 C 语言的推广可谓是功不可没。

编程语言的学习十分枯燥，学习的过程也特别艰辛，但是学成之后所获得的成就感也是无与伦比的。C 语言的灵活性在很大程度上得益于指针，而指针也往往是让初学者头疼的地方，甚至成为一个梦魇。我深知其学习的艰辛，对于自己在学习过程中的彷徨犹豫，挫折困顿，至今还历历在目。《C 语言入门经典》循序渐进、深入浅出的讲解，曾经让自己大惑不解的地方，从本书看来则是如此的理所当然，水到渠成。恨自己没在初学之时，早点读到此书。

本书在前一版的基础上，最大的变化就是增加了一章：高级应用专题。其中介绍了 Unicode 字符，可移植性的专用整数类型、复数类型以及线程编程。而线程章节的出现极大地完善了前几版的 C 语言体系，可谓是一个巨大惊喜。此外，本书在总结前几版的基础上对章节进行了更精确细微的调整，使内容在逻辑上更加合理，读起来更加流畅，更符合阅读习惯。开篇增加了对 C 语言，标准库的介绍。在第 9 章中增加了结束程序的详细讲解。第 10 章基本的输入和输出操作中，使用新的 scanf_s()，printf_s()代替了旧版的 scanf()和 printf()。在第 12 章中，引入了新的 freopen_s()函数以及缓存文件的操作。另外还有一些细微的调整与删除，使得本书更加紧凑与完美。

纸上得来终觉浅，绝知此事要躬行。本书的作者深谙此理，在每章都有练习和习题供读者动手实践。学习一门语言没有比动手实践更快、更好的方法了。所以建议读者在每读完一章的时候亲自动手完成练习，而不是“读”过练习，如此方能成为理论和行动上的“巨人”。有人问大师，如何能技近乎道？大师曰：读书，读好书，然后实践之。万事无他，惟手熟尔！

在这里要感谢清华大学出版社的李阳和于平编辑，她们为本书的翻译投入了巨大的热情并付出了很多心血。没有你们的帮助和鼓励，本书不可能顺利付梓。

对于这本经典之作，译者本着“诚惶诚恐”的态度，在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。感激不尽！本书全部章节由杨浩翻译，参与翻译活动的还有孔祥亮、陈跃华、杜思明、熊晓磊、曹汉鸣、陶晓云、王通、方峻、李小凤、曹晓松、蒋晓冬、邱培强、洪妍、李亮辉、高娟妮、曹小震、陈笑。

最后，希望读者通过阅读本书能早日步入 C 语言编程的殿堂，领略 C 语言之美！

第 1 章

C 语言编程

C 语言是一种功能强大、简洁的计算机语言，通过它可以编写程序，指挥计算机完成指定的任务。我们可以利用 C 语言创建程序(即一组指令)，并让计算机依指令行事。

用 C 语言编程并不难，本书将用浅显易懂的方法介绍 C 语言的基础知识，读完本章，读者就可以编写第一个 C 语言程序了，其实 C 语言很简单。

本章的主要内容：

- C 语言标准
- 标准库的概念
- 如何创建 C 程序
- 如何组织 C 程序
- 如何编写在屏幕上显示文本的程序

1.1 C 语言

C 是相当灵活的，用于执行计算机程序能完成的几乎所有任务，包括会计应用程序、字处理程序、游戏、操作系统等。它不仅是更高级语言(如 C++)的基础，目前还以 Objective C 的形式开发手机应用程序。Objective C 是标准的 C 加上一小部分面向对象编程功能。C 很容易学习，因为它很简洁。因此，如果你立志成为一名程序员，最好从 C 语言开始学起，能快速而方便地获得编写实际应用程序的足够知识。

C 语言由一个国际标准定义，目前，其最新版本由 ISO/IEC 9899:2011 文档定义。当前的标准一般称为 C11，本书介绍的语言遵循 C11 标准。但要知道，C11 定义的一些语言元素是可选的。这表示，遵循 C11 标准的 C 编译器可能没有实现该标准中的所有功能。(编译器只是一个程序，它可以把用我们能理解的术语所编写的程序转换为计算机能理解的术语)。本书会标识出 C11 中的可选语言特性，这样读者就知道，自己的编译器可能不支持它。

C11 编译器还有可能没有实现 C11 标准强制的所有语言特性。实现新语言功能是需要时间的，所以编译器开发人员常常采用逐步接近的方式实现它们。这也是程序可能不工作的另一个原因。尽管如此，根据我的经验，C 程序不能工作的最常见原因，至少有 99.9%的可能性是出现了错误。

1.2 标准库

C 的标准库也在 C11 标准中指定。标准库定义了编写 C 程序时常常需要的常量、符号和函数。它还提供了基本 C 语言的一些可选扩展。取决于机器的特性，例如计算机的输入输出，由标准库以不依赖机器的形式实现。这意味着，在 PC 中用 C 代码把数据写入磁盘文件的方式，与在其他计算机上相同，尽管底层的硬件处理相当不同。库提供的标准功能包括大多数程序员都可能需要的功能，例如处理文本字符串或数学计算，这样就免除了自己实现这些功能所需的大量精力。

标准库在一系列标准文件——头文件中指定。头文件的扩展名总是.h。为了使一组标准功能可用于 C 程序文件，只需要将对应的标准头文件包含进来，其方式在本章后面介绍。我们编写的每个程序都会用到标准库。附录 E 汇总了构成标准库的头文件。

1.3 学习 C

如果对编程非常陌生，则不需要学习 C 的某些方面，至少在刚开始时不需要学习。这些功能比较特殊，或者不大常用。本书把它们放在第 14 章，这里读者可以在熟悉其他内容后，再学习它们。

所有示例的代码都可以从 Apress 网站(http://www.apress/com)上下载，但建议读者自己输入本书中的所有示例，即使它们非常简单，也要输入。自己亲自输入，以后就不容易忘记。不要害怕用代码进行实验。犯错对编程而言非常有教育性。早期犯的错误越多，学到的东西就越多。

1.4 创建 C 程序

C 程序的创建过程有 4 个基本步骤或过程：

- 编辑
- 编译
- 链接
- 执行

这些过程很容易完成(就像翻转手掌一样简单，而且可以随时翻转)，首先介绍每个过程，以及它们对创建 C 程序的作用。

1.4.1 编辑

编辑过程就是创建和修改 C 程序的源代码——我们编写的程序指令称为源代码。有些 C 编译器带一个编辑器，可帮助管理程序。通常，编辑器是提供了编写、管理、开发与测试程序的环境，有时也称为集成开发环境(Integrated Development Environment，IDE)。

也可以用一般的文本编辑器来创建源文件，但它们必须将代码保存为纯文本，而没有嵌入附加的格式化数据。不要使用字处理器(例如微软的 Word)，字处理器不适合编写程序代码，因为它们在保存文本时，会附加一些格式化信息。一般来说，如果编译器系统带有编辑器，就会提供很多更便于编写及组织程序的功能。它们通常会自动编排程序文本的格式，并将重要的语言元素以高亮颜色显示，这样不仅让程序容易阅读，还容易找到单词输入错误。

在 Linux 上，最常用的文本编辑器是 Vim 编辑器，也可以使用 GNU Emacs 编辑器。对于 Microsoft Windows，可以使用许多免费(freeware)或共享(shareware)的程序设计编辑器。这些软件提供了许多功能，例如，高亮显示特殊的语法及代码自动缩进等功能，帮助确保代码是正确的。Emacs 编辑器也有 Microsoft Windows 版本。UNIX 环境的 Vi 和 VIM 编辑器也可用于 Windows，甚至可以使用 Notepad++(http://notepad-plus-plus.org/)。

当然，也可以购买支持 C 语言的专业编程开发环境，例如微软或 Borland 的相关产品，它们能大大提高代码编辑能力。不过，在付款之前，最好检查一下它们支持的 C 级别是否符合当前的 C 语言标准 C11。因为现在很多编辑器产品主要面向 C++开发人员，C 语言只是一个次要目标。

1.4.2 编译

编译器可以将源代码转换成机器语言，在编译的过程中，会找出并报告错误。这个阶段的输入是在编辑期间产生的文件，常称为源文件。

编译器能找出程序中很多无效或无法识别的错误，以及结构错误，例如程序的某部分永远不会执行。编译器的输出结果称为对象代码(object code)，存放它们的文件称为对象文件(object file)，这些文件的扩展名在 Microsoft Windows 环境中通常是.obj，在 Linux/UNIX 环境中通常是.o。编译器可以在转换过程中找出几种不同类型的错误，它们大都会阻止对象文件的创建。

如果编译成功，就会生成一个文件，它与源文件同名，但扩展名是.o 或者.obj。

如果在 UNIX 系统下工作，在命令行上编译 C 程序的标准命令是 cc(若编译器是 GNU's Not UNIX(GNU)，则命令为.gcc)。下面是一个示例：

```
cc -c myprog.c
```

其中，myporg.c 是要编译的程序，如果省略了 - c 这个参数，程序还会自动链接。成功编译的结果是生成一个对象文件。

大多数 C 编译器都有标准的编译选项，在命令行(如 cc myprog.c)或集成开发环境下的菜单选项(Compile 菜单选项)里都可找到。在 IDE 中编译常常比使用命令行容易得多。

编译过程包括两个阶段。第一个阶段称为预处理阶段，在此期间会修改或添加代码，第二个阶段是生成对象代码的实际编译过程。源文件可以包含预处理宏，它们用于添加或修改 C 程序语句。如果现在不理解它们，不必担心，本书后面将进行详细论述。

1.4.3　链接

链接器(linker)将源代码文件中由编译器产生的各种对象模块组合起来，再从 C 语言提供的程序库中添加必要的代码模块，将它们组合成一个可执行的文件。链接器也可以检测和报告错误，例如，遗漏了程序的某个部分，或者引用了一个根本不存在的库组件。

实际上，如果程序太大，可将其拆成几个源代码文件，再用链接器连接起来。因为很难一次编写一个很大的程序，也不可能只使用一个文件。如果将它拆成多个小源文件，每个源文件提供程序的一部分功能，程序的开发就容易多了。这些源文件可以分别编译，更容易避免简单输入错误的发生。再者，整个程序可以一点一点地开发，组成程序的源文件通常会用同一个项目名称集成，这个项目名称用于引用整个程序。

程序库提供的例程可以执行非 C 语言的操作，从而支持和扩展了 C 语言。例如，库中包含的例程支持输入、输出、计算平方根、比较两个字符串，或读取日期和时间信息等操作。

链接阶段出现错误，意味着必须重新编辑源代码；反过来，如果链接成功，就会产生一个可执行文件，但这并不一定表示程序能正常工作。在 Microsoft Windows 环境下，这个可执行文件的扩展名为.exe；在 UNIX 环境下，没有扩展名，但它是一个可执行的文件类型。多数 IDE 也有 Build(建立)选项，它可一次完成程序的编译和链接。

1.4.4　执行

执行阶段就是当成功完成了前述 3 个过程后，运行程序。但是，这个阶段可能会出现各种错误，包括输出错误及什么也不做，甚至使计算机崩溃。不管出现哪种情况，都必须返回编辑阶段，检查并修改源代码。

在这个阶段，计算机最终会精确地执行指令。在 UNIX 和 Linux 下，只要键入编译和链接后的文件名，即可执行程序。在大多数 IDE 中，都有一个相应的菜单命令来运行或者执行已编译的程序。这个 Run 命令或者 Execute 命令可能有自己的菜单，也可能位于 Compile 菜单项下。在 Windows 环境中，运行程序的.exe 文件即可，这与运行其他可执行程序一样。

在任何环境及任何语言中，开发程序的编辑、编译、链接与执行这 4 个步骤都是一样的。图 1-1 总结了创建 C 程序的各个过程。

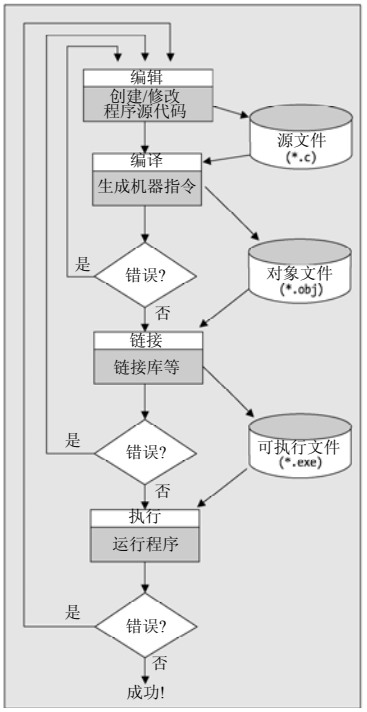

图 1-1　创建和执行程序

1.5 创建第一个程序

本节先浏览一下创建 C 语言程序的流程，从输入代码到执行程序的所有 4 个步骤。在这个阶段，若不了解所键入的代码信息，别担心，笔者会解释每一个步骤。

试试看：C 程序示例

打开编辑器，输入下面的程序，请注意标点符号不要输错，第 4 行及最后一行的括号是大括号{}，而不是方括号[]或者圆括号()——这很重要。另外一定要键入斜杠(/)，以后也会用到反斜杠(\)。最后别忘了行末的分号(;)。

```
/*  Program 1.1 Your Very First C Program - Displaying Hello World */
#include  <stdio.h>

int main(void)
{
  printf("Hello  world! ");
  return 0;
}
```

在输入了上面的源代码后，将程序保存为 hello.c。可以用任意名字替代 hello，但扩展名必须是.c。这个扩展名在编写 C 程序时是一个通用约定，它表示文件的内容是 C 语言源代码。大多数 C 编译器都要求源文件的扩展名是.c，否则编译器会拒绝处理它。

下面编译程序(如本章前面“编译”一节所述)，链接所有必要的内容，创建一个可执行程序(如本章前面“链接”一节所述)。编译和链接一般在一个操作中完成，通常称为“构建操作”。源代码编译成功后，链接器就添加程序需要的标准库代码，为程序创建一个可执行文件。

最后，执行程序。这有几种方式，在 Windows 环境下，一般只需要在 Windows Explorer 中双击.exe 文件，但最好打开一个命令行窗口，输入执行它的命令，因为在程序执行完毕后，显示输出的窗口就会消失。在所有的操作系统环境上，都可以从命令行上运行程序。只需要启动一个命令行会话，把当前目录改为包含程序可执行文件的目录，再输入程序名，就可以执行它了。

如果没有出现错误，就大功告成了。这个程序会在屏幕上输出如下信息：

```
Hello world!
```

1.6 编辑第一个程序

我们可以修改程序，在屏幕上输出其他信息，例如可以将程序改成：

```
/*Program 1.2 Your Second C Program */
#include <stdio.h>

int main(void)
```

```
{
  printf("\"If at first you don't succeed, try, try, try again!\"");
  return 0;
}
```

这个版本的输出是：

```
"If at first you don't succeed, try, try, try again! "
```

在要显示的文本中，\”序列称为转义序列(escape sequence)。文本中包含几个不同的转义序列。\”是在文本中包含双引号的特殊方式，因为双引号通常表示字符串的开头和结尾。转义序列\”使双引号出现在输出的开头和结尾。如果不使用转义序列，不仅双引号不会出现在输出中，而且程序不会编译。本章后面的“控制字符”一节将详细介绍转义序列。

修改完源代码后，可以重新编译，链接后执行。反复练习，熟悉整个流程。

1.7 处理错误

犯错乃人之常情，没什么难为情的。幸好计算机一般不会出错，而且非常擅长于找出我们犯的错误。编译器会列出在源代码中找到的一组错误信息(甚至比我们想象的多)，通常会指出有错误的语句。此时，我们必须返回编辑阶段，找出有错误的代码并更正。

有时一个错误会使后面本来正确的语句也出现错误。这多半是程序的其他部分引用了错误语句定义的内容所造成的。当然，定义语句有错，但被定义的内容不一定有错。

下面看看源代码在程序中生成了一个错误时，会是什么样的情况。编辑第二个程序示例，将 printf()行最后的分号去掉，如下所示：

```
/*Program 1.2 Your Second C Program */
#include <stdio.h)

int main(void)
{
  printf("\"If at first you don't succeed, try, try, try again!\"")
  return 0;
}
```

编译这个程序后，会看到错误信息，具体信息随编译器的不同而略有区别。下面是一个比较常见的错误信息：

```
Syntax error : missing ';' before '}'
HELLO.C - 1 error(s), 0 warning(s)
```

编译器能精确地指出错误及其出处，在这里，printf()行的结尾处需要一个分号。在开始编写程序时，可能有很多错误是简单的拼写错误造成的。还很容易忘了逗号、括号，或按错了键。没关系，许多有经验的老手也常犯这种错误。

如前所述，有时一点小错误会造成大灾难，编译器会显示许多不同的错误信息。不要被错误的数量吓倒，仔细看过每一个错误信息后，返回并改掉错误部分，不懂的先不

管它，然后再编译一次源文件，就会发现错误一次比一次少。

返回编辑器，重新输入分号，再编译，看看有没有其他错误，如果没有错误，程序就可以执行了。

1.8 剖析一个简单的程序

编写并编译了第一个程序后，下面是另一个非常类似的例子，了解各行代码的作用：

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
#include  <stdio.h>

int  main(void)
{
  printf("Beware  the  Ides  Of  March!");
  return 0;
}
```

这和第一个程序完全相同，这里把它作为练习，用编辑器输入这个示例，编译并执行。若输入完全正确，会看到如下输出：

```
Beware  the  Ides  Of  March!
```

1.8.1 注释

上述示例的第一行代码如下：

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
```

这不是程序代码，因为它没有告诉电脑执行操作，它只是一个注释，告诉阅读代码的人，这个程序要做什么。位于/*和*/之间的任意文本都是注释。只要编译器在源文件中找到/*，就忽略它后面的内容(即使其中的文本很像程序代码)，一直到表示注释结束的*/为止。/*可以和*/放在同一行代码上，也可以放在不同的代码行上。如果忘记包含对应的*/，编译器就会忽略/*后面的所有内容。下面使用一个注释说明代码的作者及版权所有：

```
/*
 * Written by Ivor Horton
 * Copyright 2012
*/
```

也可以修饰注释，使它们比较突出：

```
/*****************************************
 * This is a very important comment      *
 * so please read this.                  *
 *****************************************/
```

使用另一种记号，可以在代码行的末尾添加一个注释，如下所示：

```
printf("Beware the Ides of March!");  // This line displays a quotation
```

代码行上两个斜杠后面的所有内容都会被编译器忽略。这种形式的注释没有前一种记号那么凌乱，尤其是在注释只占一行的情形下。

应养成给程序添加注释的习惯，当然程序也可以没有注释，但在编写较长的程序时，可能会忘记这个程序的作用或工作方式。添加足够的注释，可确保日后自己(和其他程序员)能理解程序的作用和工作方式。

下面给程序再添加一些注释：

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
#include <stdio.h>                  // This is a preprocessor directive

int main(void)                      // This identifies the function main()
{                                   // This marks the beginning of main()
  printf("Beware the Ides of March!"); // This line outputs a quotation
  return 0;                         // This returns control to the operating system
}                                   // This marks the end of main()
```

可以看出，使用注释是一种非常有效的方式，可以解释程序中要发生的事情。注释可以放在程序中的任意位置，说明代码的一般作用，指定代码是如何工作的。

1.8.2 预处理指令

下面的代码行：

```
#include <stdio.h>              // This is a preprocessor directive
```

严格说来，它不是可执行程序的一部分，但它很重要，事实上程序没有它是不执行的。符号#表示这是一个预处理指令(preprocessing directive)，告诉编译器在编译源代码之前，要先执行一些操作。编译器在编译过程开始之前的预处理阶段处理这些指令。预处理指令相当多，大多放于程序源文件的开头。

在这个例子中，编译器要将 stdio.h 文件的内容包含进来，这个文件称为头文件(header file)，因为它通常放在程序的开头处。在本例中，头文件定义了 C 标准库中一些函数的信息，但一般情况下，头文件指定的信息应由编译器用于在程序中集成预定义函数或其他全局对象，所以有时需要创建自己的头文件，以用于程序。本例要用到标准库中的 printf()函数，所以必须包含 stdio.h 头文件。stdio.h 头文件包含了编译器理解 printf()以及其他输入/输出函数所需要的信息。名称 stdio 是标准输入/输出(standard input/output)的缩写。C 语言中所有头文件的扩展名都是.h，本书的后面会用到其他头文件。

注意：

在一些系统中，头文件名是不区分大小写的，但在#include 指令里，这些文件名通常是小写。

每个符合 C11 标准的 C 编译器都有一些标准的头文件。这些头文件主要包含了与 C 标准库函数相关的声明。所有符合该标准的 C 编译器都支持同一组标准库函数，有同一

组标准头文件，但一些编译器有额外的库函数，它们提供的功能一般是运行编译器的计算机所专用的。

注意：
附录 E 列出了所有的标准头文件。

1.8.3 定义 main()函数

下面的 5 行指令定义了 main()函数：

```
int main(void)                          // This identifies the function main()
{                                       // This marks the beginning of main()
  printf("Beware the Ides of March!");  // This line outputs a quotation
  return 0;                             // This returns control to the operating system
}                                       // This marks the end of main()
```

函数是两个括号之间执行某组操作的一段代码。每个 C 程序都由一个或多个函数组成，每个 C 程序都必须有一个 main()函数——因为每个程序总是从这个函数开始执行。因此假定创建、编译、链接了一个名为 progname.exe 的文件。执行它时，操作系统会执行这个程序的 main()函数。

定义 main()函数的第一行代码如下：

```
int main(void)            // This identifies the function main()
```

它定义了 main()函数的起始，注意这行代码的末尾没有分号。定义 main()函数的第一行代码开头是一个关键字 int，它表示 main()函数的返回值的类型，关键字 int 表示 main()函数返回一个整数值。执行完 main()函数后返回的整数值表示返回给操作系统的一个代码，它表示程序的状态。在下面的语句中，指定了执行完 main()函数后要返回的值：

```
return 0;                 // This returns control to the operating system
```

这个 return 语句结束 main()函数的执行，把值 0 返回给操作系统。从 main()函数返回 0 表示，程序正常终止，而返回非 0 值表示异常。换言之，在程序结束时，发生了不应发生的事情。

紧跟在函数名 main 后的括号，带有函数 main()开始执行时传递给它的信息，在这个例子里，括号内是 void，表示没有给函数 main()传递任何数据，后面会介绍如何将数据传递给函数 main()或程序内的其他函数。

函数 main()可以调用其他函数，这些函数又可以调用其他函数。对于每个被调用的函数，都可以在函数名后面的括号中给函数传递一些信息。在执行到函数体中的 return 语句时，就停止执行该函数，将控制权返回给调用函数(对于函数 main()，则将控制权返回给操作系统)。一般函数会定义为有返回值或没有返回值。函数返回一个值时，该值总是特定的类型。对于函数 main()，返回值的类型是 int，即整数。

1.8.4 关键字

在 C 语言中，关键字是有特殊意义的字，所以在程序中不能将关键字用于其他目的。关键字也称为保留字。在前面的例子里，int 就是一个关键字，void 和 return 也是关键字。C 语言有许多关键字，我们在学习 C 语言的过程中，将逐渐熟悉这些关键字。附录 C 列出了完整的 C 语言关键字表。

1.8.5 函数体

main()函数的一般结构如图 l-2 所示：

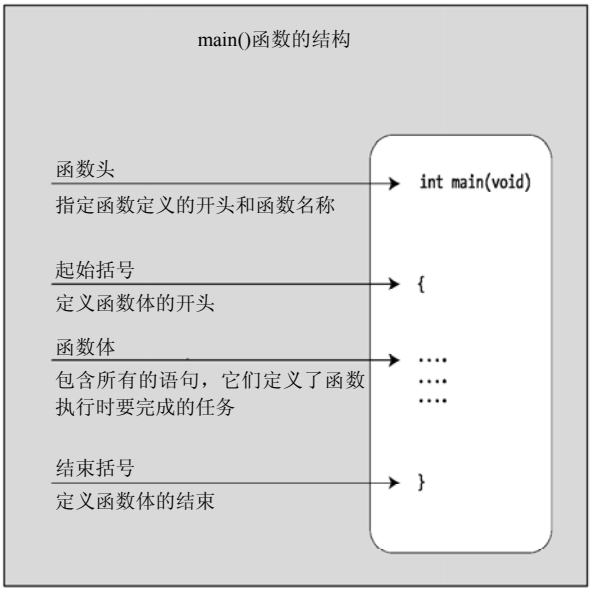

图 l-2 函数 main()的结构

函数体是在函数名称后面位于起始及结束两个大括号之间的代码块。它包含了定义函数功能的所有语句。这个例子的 main()函数体非常简单，只有两个语句：

```
{                                       // This marks the beginning of main()
  printf("Beware the Ides of March!");  // This line outputs a quotation
  return 0;                          // This returns control to the operating system
}                                       // This marks the end of main()
```

每个函数都必须有函数体，但函数体可以是空的，仅有起始及结束两个大括号，里面没有任何语句，在这种情况下，这个函数什么也不做。

这样的函数有什么用？事实上，在开发一个包含很多函数的程序时，这种函数是非常有用的。我们可以声明一些用来解决手头问题的空函数，确定需要完成的编程工作，再为每个函数创建程序代码。这个方法有助于条理分明地、系统地建立程序。

注意：

程序 1.3 将大括号单独排为一行，并缩进大括号之间的代码。这么做可清楚地表示括号框起来的语句块从哪里起始和结束。大括号之间的语句通常缩进两个或多个空格，使大括号突出在前。这是个很好的编程格式，可以使语句块更容易阅读。

代码中的大括号可以用其他方式摆放。例如：

```
int main(void) {
  printf("Beware the Ides of March!");   // This line outputs a quotation
  return 0;
}
```

提示：

无论源代码采用什么方式摆放，都要一直采用这种方式，这很重要。

1.8.6 输出信息

例子中的 main()函数体包含了一个调用 printf()函数的语句：

```
printf("Beware the Ides of March!");    // This line outputs a quotation
```

printf()是一个标准的库函数，它将函数名后面引号内的信息输出到命令行上(实际上是标准输出流，默认为命令行)。在这个例子中，调用这个函数会显示双引号内的一段警示语：双引号内的字符串称为字符串字面量。注意这行代码用分号作为结尾。

1.8.7 参数

包含在函数名(如上面语句中的 printf()函数)后的圆括号内的项称为参数，它指定要传送给函数的数据。当传送给函数的参数多于一个时，要用逗号分开。

在上面的例子中，函数的参数是双引号内的文本字符串。如果不喜欢例子中引号内的文本，可以改用自己想输出的句子。例如，使用如下语句：

```
printf("Out, damned Spot! Out I say!");
```

修改源代码后，必须再次编译及链接程序，才可执行。

注意：

与 C 语言中所有可执行的语句一样，printf()行的末尾必须有分号(这与定义语句或指令语句不同)。这是一个很容易犯的错误，尤其是初次使用 C 编程的人，老是忘了分号。

1.8.8 控制符

前面的程序可以改为输出两段句子。输入以下的代码：

```
// Program 1.4 Another Simple C Program - Displaying a Quotation
#include <stdio.h>
```

```
int main(void)
{
  printf("My formula for success?\nRise early, work late, strike oil.\n");
  return 0;
}
```

输出的结果是：

```
My formula for success?
Rise early, work late, strike oil.
```

在 printf()语句中，在文本的开头和第一句的后面，增加了字符\n，它是另一个转义序列，代表换行符。这样输出光标就会移动到下一行，后续的输出就会显示在新行上。

反斜杠(\)在文本字符串里有特殊的意义，它表示转义序列的开始。反斜杠后面的字符表示是哪种转义序列。对于\n，n 表示换行。还有其他许多转义序列。显然，反斜杠是有特殊意义的，所以需要一种方式在字符串中指定反斜杠。为此，应使用两个反斜杠(\\)。

输入以下的程序：

```
// Program 1.5 Another Simple C Program - Displaying Great Quotations
#include <stdio.h>

int main(void)
{
  printf("\"It is a wise father that knows his own child.\"\nShakespeare\n");
  return 0;
}
```

输出的结果如下：

```
"It is a wise father that knows his own child."
Shakespeare
```

输出中包含双引号，因为在字符串中使用了双引号的转义序列。Shakespeare 显示在下一行，因为在\”的后面有\n 转义序列。

在输出字符串中使用转义序列\a 可以发出声音，说明发生了有趣或重要的事情。输入以下的程序并执行：

```
// Program 1.6 A Simple C Program – Important
#include <stdio.h>

int main(void)
{
  printf("Be careful!!\n\a");
  return 0;
}
```

这个程序的输出如下所示且带有声音。仔细聆听，电脑的扬声器会发出鸣响。

```
Be  careful!!
```

转义序列\a 表示发出鸣响。表 1-1 是转义序列表。

表 1-1 转义序列

转 义 序 列	说 明
\n	换行
\r	回车键
\b	退后一格
\f	换页
\t	水平制表符
\v	垂直制表符
\a	发出鸣响
\?	插入问号(?)
\"	插入双引号(")
\'	插入单引号(')
\\	插入反斜杠(\)

试着在屏幕上显示多行文本，在该文本中插入空格。使用 \n 可以把文本放在多个行上，使用\t 可以给文本加上空格。本书将大量使用这些转义序列。

1.8.9 三字母序列

一般可以直接在字符串中使用问号。\?转义序列存在的唯一原因是，有 9 个特殊的字母序列，称为三字母序列，这是包含三个字母的序列，分别表示#、[、]、\、^、~、\、{和}：

??=转换为#	??(转换为[	??)转换为]
??/转换为\	??<转换为{	??>转换为}
??'转换为^	??!转换为|	??-转换为~

在 International Organization for Standardization(ISO)不变的代码集中编写 C 代码时，就需要它们，因为它没有这些字符。这可能不适用于你。可以完全不理会它们，除非希望编写如下语句：

```
printf("What??!\n");
```

这个语句生成的输出如下：

```
What|
```

三字母序列??!会转换为|。为了获得希望的输出，需要把上述语句写成：

```
printf("What?\?!\n");
```

现在三字母序列不会出现，因为第二个问号用其转义序列指定。使用三字母序列时，编译器会发出一个警告，因为通常是不应使用三字母序列的。

1.9 预处理器

上述示例介绍了如何使用预处理指令，把头文件的内容包含到源文件中。编译的预处理阶段可以做的工作远不止此。除了指令之外，源文件还可以包含宏。宏是提供给预处理器的指令，来添加或修改程序中的 C 语句。宏可以很简单，只定义一个符号，例如 INCHES_PER_FOOT，只要出现这个符号，就用 12 替代。其指令如下：

```
#define INCHES_PER_FOOT 12
```

在源文件中包含这个指令，则代码中只要出现 INCHES_PER_FOOT，就用 12 替代它。例如：

```
printf("There are %d inches in a foot.\n", INCHES_PER_FOOT);
```

预处理后，这个语句变成：

```
printf("There are %d inches in a foot.\n", 12);
```

INCHES_PER_FOOT 不再出现，因为该符号被#define 指令中指定的字符串替代。对于源文件中的每个符号实例，都会执行这个替代。

宏也可以很复杂，根据特定的条件把大量代码添加到源文件中。这里不进一步介绍。第 13 章将详细讨论预处理器宏。在此之前我们会遇到一些宏，那时会解释它们。

1.10 用 C 语言开发程序

如果读者从未写过程序，对 C 语言开发程序的过程就不会很清楚，但它和我们日常生活的许多事务是相同的，万事开头难。一般首先大致确定要实现的目标，接着把该目标转变成比较准确的规范。有了这个规范后，就可以制订达到最终目标的一系列步骤了。就好比光知道要盖房子是不够的，还得知道需要盖什么样的房子，它有多大，用什么材料，要盖在哪里。这种详细规划也需要运用到编写程序上。下面介绍编写程序时需要完成的基本步骤。房子的比喻是很有帮助的，因此就利用这个比喻。

1.10.1 了解问题

第一步是弄清楚要做什么。在不清楚应提供什么设施：多少间卧房、多少间浴室、各房间多大等等之前就开始建造房子，会有不知所措之感。所有这些都会影响建造房子所需的材料和工作量，从而影响整个房子的成本。一般来说，在满足需求和完成项目的有限资金、人力及时间之间总会达成某种一致。

这和开发一个任意规模的程序是相同的。即使是很简单的问题，也必须知道有什么输入，对输入该做什么处理，要输出什么，以及输出哪种格式。输入可以来自键盘，也可以来自磁盘文件的数据，或来自电话或网络的信息。输出可以显示在屏幕上，或打印

出来，也可以是更新磁盘上的数据文件。

对于较复杂的程序，需要多了解程序的各个方面。清楚地定义程序要解决的问题，对于理解制订最终方案所需的资源与努力，是绝对必要的一部分。好好考虑这些细节，还可以确定项目是否切实可行。对于新项目缺乏精准、详细的规范，常常使项目所花的时间和资金大大超出预算，因而中断项目的例子有很多。

1.10.2 详细设计

要建造房子，必须有详细的计划。这些计划能让建筑工人按图施工，并详细描述房子如何建造——具体的尺寸、要使用的材料等。还需要确定何时完成什么工作。例如，在砌墙之前要先挖地基，所以这个计划必须把工作分为可管理的单元，以便执行起来井然有序。

写程序也是一样。首先将程序分解成许多定义清楚且互相独立的小单元，描述这些独立单元相互沟通的方式，以及每个单元在执行时需要什么信息，从而开发出富有逻辑、相互独立的单元。把大型程序编写为一个大单元肯定是不可行的。

1.10.3 实施

有了房子的详细设计，就可以开始工作了。每组建筑工人必须按照进度完成他们的工作。在下一阶段开始前，必须先检查每个阶段是否正确完成。省略了这些检查，将可能导致整栋房子倒塌。

当然，假使程序很大，可以一次编写一部分。一个部分完成后，再写下一部分。每个部分都要基于详细的设计规范，在进行下一个部分之前，应尽可能详细地检查每个部分的功能。这样，程序就会逐步完成预期的任务。

大型编程项目常常涉及一组程序员。项目应分成相当独立的单元，分配给程序员组中的各个成员。这样就可以同时开发几个代码单元。如果代码单元要相互连接为一个整体，就必须精确定义代码单元与程序其余部分之间的交互。

1.10.4 测试

房子完成了，还要进行许多测试：排水设备、水电设施、暖气等。任何部分都有可能出问题，这些问题必须解决。这有时是一个反复的过程，一个地方的问题可能会造成其他地方出问题。

这个机制与写程序是类似的。每个程序模块——组成程序的单元——都需要单独测试。若它们工作不正常，就必须调试。调试(Debugging)是一个找出程序中的问题及更正错误的过程。调试的由来有个说法，曾经有人在查找程序的错误时，使用计算机的电路图来跟踪信息的来源及其处理方式，竟然发现计算机程序出现错误，是因为一只虫子在电脑里，让里面的线路短路而发生的，后来，bug 这个词就成了程序错误的代名词。

对于简单的程序，通常只要检查代码，就可以找出错误。然而一般来说，调试过程

通常会使用调试器临时插入一些代码，确定在出错时会发生什么。这包括插入断点，当暂停执行，检查代码中的值。还可以单步执行代码。如果没有调试器，就要加入额外的程序代码，输出一些信息，来确定程序中事件的发生顺序，以及程序执行时生成的中间值。在大型的程序里，还需要联合测试各个程序模块，因为各个模块或许能正常工作，但并不保证它能和其他模块一起正常工作。在程序开发的这个阶段，有个专业术语叫集成测试(integration testing)。

1.11 函数及模块化编程

到目前为止，“函数”这个词已出现过好几次了，如 main()、printf()、函数体等。下面将深入研究函数是什么，为什么它们那么重要。

大多数编程语言(包含 C 语言)都提供了一种方法，将程序切割成多个段，各段都可以独立编写。在 C 语言中，这些段称为函数。一个函数的程序代码与其他函数是相互隔绝的。函数与外界有一个特殊的接口，可将信息传进来，也可将函数产生的结果传出去。这个接口在函数的第一行即在函数名的地方指定。

图 1-3 的简单程序例子由 4 个函数组成，用于分析棒球分数。

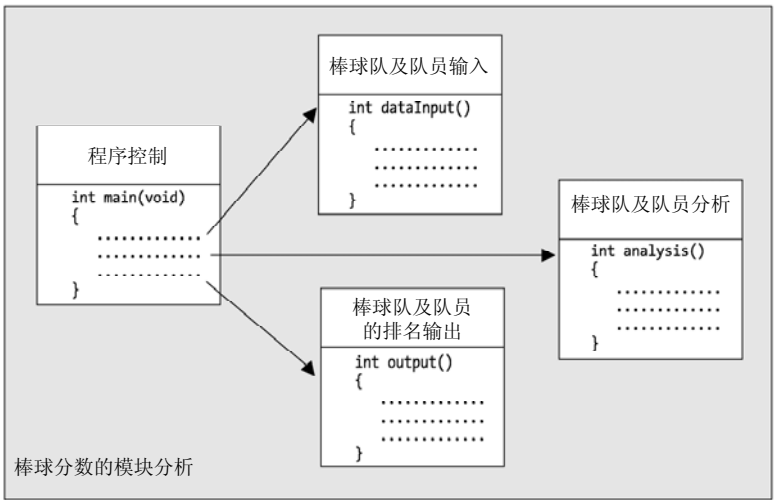

图 1-3 模块化编程

这 4 个函数都完成一个指定的、定义明确的工作。程序中操作的执行由一个模块 main()总体掌控。一个函数负责读入及检查输入数据，另一个函数进行分析。读入及分析了数据后，第 4 个函数就输出球队及球员的排名。

将程序分割成多个易于管理的小单元，对编程是非常重要的，其理由如下：

- 可以单独编写和测试每个函数，大大简化了使整个程序运转起来的过程。
- 几个独立的小函数比一个大函数更容易处理和理解。
- 库就是供人使用的函数集。因为它们是事先写好，且经过测试，能正常工作，所以可以放心地使用，无须细究它的代码细节。这就加快了开发程序的速度，因为

我们只需要关注自己的代码，这是 C 语言的一个基本组成部分。C 语言中丰富的函数库大大增强了 C 语言的能力。

- 也可以编写自己的函数库，应用于自己感兴趣的程序类型。如果发现经常编写某个函数，就可以编写它的通用版本，以满足自己的需求，并将它加入自己的库中。以后需要用到这个函数时，就可使用它的库版本了。
- 在开发包含几千到几百万行代码的大型程序时，可以由一些程序设计团队来进行，每个团队负责一个指定的函数子组，最后把它们组成完整的程序。

第 8 章将详细介绍 C 函数。C 程序的结构在本质上就是函数的结构，本章的第一个例子就用到一个标准的库函数 printf()。

注意：

在其他一些编程语言中，用术语“方法”表示自包含的代码单元。因此方法的含义与函数相同。

试试看：将所学的知识用于实践

下面的例子将前面学到的知识用于实践。首先，看看下面的代码，检查自己是否理解它的作用。然后输入这些代码，编译、链接并执行，看看会发生什么。

```
// Program 1.7 A longer program
#include <stdio.h>              // Include the header file for input and output

int main(void)
{
  printf("Hi there!\n\n\nThis program is a bit");
  printf(" longer than the others.");
  printf("\nBut really it's only more text.\n\n\n\a\a");
  printf("Hey, wait a minute!! What was that???\n\n");
  printf("\t1.\tA bird?\n");
  printf("\t2.\tA plane?\n");
  printf("\t3.\tA control character?\n");
  printf("\n\t\t\b\bAnd how will this look when it prints out?\n\n");
  return 0;
}
```

输出如下：

```
Hi there!

This program is a bit longer than the others.
But really it's only more text.

Hey, wait a minute!! What was that???
1. A bird?
2. A plane?
3. A control character?

And how will this look when it prints out?
```

代码的说明

这个程序看起来有点复杂，这只是因为括号内的文本字符串包含了许多转义序列。每个文本字符串都由一对双引号括起来。但这个程序只是连续调用 printf()函数，说明屏幕输出是由传送给 printf()函数的数据所控制。

本例通过预处理指令包含了标准库中的 stdio.h 文件:

```
#include <stdio.h>            // Include the header file for input and output
```

这是一个预处理指令，因为它以符号#开头。stdio.h 文件提供了使用 printf()函数所需的定义。

然后，定义 main()函数头，指定它返回一个整数值:

```
int  main(void)
```

括号中的 void 关键字表示不给 main()函数传递信息。下一行的大括号表示其下是函数体:

```
{
```

下一行语句调用标准库函数 printf()，将“Hi there!”输出到屏幕上，接着空两行，输出“This program is a bit”。

```
printf("Hi there!\n\n\nThis  program  is  a  bit");
```

空两行是由 3 个转义序列\n 生成的。转义序列\n 会把字符显示在新行上。第一个转义序列\n 结束了包含“Hi there!”的行，之后的两个转义序列\n 生成两个空行，文本“This program is a bit”显示在第 4 行上。这行代码在屏幕上生成了 4 行输出。

下一个 printf()生成的输出跟在上一个 printf()输出的最后一个字符后面。下面的语句输出文本“ longer than the others.”，其中的第一个字符是一个空白:

```
printf(" longer  than  the  others.");
```

这个输出跟在上一个输出的后面，紧临 bit 中的 t。所以在文本的开头需要一个空格，否则计算机就会显示“This program is a bitlonger than the others. ”，这不是我们希望的结果。

下一个语句在输出前会先换行，因为双引号中文本字符串的开头是\n:

```
printf("\nBut  really  itS  only  more  text.\n\n\n\a\a");
```

显示完文本后会空两行(因为有 3 个\n 转义序列)，然后发出两次鸣响。下一个屏幕输出从空的第二行开始。

下一个输出语句如下:

```
printf("Hey, wait  a  minute!!  What  was  that???\n\n");
```

输出文本后空一行。其后的输出在空的这行开始。

以下 3 行语句各插入一个制表符，显示一个数字后，再插入另一个制表符，之后是一些文本，结束后换行。这样，输出更容易阅读：

```
printf("\t1.\tA  bird?\n");
printf("\t2.\rA  plane?\n");
printf("\t3.\tA  control  character?\n");
```

这几个语句会生成 3 行带编号的输出。

下一语句先输出一个换行符，所以在前面输出的后面是一个空行，然后输出两个制表符和两个空格，接着退回两个空格，最后显示文本并换行：

```
printf("\n\t\t\b\bAnd how will this look when it prints out?\n\n");
```

函数体中的最后一个语句如下：

```
return 0;
```

这个语句结束 main()的执行，把 0 返回给操作系统。

结束大括号表示函数体结束：

```
}
```

注意：

输出中制表符和退格的实际效果随编译器的不同而不同。

1.12 常见错误

错误是生活中的一部分。用 C 语言编写计算机程序时，必须用编译器将源代码转换成机器码，所以必须用非常严格的规则控制使用 C 语言的方式。漏掉一个该有的逗点，或添加不该有的分号，编译器都不会将程序转换成机器码。

即使实践了多年，程序中也很容易出现输入错误。这些错误可能在编译或链接程序时找出。但有些错误可能使程序执行时，表面上看起来正常，却不定时地出错，这就需要花很多时间来跟踪错误了。

当然，不是只有输入错误会带来问题，具体实施时也常常会发现问题。在处理程序中复杂的判断结构时，很容易出现逻辑错误。从语言的观点看，程序是正确的，编译及运行也正确，但得不到正确的结果。这类错误最难查找。

1.13 要点

温习第一个程序是个不错的方法，图 1-4 列出了重点。

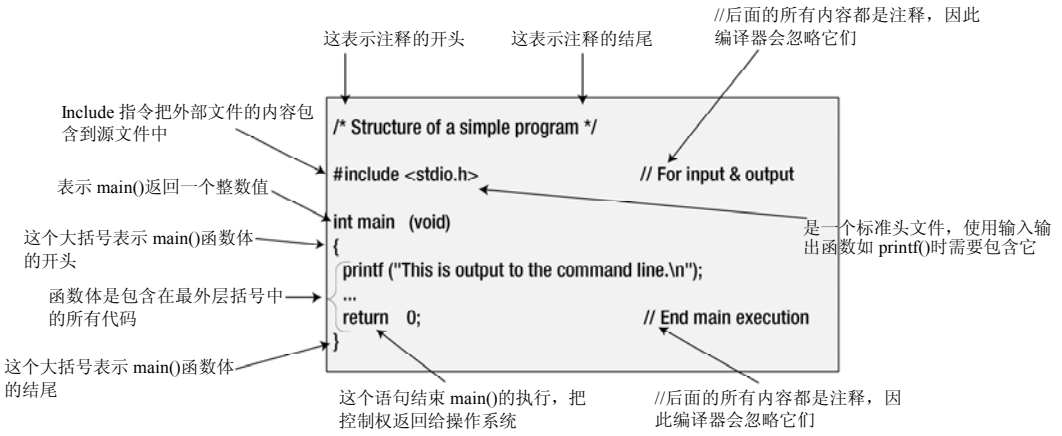

图 1-4 简单程序的要素

1.14 小结

本章编写了几个 C 程序。我们学习了许多基础知识，本章的重点是介绍一些基本概念，而不是详细探讨 C 程序语言。现在读者应该对编写、编译及链接程序很有信心了。也许读者目前对如何构建 C 程序只有模糊的概念。以后学了更多的 C 语言知识，编写了一些程序后，就会清楚明白了。

下一章将学习较复杂的内容，而不只是用 printf()输出文本。我们要处理信息，得到更有趣的结果。另外，printf()不只是显示文本字符串，它还有其他用途。

1.15 习题

以下的习题能让读者测试本章所学的成果。如果有不懂的地方，可以翻看本章的内容，还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 1.1 编写一个程序，用两个 printf()语句分别输出自己的名字及地址。

习题 1.2 将上一个练习改成所有的输出只用一个 printf()语句。

习题 1.3 编写一个程序，输出下列文本，格式如下所示：

```
"It's  freezing  in  here," he said coldly.
```

第 2 章

编 程 初 步

现在读者一定很渴望编写程序，让计算机与外界进行实际的交互。我们不希望程序只能做打字员的工作，显示包含在程序代码中的固定信息。的确，编程的内涵远不止此。理想情况下，我们应能从键盘上输入数据，让程序把它们存储在某个地方，这会让程序更具多样性。程序可以访问和处理这些数据，而且每次执行时，都可以处理不同的数据值。每次运行程序时输入不同的信息正是整个编程业的关键。在程序中存储数据项的地方是可以变化的，所以叫做变量(variable)，而这正是本章的主题。

本章的主要内容：

- 内存的用法及变量的概念
- 在 C 中如何计算
- 变量的不同类型及其用途
- 强制类型转换的概念及其使用场合
- 编写一个程序，计算树木的高度

2.1 计算机的内存

首先看看计算机如何存储程序要处理的数据。为此，就要了解计算机的内存，在开始编写第一个程序之前，先简要介绍计算机的内存。

计算机执行程序时，组成程序的指令和程序所操作的数据都必须存储到某个地方。这个地方就是机器的内存，也称为主内存(main memory)，或随机访问存储器(Random Access Memory，RAM)。RAM 是易失性存储器。关闭 PC 后，RAM 的内容就会丢失。PC 把一个或多个磁盘驱动器作为其永久存储器。要在程序结束执行后存储起来的任何数据，都应打印出来或写入磁盘，因为程序结束时，存储在 RAM 中的结果就会丢失。

可以将计算机的 RAM 想象成一排井然有序的盒子。每个盒子都有两个状态：满为 l，空为 0。因此每个盒子代表一个二进制数：0 或 1。计算机有时用真(true)和假(false)表示它们：1 是真，0 是假。每个盒子称为一个位(bit)，即二进制数(binary digit)的缩写。

注意：

如果读者不记得或从来没学过二进制数，可参阅附录 A。但如果不明白这些内容，不用担心，因为这里的重点是计算机只能处理 0 与 1，而不能直接处理十进制数。程序使用的所有数据(包括程序指令)都是由二进制数组成的。

为了方便起见，内存中的位以 8 个为一组，每组的 8 位称为一个字节(byte)。为了使用字节的内容，每个字节用一个数字表示，第一个字节用 0 表示，第二个字节用 1 表示，直到计算机内存的最后一个字节。字节的这个标记称为字节的地址(address)。因此，每个字节的地址都是唯一的。每栋房子都有一个唯一的街道地址。同样，字节的地址唯一地表示计算机内存中的字节。

总之，内存的最小单位是位(bit)，将 8 个位组合为一组，称为字节(byte)。每个字节都有唯一的地址。字节地址从 0 开始。位只能是 0 或 1，如图 2-1 所示。

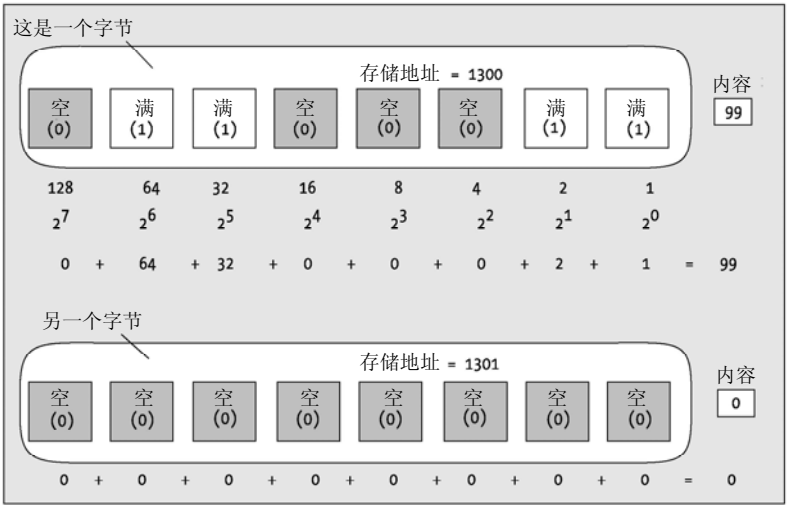

图 2-1 内存中的字节

计算机内存的常用单位是千字节(KB)、兆字节(MB)、千兆字节 (GB)。大型磁盘驱动器使用兆兆字节(TB)。这些单位的意义如下：

- 1KB 是 1 024 字节。
- 1MB 是 1 024KB，也就是 1 048 576 字节。
- 1GB 是 1 024MB，也就是 1 073 741 841 字节。
- 1TB 是 1 024GB，也就是 1 099 511 627 776 字节。

如果 PC 有 1GB 的 RAM，字节地址就是 0~1 073 741 841。为什么不使用更简单的整数，例如千、百万或亿？因为从 0 到 1023 共 1024 个数字，而在二进制中，1023 的 10 个位刚好全是 l：11 1111 1111，它是一个非常方便的二进制数。1000 是很好用的十进制数，但是在二进制的计算机里就不再那么方便了，它是 11 1110 1000。因此以 KB(1 024 字节)为单位，是为了方便计算机使用。同样，MB 需要 20 个位，GB 需要 30 个位。

但是硬盘的容量可能出现混乱。磁盘制造商常常宣称他们生产的磁盘的容量是 256GB 或 1TB，而实际上这两个数字表示 2560 亿字节及 1 万亿字节。当然，2560 亿字节只有 231MB，而 1 万亿字节只有 911GB，所以磁盘制造商给出的硬盘容量有误导作用。

有了字节的概念，下面看看如何在程序里使用这些内存。

2.2 什么是变量

变量是计算机里一块特定的内存，它是由一个或多个连续的字节所组成，一般是 1、2、4、8 或 16 字节。每个变量都有一个名称，可以用该名称表示内存的这个位置，以提取它包含的数据或存储一个新数值。

下面编写一个程序，用第 1 章介绍的 printf()函数显示你的薪水。假设你的薪水是 10 000 元/月，则很容易编写这个程序。

```
// Program 2.1 What is a Variable?
#include <stdio.h>

int main(void)
{
  printf("My salary is $10000");
  return 0;
}
```

这个程序的工作方式不需要多做解释，它和第一章开发的程序差不多。如何修改这个程序，让它能够根据存储在内存中的值，定制要显示的信息？这有几种方法，它们有一个共同点：使用变量。

在这个例子里，可以分配一块名为 salary 的内存，把值 10 000 存储在该变量中。要显示薪水时，可以使用给变量指定的名称 salary，将存储在其中的值 10 000 显示出来。程序用到变量名时，计算机就会访问存储在其中的值。变量的使用次数是不受限制的。当薪水改变时，只要改变 salary 变量存储的值，整个程序就会使用新的值。当然，在计算机中，所有的值都存储为二进制数。

程序中变量的数量是没有限制的。在程序执行过程中，每个变量包含的值由程序的指令来决定。变量的值不是固定的，而可以随时改变，且没有次数的限制。

注意：

变量可以有一个或多个字节，那么，计算机如何知道变量有多少个字节？下一节会提到，每个变量都有类型来指定变量可以存储的数据种类。变量的类型决定了为它分配多少个字节。

变量的命名

给变量指定的名称一般称为变量名。变量的命名是很有弹性的。它可以是一个或多个大写或小写字母、数字和下划线(_)(有时下划线也算作字母)，但要以字母开头。下面是一些正确的变量名：

```
Radius   diameter   Auntie_May   Knotted_Wool   D678
```

变量名不能以数字开头，所以 8_Ball 和 6_pack 都是不合法的名称。变量名只能包含字母、下划线和数字，所以 Hash!及 Mary-Lou 都不能用作变量名。Mary-Lou 是一个常

见的错误，但是 Mary_Lou 就是可以接受的。变量名中不能有空格，所以 Mary Lou 会被视为两个变量名 Mary 和 Lou。以一或两个下划线开头的变量名常用在头文件中，所以在给变量命名时，不要将下划线用作第一个字符，以免和标准库里的变量名冲突。例如最好避免使用_this 和_that 这样的变量名。变量名的另一个要点是，变量名是区分大小写的，因此 Democrat 和 democrat 是不同的。

可以在上述限制内随意指定变量名，但最好使变量名有助于了解该变量包含的内容，例如用变量名 x 来存储薪水信息就不好，而使用变量名 salary 就好得多，对其用途不会有什么疑义。

警告：

变量名可以包含的字符数取决于编译器，遵循 C 语言标准的编译器至少支持 31 个字符，只要不超过这个长度就没问题。建议变量名不要超过这个长度，因为这样的变量名比较繁琐，代码也难以理解。有些编译器会截短过长的变量名。

2.3 存储整数的变量

变量有几种不同的类型，每种变量都用于存储特定类型的数据。有几种变量可存储整数、非整数的数值和字符。一些类型存储特定的数据(例如整数)，它们之间的区别是它们占用的内存量和可以存储的数值范围。首先看看用于存储整数的变量。

整数是没有小数点的数字。下面是一个例子：

```
123      10,999,000,000      20,000      88      1
```

这些数值是整数，但这对程序而言并不完全正确。整数是不能包含逗号的，所以第二个值在程序里应该写成 10999000000，第三个值应写成 20000。

下面是一些不是整数的例子：

```
1.234    999.9    2.0    -0.0005    3.14159265
```

2.0 一般算作整数，但是计算机不将它算作整数，因为它带有小数点。在程序里，必须把这个数字写作 2，不带小数点。在 C 程序中，整数总是写成不带小数点的数字，如果数字中有小数点，就不是整数，而是浮点数，详见后面的内容。在详细讨论整型变量之前，先看看程序里一个简单的变量，学习变量的用法。

试试看：使用变量

回到输出薪水的例子。将前面的程序改为使用一个 int 型变量:

```
// Program 2.2 Using a variable
#include <stdio.h>

int main(void)
{
  int salary;                          // Declare a variable called salary
```

```
  salary = 10000;                    // Store 10000 in salary
  printf("My salary is %d.\n", salary);
  return 0;
}
```

输入这个例子，编译、链接并执行，会得到下面的结果：

```
My salary is 10000.
```

代码的说明

前三行和前一个例子相同，下面看看新的语句。用来存放薪水的变量声明语句如下：

```
int salary;                    // Declare a variable called salary
```

这个语句称为变量声明，因为它声明了变量的名称。在这个程序中，变量名是 salary。

警告：
变量声明语句以分号结束。如果漏掉分号，程序编译时会产生错误。

变量声明也指定了这个变量存储的数据类型，这里使用关键字 int 指定，salary 用来存放一个整数。关键字 int 放在变量名称之前。这是可用于存储整数的几个类型之一。

如后面所述，声明存储其他数据类型的变量时，要使用另一个关键字指定数据类型，其方式大致相同。

注意：
关键字是特殊的 C 保留字，对编译器有特殊的意义。不能将它们用作变量名称或代码中的其他实体，否则编译器会生成错误消息。

变量声明也称为变量的定义，因为它分配了一些存储空间，来存储整数值，该整数可以用变量名 salary 来引用。

注意：
声明引入了一个变量名，定义则给变量分配存储空间。有这个区别的原因在本书后面会很清楚。

当然，现在还未指定变量 salary 的值，所以此刻该变量包含一个垃圾值，即上次使用这块内存空间时遗留在此的值。

下一个语句是：

```
salary = 10000;                    // Store 10000 in salary
```

这是一个简单的算术赋值语句，它将等号右边的数值存储到等号左边的变量中。这里声明了变量 salary，它的值是 10 000。将右边的值 10 000 存储到左边的变量 salary 中。等号“=”称为赋值运算符，它将右边的值赋予左边的变量。

然后是熟悉的 printf()语句，但这里的用法和之前稍有不同：

```
printf("My salary is %d.", salary);
```

括号内有两个参数，用逗号分开。参数是传递给函数的值。在这个程序语句中，传给 printf()函数的两个参数如下：

- 参数 1 是一个控制字符串，用来控制其后的参数输出以什么方式显示，它是放在双引号内的字符串，也称为格式字符串，因为它指定了输出数据的格式。
- 参数 2 是变量名 salary。这个变量值的显示方式是由第一个参数——控制字符串来确定。

这个控制字符串和前一个例子相当类似，都包含一些要显示的文本。但在本例的这个字符串中有一个%d，它称为变量值的转换说明符(conversion specifier)。

转换说明符确定变量在屏幕上的显示方式，换言之，它们指定最初的二进制值转换为什么形式，显示在屏幕上。在本例中使用了 d，它是应用于整数值的十进制说明符，表示第二个参数 salary 输出为一个十进制数。

注意：

转换说明符总是以%字符开头，以便 printf()函数识别出它们。控制字符串中的%总是表示转换说明符的开头，所以如果要输出%字符，就必须用转义序列%%。

试试看：使用更多的变量

试试一个稍大的程序：

```
// Program 2.3 Using more variables
#include <stdio.h>

int main(void)
{
  int brothers;                    // Declare a variable called brothers
  int brides;                      // and a variable called brides

  brothers = 7;                    // Store 7 in the variable brothers
  brides = 7;                      // Store 7 in the variable brides

  // Display some output
  printf("%d brides for %d brothers\n", brides, brothers);
  return 0;
}
```

执行程序的结果如下：

```
7 brides for 7 brothers;
```

代码的说明

这个程序和前一个例子相当类似。首先声明两个变量 brothers 和 brides，语句如下：

```
int brothers;                      // Declare a variable called brothers
int brides;                        // and a variable called brides
```

两个变量都声明为 int 类型，都存储整数值。注意，它们在两个语句中声明。由于这两个变量的类型相同，故可以将它们放在同一行代码上声明：

```
int brothers, brides;
```

在一个语句中声明多个变量时，必须用逗号将数据类型后面的变量名分开，该语句要用分号结束。这是一种很方便的格式，但有一个缺点：每个变量的作用不很明显，因为它们全放在一行代码上，不能加入注释来描述每个变量。因此可以将它们分成两行，语句如下：

```
int brothers,                    // Declare a variable called brothers
    brides;                      // and a variable called brides
```

将语句分成两行，就可以加入注释了。这些注释会被编译器忽略，因此和最初没加入注释的语句相同。可以将 C 语句分成好几行。分号决定语句的结束，而不是代码行的结束。

当然也可以编写两个声明语句。一般最好在一个语句中定义一个变量。变量声明常常放在函数的可执行语句的开头，但这不是必须的。一般把要在一块代码中使用的变量声明放在该起始括号的后面。

之后的两个语句给两个变量赋值 7：

```
brothers = 7;                    // Store 7 in the variable brothers
brides = 7;                      // Store 7 in the variable brides
```

注意，声明这些变量的语句放在上述语句之前。如果遗漏了某个声明，或把声明语句放在后面，程序就不会编译。变量在其声明之前在代码中是不存在的，必须总是在使用变量之前声明它。

下一个语句调用 printf()函数，它的第一个参数是一个控制字符串，以显示一行文本。这个字符串还包含规范，指定后续参数的值如何解释和显示在文本中。这个控制字符串中的两个转换说明符%d 会分别被 printf()函数的第二个参数 brides 和第三个参数 brothers 的值取代：

```
printf("%d brides for %d brothers\n", brides, brothers);
```

转换说明符按顺序被 printf()函数的第二个参数 brides 和第三个参数 brothers 的值取代：变量 brides 的值对应第一个%d，变量 brothers 的值对应第二个%d。如果将设置变量值的语句改为如下所示，将会更清楚：

```
brothers = 8;                    // Store 8 in the variable brothers
brides = 4;                      // Store 4 in the variable brides
```

在这个比较明确的例子中，printf()语句会清楚地显示变量和转换说明符的对应关系，因为输出如下所示：

```
4 brides for 8 brothers
```

为了演示变量名是区分大小写的，修改 printf()函数，使其中一个变量名以大写字母

开头，如下所示：

```
// Program 2.3A Using more variables
#include <stdio.h>

int main(void)
{
  int brothers;                          // Declare a variable called brothers
  int brides;                            // and a variable called brides

  brothers = 7;                          // Store 7 in the variable brothers
  brides = 7;                            // Store 7 in the variable brides

  // Display some output
  printf("%d brides for %d brothers\n", Brides, brothers);
  return 0;
}
```

编译这个版本的程序时，会得到一个错误信息。编译器把 brides 和 Brides 解释为两个不同的变量，所以它不理解 Brides 这个变量，因为没有声明它。这是一个常见的错误，如前所述，打字和拼写错误是出错的一个主要原因。变量必须在使用之前声明，否则编译器就无法识别，将该语句标识为错误。

2.3.1 变量的使用

前面介绍了如何声明及命名变量，但这和在第一章学到的知识相比并没有太多用处。下面编写另一个程序，在产生输出前使用变量的值。

试试看：作一个简单的计算

这个程序用变量的值做简单的计算：

```
// Program 2.4 Simple calculations
#include <stdio.h>

int main(void)
{
  int total_pets;
  int cats;
  int dogs;
  int ponies;
  int others;

  // Set the number of each kind of pet
  cats = 2;
  dogs = 1;
  ponies = 1;
  others = 46;

  // Calculate the total number of pets
  total_pets = cats + dogs + ponies + others;

  printf("We have %d pets in total\n", total_pets);    // Output the result
```

```
  return 0;
}
```

执行程序的结果如下:

```
We have 50 pets in total
```

代码的说明

与前面的例子一样，大括号中的所有语句都有相同的缩进量，这说明这些语句都包含在这对大括号中。应当仿效此法组织程序，使位于一对大括号之间的一组语句有相同的缩进量，使程序更易于理解。

首先，定义 5 个 int 类型的变量:

```
int total_pets;
int cats;
int dogs;
int ponies;
int others;
```

因为这些变量都用来存放动物的数量，它们肯定是整数，所以都声明为 int 类型。

下面用 4 个赋值语句给变量指定特定的值:

```
cats = 2;
dogs = 1;
ponies = 1;
others = 46;
```

现在，变量 Total_Pets 还没有设定明确的值，它的值是使用其他变量进行计算的结果:

```
total_pets = cats + dogs + ponies + others;
```

在这个算术语句中，把每个变量的值加在一起，计算出赋值运算符右边的所有宠物数的总和，再将这个总和存储到赋值运算符左边的变量 Total_Pets 中。这个新值替代了存储在变量 Total_Pets 中的旧值。

printf()语句显示了变量 Total_Pets 的值，即计算结果:

```
printf("We have %d pets in total\n", total_pets);
```

试着改变某些宠物的值，或增加一些宠物。记住要声明它们，给它们设定数值，将它们加进变量 Total_Pets 中。

2.3.2 变量的初始化

在上面的例子，用下面的语句声明每个变量:

```
int cats;                // The number of cats as pets
```

用下面的语句设定变量 Cats 的值：

```
Cats = 2;
```

将变量 Cats 的值设为 2。这个语句执行之前，变量的值是什么？它可以是任何数。第一个语句创建了变量 Cats，但它的值是上一个程序在那块内存中留下的数值。其后的赋值语句将变量 Cats 的值设置为 2。但最好在声明变量时，就初始化它，语句如下所示：

```
int Cats = 2;
```

这个语句将变量 Cat 声明为 int 类型，并设定初值为 2。声明变量时就初始化它一般是很好的做法。它可避免对初始值的怀疑，当程序运作不正常时，它有助于追踪错误。避免在创建变量时使用垃圾值，可以减少程序出错时计算机崩溃的机会。随意使用垃圾值可能导致各种问题，因此从现在起，就养成初始化变量的好习惯，即使是 0 也好。

上面的程序是第一个真正做了些事情的程序。它非常简单，仅仅相加了几个数字，但这是非常重要的一步。它是运用算术语句进行运算的一个基本例子。下面介绍一些更复杂的计算。

1. 基本算术运算

在 C 语言中，算术语句的格式如下：

```
变量名 = 算术表达式;
```

赋值运算符右边的算术表达式指定使用变量中存储的值和/或明确给出的数字，以及算术运算符如加(+)、减(−)、乘(*)及除(/)进行计算。在算术表达式中也可以使用其他运算符，如后面所述。

前面例子中的算术语句如下：

```
total_pets = cats + dogs + ponies + others;
```

这个语句先计算等号右边的算术表达式，再将所得的结果存到左边的变量中。

在 C 语言中，符号“＝”定义了一个动作，而不是像数学中那样说明两边相等。它指定将右边表达式的结果存到左边的变量中。因此可以编写下面的语句：

```
total_pets = total_pets + 2;
```

以数学的观点来看，它是很荒唐的，但对编程而言它是正确的。假定重新编写程序，添加上面的语句。添加了这个语句的程序段如下：

```
total_pets = cats + dogs + ponies + others;
total_pets = total_pets + 2;
printf("The total number of pets is: %d", total_pets);
```

在执行完第一个语句后，Total_Pets 的值是 50。之后，第二行提取 Total_Pets 的值，给该值加 2，再将结果存储回变量 Total_Pets。因此最后显示出来的总数是 52。

注意：

在赋值运算中，先计算等号右边的表达式，然后将结果存到等号左边的变量中。新

的值取代赋值运算符左边的变量中的原值。赋值运算符左边的变量称为 lvalue，因为在这个位置可以存储一个值。执行赋值运算符右边的表达式所得的值称为 rvalue，因为它是计算表达式所得的一个值。

计算结果是数值的表达式称为算术表达式，下面都是算术表达式：

```
3      1 + 2      total_pets      cats + dogs – ponies      -data
```

计算这些表达式，都会得到一个数值。注意，变量名也是一个表达式，它的计算结果是一个值，即该变量包含的值。最后一个例子的值是 data 的负值，所以如果 data 包含 −5，表达式−data 的值就是 5。当然，data 的值仍是−5。稍后将详细讨论如何构建表达式，并学习运算规则。这里先用基本算术运算符做一些简单的例子。表 2-1 列出了这些算术运算符。

表 2-1 基本算术运算符

运 算 符	动 作
+	加
−	减
*	乘
/	除
%	取模(Modulus)

应用运算符的数据项一般称为操作数，两边的操作数都是整数时，所有这些运算符都生成整数结果。前面没有提到过取模运算符。它用运算符左边的表达式值去除运算符右边的表达式值，并求出其余数，所以有时称为余数运算符。表达式 12 % 5 的结果是 2。因为 12 除以 5 的余数是 2。下一节将详细介绍。所有这些运算符的工作方式都与我们的常识相同，只有除法运算符例外，它应用于整数时有点不直观。下面进行一些算术运算。

注意：

应用运算符的值称为操作数。需要两个操作数的运算符（如%）称为二元运算符。应用于一个值的运算符称为一元运算符。因此-在表达式 a-b 中是二元运算符，在表达式 −data 中是一元运算符。

试试看：减和乘

下面基于食物的程序演示了减法和乘法：

```
// Program 2.5 Calculations with cookies
#include <stdio.h>

int main(void)
{
  int cookies = 5;
  int cookie_calories = 125;           // Calories per cookie
```

```
  int total_eaten = 0;                  // Total cookies eaten

  int eaten = 2;                        // Number to be eaten
  cookies = cookies - eaten;            // Subtract number eaten from cookies
  total_eaten = total_eaten + eaten;
  printf("\nI have eaten %d cookies. There are %d cookies left",
                                                          eaten, cookies);

  eaten = 3;                           // New value for cookies eaten
  cookies = cookies - eaten;           // Subtract number eaten from cookies
  total_eaten = total_eaten + eaten;
  printf("\nI have eaten %d more. Now there are %d cookies left\n", eaten, cookies);
  printf("\nTotal energy consumed is %d calories.\n", total_eaten*cookie_calories);
  return 0;
}
```

这个程序产生如下输出:

```
I have eaten 2 cookies. There are 3 cookies left
I have eaten 3 more. Now there are 0 cookies left

Total energy consumed is 625 calories.
```

代码的说明

首先声明并初始化 3 个 int 类型的变量:

```
int cookies = 5;
int cookie_calories = 125;            // Calories per cookie
int total_eaten = 0;                  // Total cookies eaten
```

在程序中，使用变量 total_eaten 计算吃掉的饼干总数，所以要将它初始化为 0。下一个声明并初始化的变量存储吃掉的饼干数，如下:

```
int eaten = 2;                        // Number to be eaten
```

用减法运算符从 cookies 中减掉 eaten:

```
cookies = cookies - eaten;            // Subtract number eaten from cookies
```

减法运算的结果存回 cookies 变量，所以 cookies 的值变成 3。因为吃掉了一些饼干，所以要给 total_eaten 增加吃掉的饼干数:

```
total_eaten = total_eaten + eaten;
```

将 eaten 变量的当前值 2 加到 total_eaten 的当前值 0 上,结果存储回变量 total_eaten。printf()语句显示剩下的饼干数:

```
printf("\nI have eaten %d cookies. There are %d cookies left",
                                                          eaten, cookies);
```

这个语句在一行上放不下，所以在 printf()的第一个参数后的逗号后面，将该语句的其他内容放在下一行上。可以像这样分拆语句，使程序易于理解，或放在屏幕的指定宽

度之内。注意不能用这种方式拆分第一个字符串参数。不能在字符串的中间放置换行符。需要将字符串拆开成两行或多行时，一行上的每一段字符串必须有自己的一对双引号。例如，上面的语句可以写成:

```
printf("\nI have eaten %d cookies. "
       " There are %d cookies left",
       eaten, cookies);
```

如果两个或多个字符串彼此相邻，编译器会将它们连接起来，构成一个字符串。

用整数值的转换说明符%d 将 eaten 和 cookies 的值显示出来。在输出字符串中，eaten 的值取代第一个%d， cookies 的值取代第二个%d。字符串在显示之前会先换行，因为开头处有一个\n。

下一个语句将变量 eaten 的值设为一个新值:

```
eaten = 3;                              // New value for cookies to be eaten
```

新值 3 取代 eaten 变量中的旧值 2。然后完成和以前一样的操作序列:

```
cookies = cookies - eaten;              // Subtract number eaten from cookies
total_eaten = total_eaten + eaten;
printf("\nI have eaten %d more. Now there are %d cookies left\n", eaten, cookies);
```

最后，在执行 return 语句，结束程序前，计算并显示被吃掉饼干的卡路里数:

```
printf("\nTotal energy consumed is %d calories.\n", total_eaten*cookie_calories);
```

printf()函数的第二个参数是一个算术表达式，而不是变量。编译器会将表达式 total_eaten*cookie_calories 的计算结果存储到一个临时变量中，再把该值作为第二个参数传送给 printf()函数。函数的参数总是可以使用算术表达式，只要其计算结果是需要的类型即可。

下面看看除法和取模运算符。

试试看：除法和取模运算符

假设你有一罐饼干(其中有 45 块饼干)和 7 个孩子。要把饼干平分给每个孩子，计算每个孩子可得到几块饼干，分完后剩下几块饼干。

```
// Program 2.6 Cookies and kids
#include <stdio.h>

int main(void)
{
  int cookies = 45;                     // Number of cookies in the jar
  int children = 7;                     // Number of children
  int cookies_per_child = 0;            // Number of cookies per child
  int cookies_left_over = 0;            // Number of cookies left over

  // Calculate how many cookies each child gets when they are divided up
  cookies_per_child = cookies/children; // Number of cookies per child
```

```
  printf("You have %d children and %d cookies\n", children, cookies);
  printf("Give each child %d cookies.\n", cookies_per_child);

  // Calculate how many cookies are left over
  cookies_left_over = cookies%children;
  printf("There are %d cookies left over.\n", cookies_left_over);
  return 0;
}
```

执行程序后的输出:

```
You have 7 children and 45 cookies
Give each child 6 cookies.
There are 3 cookies left over.
```

代码的说明

下面一步一步地解释这个程序。下面的语句声明并初始化 4 个整数变量: cookies、children、cookies_per_child、cookies_left_over:

```
int cookies = 45;                       // Number of cookies in the jar
int children = 7;                       // Number of children
int cookies_per_child = 0;              // Number of cookies per child
int cookies_left_over = 0;              // Number of cookies left over
```

使用除号运算符“/”将饼干数量除以孩子的数量，得到每个孩子分得的饼干数:

```
cookies_per_child = cookies/children;    // Number of cookies per child
```

下面两个语句输出结果，即 cookies/children 变量的值:

```
printf("You have %d children and %d cookies\n", children, cookies);
printf("Give each child %d cookies.\n", cookies_per_child);
```

从输出结果可以看出，cookies_per_child 的值是 6。这是因为当操作数是整数时，除法运算符总是得到整数值。45 除以 7 的结果是 6，余 3。下面的语句用取模运算符计算余数:

```
cookies_left_over = cookies%children;
```

赋值运算符右边的表达式计算 cookies 除以 children 得到的余数。最后一个语句输出余数:

```
printf("There are %d cookies left over.\n", cookies_left_over);
```

2. 深入了解整数除法

当一个操作数是负数时，使用除法和模数运算符的结果是什么？在执行除法运算时，如果操作数不同号，结果就是负数。因此，表达式-45/7 和 45/-7 的结果相同，都是-6。如果操作数同号，都是正数或都是负数，结果就是正数。因此 45/7 和-45/-7 结果都是 6。至于模数运算符，不管操作数是否同号，其结果总是和左操作数的符号相同。因此 45%-7

等于 3，-45/7 等于-3，-45/-7 也等于-3。

3. 一元运算符

例如，乘法运算符是一个二元运算符。因为它有两个操作数，其结果是一个操作数乘以另一个操作数。还有一些运算符是一元运算符，即它们只需一个操作数。后面将介绍更多的例子。但现在看看一个最常用的一元运算符。

4. 一元减号运算符

前面使用的运算符都是二元运算符，因为它们都操作两个数据项。C 语言中也有操作一个数据项的一元运算符。一元减号运算符就是一个例子。若操作数为负，它就生成正的结果，若操作数为正，它就生成负的结果。要了解一元减号运算符的使用场合，考虑一下追踪银行账号。假定我们在银行存了 200 元。在簿子里用两列记录这笔钱的收支情况，一列记录付出的费用，另一列记录得到的收入，支出列是负数，收入列是正数。

我们决定购买一片价值 50 元的 CD 和一本价值 25 元的书。假使一切顺利，从银行的初始值中减掉支出的 75 元后，就得到了余额。表 2-2 说明这些项的记录情况。

表 2-2 收入与支出记录

项	收 入	支 出	存 款 余 额
支票收入	$200		$200
CD		$50	$150
书		$25	$125
结余	$200	$75	$125

如果将这些数字存储到变量中，可以将收入及支出都输入为正数，只有计算余额时，才会把这些数字变成负数。为此，可以将一个负号(-)放在变量名的前面。

要把总支出输出为负数，可编写如下语句：

```
int expenditure = 75;
printf("Your balance has changed by %d.", -expenditure);
```

这会产生如下结果：

```
Your balance has changed by -75.
```

负号表示花掉了这笔钱，而不是赚了。注意，表达式-expenditure 不会改变 expenditure 变量的值，它仍然是 75。这个表达式的值是-75。

在表达式-expenditure 中，一元减号运算符指定了一个动作，其结果是翻转 expenditure 变量的符号：将负数变成正数，将正数变成负数。这和编写一个负数(如-75 或-1.25)时使用的负号运算符是不同的。此时，负号不表示一个动作，程序执行时，不需要执行指令。它只是告诉编译器，在程序里创建一个负的常量。

2.4 变量与内存

前面介绍了整数变量，但未考虑过它们占用多少内存空间。每次声明给定类型的变量时，编译器都会给它分配一块足够大的内存空间，来保存该类型的变量。相同类型的不同变量总是占据相同大小的内存(字节数)。但不同类型的变量需要分配的内存空间就不一样了。

本章的开头介绍了，计算机的内存组织为字节。每个变量都会占据一定数量的内存字节，那么存储整数需要几个字节？这取决于整数值有多大。一个字节能存储−128~+127的整数。这对于前面的例子而言已经足够，但是如何存储一双及膝的长筒袜上的平均针脚数？一个字节就不够了。另一方面，如果要记录一个人在两分钟中能吃掉的汉堡包个数，一个字节就足够了，此时分配更多的字节就是浪费内存了。因此在 C 语言中有不同类型的变量来存储不同类型的数字，其中一个就是整数。整数变量还有几种不同的变体，以存储不同范围的整数。

2.4.1 带符号的整数类型

有 5 种基本的变量类型可以声明为存储带符号的整数值(无符号的整数值参见下一节)。每种类型都用不同的关键字或关键字组合来指定，如表 2-3 所示。

表 2-3 整数变量类型的名称

类 型 名 称	字 节 数
signed char	1
short int	2
int	4
long int	4
long long int	8

下面是这些类型的变量声明：

```
short shoe_size;
int house_number;
long long star_count;
```

类型名称 short、long 和 long long 可以用作 short int、long int 和 long long int 的缩写，前面还可以带有 signed 关键字。但是，这些类型几乎总是用表 2-3 列出的缩写形式。Int 类型也可以写作 signed int，但不常用。表 2-3 列出了每个类型的字节数，但这些变量类型所占的内存空间，以及可以存储的取值范围，取决于所使用的编译器。很容易确定编译器允许的极限值，因为它们在 limits.h 头文件中定义，本章后面会介绍。

2.4.2 无符号的整数类型

有些数据总是正的，例如河滩上的鹅卵石数目。对于每个存储带符号整数的类型，都有一个对应的类型来存储无符号的整数，它们占用的内存空间与无符号类型相同。每个无符号的类型名称都与带符号的类型名称相同，但要在前面加上关键字 unsigned。表 2-4 列出了可用的无符号整数类型。

表 2-4 无符号整数类型的名称

类 型 名 称	字 节 数
unsigned char	1
unsigned short int 或 unsigned short	2
unsigned int	4
unsigned long int 或 unsigned long	4
unsigned long long int 或 unsigned long long	8

如果位数给定，可以表示的数值就是固定的。32 位的变量可以表示 4 294 967 295 个不同的值。因此，使用无符号类型所提供的值不会多于对应的带符号类型，但其表示的数字比对应的带符号类型大一倍。

下面是声明无符号整型变量的示例：

```
unsigned int count;
unsigned long population;
```

注意：

如果变量的类型不同，但占用相同的字节数，则它们仍是不同的。Long 和 int 类型占用相同的内存量，但它们仍是不同的类型。

2.4.3 指定整数常量

整数变量有不同的类型，整数常量也有不同的类型。例如，如果将整数写成 100，它的类型就是 int。如果要确保它是 long 类型，就必须在这个数值的后面加上一个大写 L 或小写 l。所以，long 类型的整数 100 应写为 100L。虽然写为 100l 也是合法的，但应尽量避免，因为小写字母 l 与数字 1 很难辨别。

声明并初始化 Big_Number 的语句如下：

```
long Big_Number = 1287600L;
```

负整数常量的定义要用负号，例如：

```
int decrease = -4;
long below_sea_level = -100000L;
```

将整数常量指定为 long long 类型时，应添加两个 L：

```
long long really_big_number = 123456789LL;
```

如前所述，将常量指定为无符号类型时，应添加 U，如下所示：

```
unsigned int count = 100U;
unsigned long value = 999999999UL;
```

要存储取值范围最大的整数，可以按如下方式定义变量：

```
unsigned long long metersPerLightYear = 9460730472580800ULL;
```

ULL 指定，初始值的类型是 unsigned long long。

1. 十六进制常量

也可以用十六进制编写整数，即以 16 为基底。十六进制的数字等价于十进制的 0~15，表示方式是 0~9 和 A~F(或 a~f)。因为需要一种方式区分十进制的 99 和十六进制的 99，所以在十六进制数的前面加上 0x 或 0X。因此在程序中，十六进制的 99 可以编写成 0x99 或 0X99。十六进制常量也可以有后缀。下面是十六进制常量的一些示例：

```
0xFFFF    0xdead    0xfade    0xFade    0x123456EE    0xafL    0xFABABULL
```

最后一个示例的类型是 unsigned long long，倒数第二个示例的类型是 long。

十六进制常量常用来表示位模式，因为每一个十六进制的数对应于 4 个二进制位。两个十六进制的数指定一个字节。第 3 章介绍的按位运算符一般与十六进制常量一起用于定义掩码。如果不熟悉十六进制，可以参阅附录 A。

2. 八进制常量

八进制数以 8 为基底。八进制数字为 0~7，对应于二进制中的 3 位。八进制数起源于计算机内存采用 36 位字的时代，那时一个字是 3 位的组合。因此，36 位二进制字可以写成 12 个八进制数。八进制数目前很少使用，需要知道它们，以免错误地指定八进制数。

以 0 开头的整数常量，例如 014，会被编译器看作八进制数。因此，014 等价于十进制的 12，而不是十进制的 14。所以，不要在整数中加上前导 0，除非要指定八进制数。很少需要使用八进制数。

3. 默认的整数常量类型

如前所述，没有后缀的整数常量默认为 int 类型，但如果该值太大，在 int 类型中放不下，该怎么办？对于这种情形，编译器创建了一个常量类型，根据值是否有后缀，来判断该值是否是十进制。表 2-5 列出了编译器如何判断各种情形下的整数类型。

表 2-5 无符号整数类型的名称

后 缀	十进制常量	八或十六进制常量
无	1. int 2. long 3. long long	1. int 2. unsigned int 3. long 4. unsigned long 5. long long 6. unsigned long long
U	1. unsigned int 2. unsigned long 3. unsigned long long	1. unsigned int 2. unsigned long 3. unsigned long long
L	1. long 2. long long	1. long 2. unsigned long 3. long long 4. unsigned long long
UL	1. unsigned long 2. unsigned long long	1. unsigned long 2. unsigned long long
LL	1. long long	1. long long 2. unsigned long long
ULL	1. unsigned long long	1. unsigned long long

编译器选择容纳该值的第一种类型，如表中各项的数字所示。例如，后缀为 u 或 U 的十六进制常量默认为 unsigned int，否则就是 unsigned long。如果这个取值范围太小，就采用 unsigned long long 类型。当然，如果给变量指定的初始值在变量类型的取值范围中放不下，编译器就会发出一个错误消息。

2.5 使用浮点数

浮点数包含的值带小数点，也可以表示分数和整数。下面是浮点数的例子：

```
1.6     0.00008     7655.899  100.0
```

最后一个常量是整数，但它存储为浮点数，因为存在小数点。由于浮点数的表示方式，它的位数是固定的。这会限制浮点数的精确度，是一个缺点，然而它的取值范围要比整数大得多。浮点数通常表示为一个小数值乘以 10 的次方。例如前面的每一个浮点数都可以采用表 2-6 的方式来表示。

表 2-6 浮点数表示法

数 值	使用指数表示法	在 C 语言中也可以写成
1.6	0.16×10^{1}	0.16E1
0.00008	0.8×10^{-4}	0.8E-4
7655.899	0.7655899×10^{4}	0.7655899E4
100.0	1.0×10^{2}	1.0E2

中间列说明，左列的数如何用指数表示法来表示，但在 C 语言中不使用这种方式：而是表示这些数值的一个替代方法。右列说明了中间列的数字在 C 语言中的表示法。这些数字中的 E 表示指数，也可以使用小写 e。当然在程序中编写这些数字时可以不用指数，而使用左列的方式，但对于非常大或非常小的数字，指数形式比较方便。0.5E-15 当然比 0.0 000 000 000 000 005 更好。

浮点数的表示

浮点数的内部表示有点复杂。如果对计算机的内部不感兴趣，可以跳过这一节。这里包含本节，是因为理解计算机如何处理浮点数，可以更好地明白浮点数为什么有这样的值域。图 2-2 显示了在 Intel PC 的内存中，浮点数如何存储在 4 字节的字中。

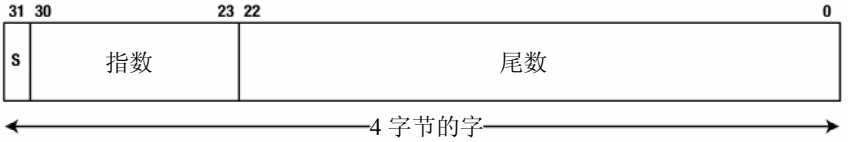

图 2-2　内存中的浮点数

这是一个单精度浮点数，在内存中占用 4 字节。该值包含三部分：

- 符号位，正值为 0，负值为 1
- 8 位的指数
- 23 位的尾数

尾数包含浮点数中的小数，占用 23 位。它假定为一个形式为 1.bbb...b 的二进制值，二进制点的右边有 23 位。因此，尾数的值总是大于等于 1，小于 2。那么，如何把 24 位值放在 23 位中，其实这很简单。最左边的一位总是 1，所以不需要存储。采用这种方式，可以给精度提供一个额外的二进制数字。

指数是一个无符号的 8 位值，所以指数值可以是 0~255。浮点数的实际值是尾数乘以 2 的指数幂 2^{exp}，其中 exp 是指数值。使用负的指数值可以表示很小的分数。为了包含这个浮点数表示，给浮点数的实际指数加上 127，这将允许把-127~128 的值表示为 8 位无符号值。因此指数为-6 会存储为 121，指数为 6 会存储为 133。但还有几个复杂的问题。

实际指数为-127，而存储的指数是 0，这是一种特殊情况。浮点数 0 表示为尾数和指数的所有位都是 0，所以实际指数为-127 时，不能用于其他值。

另一个复杂的问题是，最好能检测出除 0 的情形。于是系统保留了另外两个特殊值，来表示+无穷大和-无穷大，它们分别是正数和负数除以 0 的结果。正数除以 0 的结果是符号位为 0，所有指数位是 1，所有尾数位是 0。这个值很特殊，表示+无穷大，不是 1×2^{128}，且所有尾数位是 0。负数除以 0 的结果是这个值取负，所以 -1×2^{128} 也是一个特殊值。

最后一个复杂的问题是，最好能表示 0 除以 0 的结果。这称为 Not a Number(NaN)。这个保留值的所有指数位是 1，尾数的首位是 1 或 0，这取决于 NaN 只是一个 NaN，允

许继续执行，还是一个发出信号的 NaN，在代码中生成一个可中断执行的异常。NaN 在尾数中有一个前导 0 时，则其他尾数位中的至少一位是 1，就可以把它与无穷大区分开。

警告：

因为计算机把浮点数存储为二进制尾数和二进制指数的组合体，所以一些十进制的小数值不能用这种方式精确地表示。尾数中二进制点右边的二进制位，例如.1、.01、.001、.0001 等，等于十进制分数 1/2、1/4、1/8、1/16 等。所以二进制尾数的分数部分只能表示这些十进制分数的子集之和。可以看出，1/3 或 1/5 等值不能用二进制尾数精确地表示，因为二进制小数不能精确地组合为这些值。

2.6 浮点数变量

浮点数变量类型只能存储浮点数。表 2-7 是 3 种不同的浮点数变量。

表 2-7 浮点数变量类型

关 键 字	字 节 数	数 值 范 围
f1oat	4	±3.4E±38(精确到 6 到 7 位小数)
double	8	±1.7E±308(精确到 15 位小数)
long double	12	±1.19E±4932(精确到 18 位小数)

这是浮点数类型通常占用的字节数和取值范围。与整数一样，这些数所占用的字节数和取值范围取决于机器和编译器。在一些编译器上，类型 long double 和 double 相同。注意，小数的精确位数只是一个大约的数，因为浮点数在内部是以二进制方式存储的，十进制的浮点数在二进制中并不总是有精确的表示形式。

浮点数变量的声明方式和整数变量类似。只需要给浮点数类型使用对应的关键字即可：

```
float radius;
double biggest;
```

如果需要存储至多有 7 位精确值的数(范围从 10^{-38} 到 10^{+38})，就应需要使用 float 类型的变量。类型 float 的值称为单精度浮点数。从表 2-6 中得知，它占用 4 个字节。使用类型 double 的变量可以存储双精度浮点数。类型 double 的变量占用 8 个字节，有 15 位精确值，范围从 10^{-308} 到 10^{+308}。它足以满足大多数的需求。但某些特殊的应用程序需要更精确、更大的范围，此时可以使用 long double，但这取决于编译器。

编写一个类型为 float 的常量，需要在数值的末尾添加一个 f，以区别 double 类型。用下面的语句初始化前面的两个变量：

```
float radius=2.5f;
double biggest=123E30;
```

变量 radius 的初值是 2.5，变量 biggest 初始化为 123 后面加 30 个零。任何数，只要有小数点，就是 double 类型，除非加了 f，使它变为 float 类型。当用 E 或 e 指定指数值

时，这个常量就不需要包含小数点。例如 1E3f 是 float 类型，3E8 是 double 类型。

要声明 long double 类型的常量，需要在数字的末尾添加一个大写 L 或小写 l，例如：

```
long double huge = 1234567.89123L;
```

2.6.1 使用浮点数完成除法运算

如前所见，使用整数操作数进行除法运算时，通常会得到整数结果。除非除法运算的左操作数刚好是右操作数的整数倍，否则其结果是不正确的。当然，在将饼干分给孩子们的例子中，整数除法运算的方式是没问题的，但将 10 尺长的厚板均分成 4 块时，就有问题了。这时就需要用到浮点数了。

使用浮点数进行除法运算，会得到正确的结果——至少是一个精确到固定位数的值。下一个例子说明如何使用 float 类型的变量进行除法运算。

试试看：使用 float 类型值的除法

这个例子用一个浮点数除以另一个浮点数，然后显示其结果：

```
// Program 2.7 Division with float values
#include <stdio.h>

int main(void)
{
  float plank_length = 10.0f;          // In feet
  float piece_count = 4.0f;            // Number of equal pieces
  float piece_length = 0.0f;           // Length of a piece in feet

  piece_length = plank_length/piece_count;
  printf("A plank %f feet long can be cut into %f pieces %f feet long.\n",
                                       plank_length, piece_count, piece_length);
  return 0;
}
```

程序的结果输出如下：

```
A plank 10.000000 feet long can be cut into 4.000000 pieces 2.500000 feet long.
```

代码的说明

如何平均切割木板是很容易理解的。注意，在 printf()语句中为 float 类型的值使用了新的格式说明符。

```
printf("A plank %f feet long can be cut into %f pieces %f feet long.\n",
                                     plank_length, piece_count, piece_length);
```

使用格式说明符%f 显示浮点数。格式说明符一般必须对应输出的值的类型。如果使用格式说明符%d 输出 float 类型的值，就会得到一个垃圾值。因为浮点数会解释为整数。同样，如果使用%输出整数类型的值，也会得到垃圾值。

2.6.2 控制输出中的小数位数

在上个例子的输出中有太多不必要的 0。擅长使用量尺和锯子，并不说明能用长度为 2.500000 的量尺切割木板，更不用说用 2.500001 长度的量尺了。可以用格式说明符指定小数点后面的位数。例如，要使输出的小数点后有两位数，可以使用格式说明符%.2f。如果小数点后需要有 3 位数，则可以使用%.3f。

可以修改上一个例子中的 printf()语句，生成更适当的结果：

```
printf("A plank %.2f feet long can be cut into %.0f pieces %.2f feet long.\n",
                                plank_length, piece_count, piece_length);
```

第一个格式说明符对应于变量 plank_length，其结果的小数点后有两位数。第二个格式说明符指定小数点后没有数字，这很合理，因为 piece_count 是整数。最后一个格式说明符和第一个相同。因此执行这个版本的例子，输出如下：

```
A plank 10.00 feet long can be cut into 4 pieces 2.50 feet long.
```

这样看起来舒服多了。当然，使 piece_count 是整数类型会更好。

2.6.3 控制输出的字段宽度

输出的字段宽度是输出值所使用的总字符数(包括空格)，在这个程序中，它是默认的。printf()函数确定了输出值需要占用多少个字符位置，小数点后的位数由我们指定，并将它用作字段宽度。但我们可以自己确定字段宽度，也可以自己确定小数位数。如果要求输出一列排列整齐的数值，就应确定固定的字段宽度。如果让 printf()函数指定字段宽度，输出的数字列就不整齐。用于浮点数的格式说明符的一般形式是：

```
%[width][.precision][modifier]f
```

其中，方括号不包含在格式说明符中。它们包含的内容是可选的，所以可省略 width、.precision 或 modifier，或它们的任意组合。width 值是一个整数，指定输出的总字符数(包括空格)，即字段宽度。precision 值也是一个整数，指定小数点后的位数。当输出值的类型是 long double 时，modifier 部分就是 L，否则就省略它。

可以重写上个例子的 printf()调用，指定字段宽度及小数点后的位数，例如：

```
printf("A %8.2f plank foot can be cut into %5.0f pieces %6.2f feet long.\n",
                                  plank_length, piece_count, piece_length);
```

上面的代码略微修改了文本，使之能放在书页上。现在，第一个值的字段宽度为 8，小数点后有 2 位数。第二个值是切割的总片数，其字段宽度为 5 个字符，且没有小数部分。第三个值的字段宽度为 6，小数点后有 2 位数。

指定字段宽度时，数值默认为右对齐。如果希望数值左对齐，只需要在%的后面添加一个负号。例如，格式说明符%-10.4f 将输出一个左对齐的浮点数，其字段宽度为 10 个字符，小数点后有 4 位数。

注意，也可以对整数值指定字段宽度及对齐方式。例如%-15d 指定一个整数是左对齐，其字段宽度为 15 个字符。还有其他格式说明符，以后会学习它们。用前面的例子试试各种不同的输出，尤其是看看字段宽度太小时会出现什么情况。

2.7 较复杂的表达式

算术要比两个数相除复杂得多。事实上，如果要进行复杂的算术运算，也可以使用笔和纸。对于较复杂的计算，需要更多地控制表达式的计算顺序。括号可以提供这方面的能力。当遇到错综复杂的情况时，括号还有助于使表达式更清晰。

在算术表达式中可以使用括号，其使用次数不受限制。包含在括号中的子表达式的计算顺序是：从最内层的括号开始计算到最外层的括号，对于运算符的优先级，一般规则是先乘除后加减。因此，表达式 2*(3+3*(5+4))的值是 60。首先计算表达式 5+4，得到 9。然后乘以 3，得到 27。之后加上 3，得到 30，最后乘以 2，得到 60。

可以加入空格，将操作数和运算符分开，使算术表达式的可读性更高。需要使代码更紧凑时，则可以删除空格。无论采用哪种方式，编译器都不会受到影响，因为编译器会忽略空格。如果根据优先级规则，无法确定表达式的计算顺序，通常可以加进一些括号，确保生成需要的结果。

试试看：算术运算

这次要利用输入的直径计算一个圆桌的周长及面积。计算圆的周长及面积时，其数学公式要使用π或 pi(周长=2πr，面积=πr^2，其中 r 是半径)。如果不记得这些公式，也不用担心。这不是数学课本，所以只要理解程序是如何运作的即可。

```
// Program 2.8 calculations on a table
#include <stdio.h>

int main(void)
{
  float radius = 0.0f;               // The radius of the table
  float diameter = 0.0f;             // The diameter of the table
  float circumference = 0.0f;        // The circumference of the table
  float area = 0.0f;                 // The area of the table
  float Pi = 3.14159265f;

  printf("Input the diameter of the table:");
  scanf("%f", &diameter);            // Read the diameter from the keyboard

  radius = diameter/2.0f;            // Calculate the radius
  circumference = 2.0f*Pi*radius;    // Calculate the circumference
  area = Pi*radius*radius;           // Calculate the area

  printf("\nThe circumference is %.2f", circumference);
  printf("\nThe area is %.2f\n", area);
  return 0;
}
```

这个程序的输出如下:

```
Input the diameter of the table: 6

The circumference is 18.85.
The area is 28.27.
```

代码的说明

在第一个 printf()之前，这个程序看起来和以前的例子很类似:

```
float radius = 0.0f;                // The radius of the table
float diameter = 0.0f;              // The diameter of the table
float circumference = 0.0f;         // The circumference of the table
float area = 0.0f;                  // The area of the table
float Pi = 3.14159265f;
```

上述语句声明并初始化了 5 个变量，其中 Pi 有固定的数值。注意，所有的初值都在末尾添加了 f，因为这是 float 类型的初值。若没有 f 的话，它们的类型就是 double。不过在这里，它们仍然可行，但是编译器需要进行一些不必要的转换，将类型 double 转换为类型 float。Pi 值的位数太多，类型 float 存储不下，所以编译器提取其最左边的部分，使之能放在 float 类型中。

下一条语句输出一个从键盘上输入数据的提示:

```
printf("Input the diameter of the table:");
```

下一条语句读取圆桌的直径。这需要使用一个新的标准库函数 scanf():

```
scanf("%f", &diameter);             // Read the diameter from the keyboard
```

scanf()是另一个需要包含头文件 stdio.h 的函数。它专门处理键盘输入，提取通过键盘输入的数据，按照第一个参数指定的方式解释它，第一个参数是放在双引号内的一个控制字符串。在这里，这个控制字符串是%f。因为读取的值是 float 类型。scanf()将这个数存入第二个参数指定的变量 diameter 中。第一个参数是一个控制字符串，和 printf()函数的用法类似，但它控制的是输入，而不是输出。第 10 章将详细介绍 scanf()函数，附录 D 总结了所有的控制字符串。

注意，变量名 diameter 前的&是个新东西，它称为寻址运算符，它允许 scanf()函数将读入的数值存进变量 diameter。它的做法和将参数值传给函数是一样的。这里不详细解释它；第 8 章会详细说明。唯一要记住的是，使用函数 scanf()时，要在变量前加上寻址运算符&，而使用 printf()函数时不添加它。

在函数 scanf()的控制字符串中，%字符表示某数据项的格式说明符的开头。%字符后面的 f 表示输入一个浮点数。在控制字符串中一般有几个格式说明符，它们按顺序确定了函数中后面各参数的数据类型。在 scanf()的控制字符串后面有多少个参数，控制字符串就有多少个格式说明符，本书的后面将介绍 scanf()函数的更多运用，表2-8 列出了读取各种类型的数据时所使用的格式说明符:

表 2-8 读取数据的格式说明符

操 作	需要的控制字符串
读取 short 类型的数值	%hd
读取 int 类型的数值	%d
读取 long 类型的数值	%ld
读取 float 类型的数值	%f 或%e
读取 double 类型的数值	%lf 或%le

在%ld 和%lf 格式说明符中，l 是小写的 L。别忘了一定要在接收输入值的变量名前加上&。另外，如果使用了错误的格式说明符，如使用%d 读取 float 类型的数据，变量中的数值就不正确，但系统不会提示存储了一个垃圾值。

接下来的 3 条语句计算结果:

```
radius = diameter/2.0f;                    // Calculate the radius
circumference = 2.0f*Pi*radius;            // Calculate the circumference
area = Pi*radius*radius;                   // Calculate the area
```

第一条语句计算半径，将输入的直径除以 2。第二条语句用计算出来的半径计算桌子的周长。第三条语句计算面积。注意，如果忘了 2.0f 中的 f，编译器就会显示一个警告消息。这是因为如果没有 f，常量的类型就是 double，于是在一个表达式中混用了不同的类型。后面会详细描述这个问题。

可以编写如下语句来计算周长和面积:

```
circumference = 2.0f*Pi*(diameter/2.0f);     // Calculate the circumference
area = Pi*(diameter/2.0f)*(diameter/2.0f);   // Calculate the area
```

每个语句中的圆括号可以确保先计算半径的值，也有助于更清楚地说明正在计算的是半径的值。这些语句的缺点在于对半径的计算潜在地执行了三次，而实际上仅需要计算一次。智能的编译器可以优化这种代码，让半径的计算仅执行一次。

下面的两个语句输出计算后的数值:

```
printf("The circumference is %.2f. ", circumference);
printf("The area is %.2f.\n", area);
```

这两个 printf()语句用格式说明符%.2f 输出变量 circumference 和 area 的值。这个格式说明符指定输出的值在小数点后面有两位数。默认的字段宽度足以容纳要显示的变量值。

当然，可以执行这个程序，给直径输入任意值。试着输入各种不同形式的浮点数，例如输入 1E1f。

2.8 定义命名常量

前面的例子将 Pi 定义为变量，但它是一个不会改变的常量，π的值是一个不循环的无限小数，其值总是固定不变。唯一的问题是，在指定它时精确到几位数。最好确保它

的值在程序中保持不变，使之不会因错误而改变。

这有两种方法。第一是将 Pi 定义为一个符号，在程序编译期间用π的值取代它。此时，Pi 不是一个变量，而是它表示的值的一个别名。

试试看：定义一个常量

下面将 PI 指定为一个数值的别名：

```
// Program 2.9 More round tables
#include <stdio.h>
#define PI   3.14159f                  // Definition of the symbol PI

int main(void)
{
  float radius = 0.0f;
  float diameter = 0.0f;
  float circumference = 0.0f;
  float area = 0.0f;

  printf("Input the diameter of a table:");
  scanf("%f", &diameter);

  radius = diameter/2.0f;
  circumference = 2.0f*PI*radius;
  area = PI*radius*radius;

  printf("\nThe circumference is %.2f. ", circumference);
  printf("\nThe area is %.2f.\n", area);
  return 0;
}
```

这个输出和前面的例子完全相同。

代码的说明

在注释和头文件的#include 指令之后，有一个预处理指令：

```
#define PI 3.14159f                // Definition of the symbol PI
```

这里将 PI 定义为一个要被 3.14159f 取代的符号。使用 PI 而不是 Pi，是因为在 C 语言中有一个通用的约定：#define 语句中的标识符都是大写。只要在程序里的表达式中引用 PI，预处理器就会用#define 指令中的数值取代它。所有的取代动作都在程序编译之前完成。程序开始编译时，不再包含 PI 这个符号了，因为所有的 PI 都用#define 指令中的数值取代了。这些动作都是在编译器处理时在内部发生的，源程序没有改变，仍包含符号 PI。

警告：

预处理器在替代代码中的符号时，不会考虑它是否有意义。如果在替代字符串中出错，例如，如果编写了 3.14.159f，预处理器仍会用它替代每个 PI，而程序不会编译。

第二种方法是将 Pi 定义成变量，但告诉编译器，它的值是固定的，不能改变。声明变量时，在变量名前加上 const 关键字，可以固化变量的值，例如:

```
const float Pi = 3.14159f;          // Defines the value of Pi as fixed
```

以这种方式定义 Pi 的优点是，Pi 现在定义为指定类型的一个常量值。在前面的例子中，PI 只是一个字符序列，替代代码中的所有 PI。

在 Pi 的声明中添加关键字 const，会使编译器检查代码是否试图改变它的值。这么做的代码会被标记为错误，且编译失败。下面是它的一个例子。

试试看：定义一个其值固定的变量

在前面的例子中使用一个常量，但代码短一些:

```
// Program 2.10 Round tables again but shorter
#include <stdio.h>

int main(void)
{
  float diameter = 0.0f;                // The diameter of a table
  float radius = 0.0f;                  // The radius of a table
  const float Pi = 3.14159f;            // Defines the value of Pi as fixed

  printf("Input the diameter of the table:");
  scanf("%f", &diameter);

  radius = diameter/2.0f;

  printf("\nThe circumference is %.2f.", 2.0f*Pi*radius);
  printf("\nThe area is %.2f.\n", Pi*radius*radius);
  return 0;
}
```

代码的说明

下面是 Pi 变量的声明:

```
const float Pi = 3.14159f;            // Defines the value of Pi as fixed
```

这个语句声明了变量 Pi，并给它定义一个数值；Pi 在这里还是变量，但它的初始值是不可改变的。这是 const 修饰符的功劳。它可应用在声明任何类型的变量的语句中，固化该变量的值。编译器会检查代码是否试图改变声明为 const 的变量，如果发现有这种情况，编译器就会做出提示。可以设法骗过编译器，去改变 const 变量，但这违反了使用 const 的初衷。

下面两个语句输出程序的结果:

```
printf("\nThe circumference is %.2f.", 2.0f*Pi*radius);
printf("\nThe area is %.2f.\n", Pi*radius*radius);
```

在这个例子中，不再用变量存储周长及面积。现在这些表达式显示为 printf()函数的参数，它们的值会直接传给函数 printf()。

如前所述，传给函数的值可以是表达式的计算结果，此时，编译器会创建一个临时变量，来存储这个值，再传给函数。之后这个临时变量就被删除。这很好，只要不在其他地方使用这些数值即可。

2.8.1 极限值

当然，一定要确定程序中给定的整数类型可以存储的极限值。如前所述，头文件<limits.h>定义的符号表示每种类型的极限值。表 2-9 列出了对应于每种带符号整数类型的极限值符号名。

表 2-9 表示整数类型的极限值的符号

类 型	下 限	上 限
char	CHAR_MIN	CHAR_MAX
short	SHRT_MIN	SHRT_MAX
int	INT_MIN	INT_MAX
long	LONG_MIN	LONG_MAX
long long	LLONG_MIN	LLONG_MAX

无符号整数类型的下限都是 0，所以它们没有特定的符号。无符号整数类型的上限的符号分别是 UCHAR_MAX、USHRT_MAX、UINT_MAX、ULONG_MAX 和 ULLONG_MAX。

要在程序中使用这些符号，必须在源文件中添加<limits.h>头文件的#include 指令：

```
#include <limits.h>
```

可以用最大值初始化一个 int 变量，如下所示：

```
int number = INT_MAX;
```

这个语句把 number 的值设置为最大值，编译器会利用该最大值编译代码。

<float.h>头文件定义了表示浮点数的符号，其中一些的技术含量很高，所以这里只介绍我们感兴趣的符号。3 种浮点数类型可以表示的最大正值和最小正值如表 2-10 所示。还可以使用 FLT_DIG、DBL_DIG 和 LDBL_DIG 符号，它们指定了对应类型的二进制尾数可以表示的小数位数。下面用一个例子来说明如何使用表示整数和浮点数的符号。

表 2-10 表示浮点数类型的极限值的符号

类 型	下 限	上 限
float	FLT_MIN	FLT_MAX
double	DBL_MIN	DBL_MAX
long double	LDBL_MIN	LDBL_MAX

试试看：找出极限值

这个程序输出头文件中定义的符号的对应值。

```
// Program 2.11 Finding the limits
#include <stdio.h>            // For command line input and output
#include <limits.h>           // For limits on integer types
#include <float.h>            // For limits on floating-point types

int main(void)
{
  printf("Variables of type char store values from %d to %d\n", CHAR_MIN, CHAR_MAX);
  printf("Variables of type unsigned char store values from 0 to %u\n", UCHAR_MAX);
  printf("Variables of type short store values from %d to %d\n", SHRT_MIN, SHRT_MAX);
  printf("Variables of type unsigned short store values from 0 to %u\n", USHRT_MAX);
  printf("Variables of type int store values from %d to %d\n", INT_MIN, INT_MAX);
  printf("Variables of type unsigned int store values from 0 to %u\n", UINT_MAX);
  printf("Variables of type long store values from %ld to %ld\n", LONG_MIN, LONG_MAX);
  printf("Variables of type unsigned long store values from 0 to %lu\n", ULONG_MAX);
  printf("Variables of type long long store values from %lld to %lld\n", LLONG_MIN, LLONG_MAX);
  printf("Variables of type unsigned long long store values from 0 to %llu\n", ULLONG_MAX);

  printf("\nThe size of the smallest positive non-zero value of type float is %.3e\n", FLT_MIN);
  printf("The size of the largest value of type float is %.3e\n", FLT_MAX);
  printf("The size of the smallest non-zero value of type double is %.3e\n", DBL_MIN);
  printf("The size of the largest value of type double is %.3e\n", DBL_MAX);
  printf("The size of the smallest non-zero value of type long double is %.3Le\n", LDBL_MIN);
  printf("The size of the largest value of type long double is %.3Le\n", LDBL_MAX);

  printf("\n Variables of type float provide %u decimal digits precision. \n", FLT_DIG);
  printf("Variables of type double provide %u decimal digits precision. \n", DBL_DIG);
  printf("Variables of type long double provide %u decimal digits precision. \n",
                                                                        LDBL_DIG);
  return 0;
}
```

结果如下所示：

```
Variables of type char store values from -128 to 127
Variables of type unsigned char store values from 0 to 255
Variables of type short store values from -32768 to 32767
Variables of type unsigned short store values from 0 to 65535
Variables of type int store values from -2147483648 to 2147483647
Variables of type unsigned int store values from 0 to 4294967295
Variables of type long store values from -2147483648 to 2147483647
Variables of type unsigned long store values from 0 to 4294967295
Variables of type long long store values from -9223372036854775808 to 9223372036854775807
Variables of type unsigned long long store values from 0 to 18446744073709551615

The size of the smallest positive non-zero value of type float is 1.175e-038
The size of the largest value of type float is 3.403e+038
The size of the smallest non-zero value of type double is 2.225e-308
The size of the largest value of type double is 1.798e+308
The size of the smallest non-zero value of type long double is 3.362e-4932
The size of the largest value of type long double is 1.190e+4932
```

```
Variables of type float provide 6 decimal digits precision.
Variables of type double provide 15 decimal digits precision.
Variables of type long double provide 18 decimal digits precision.
```

代码的说明

在一系列的 printf()函数调用中，输出<limits.h>和<float.h>头文件定义的符号的值。计算机中的数值总是受限于该机器可以存储的值域，这些符号的值表示每种数值类型的极限值。这里用说明符%u 输出无符号整数值。如果用%d 输出无符号类型的最大值，则最左边的位(带符号类型的符号位)为 1 的数值就得不到正确的解释。

对浮点数的极限值使用说明符%e，表示这个数值是指数形式。同时指定精确到小数点后的 3 位数，因为这里的输出不需要非常精确。printf()函数显示的值是 long double 类型时，需要使用 L 修饰符。L 必须是大写，这里没有使用小写字母 l。%f 说明符表示没有指数的数值，它对于非常大或非常小的数来说相当不方便。在这个例子中试一试，就会明白其含义。

2.8.2 sizeof 运算符

使用 sizeof 运算符可以确定给定的类型占据多少字节。当然，在 C 语言中 sizeof 是一个关键字。表达式 sizeof(int)会得到 int 类型的变量所占的字节数，所得的值是一个 size_t 类型的整数。size_t 类型在标准头文件<stddef.h>(和其他头文件)中定义，对应于一个基本整数类型。但是，与 size_t 类型对应的类型可能在不同的 C 库中有所不同，所以最好使用 size_t 变量存储 sizeof 运算符生成的值，即使知道它对应的基本类型，也应如此。下面的语句是存储用 sizeof 运算符计算所得的数值：

```
size_t size = sizeof(long long);
```

也可以将 sizeof 运算符用于表达式，其结果是表达式的计算结果所占据的字节数。通常该表达式是某种类型的变量。除了确定某个基本类型的值占用的内存空间之外，sizeof 运算符还有其他用途，但这里只使用它确定每种类型占用的字节数。

试试看：确定给定类型占用的字节数

这个程序会输出每个数值类型占多少字节：

```
// Program 2.12 Finding the size of a type
#include <stdio.h>

int main(void)
{
  printf("Variables of type char occupy %u bytes\n", sizeof(char));
  printf("Variables of type short occupy %u bytes\n", sizeof(short));
  printf("Variables of type int occupy %u bytes\n", sizeof(int));
  printf("Variables of type long occupy %u bytes\n", sizeof(long));
  printf("Variables of type long long occupy %u bytes\n", sizeof(long long));
  printf("Variables of type float occupy %u bytes\n", sizeof(float));
  printf("Variables of type double occupy %u bytes\n", sizeof(double));
```

```
  printf("Variables of type long double occupy %u bytes\n", sizeof(long double));
  return 0;
}
```

输出如下:

```
Variables of type char occupy 1 bytes
Variables of type short occupy 2 bytes
Variables of type int occupy 4 bytes
Variables of type long occupy 4 bytes
Variables of type long long occupy 8 bytes
Variables of type float occupy 4 bytes
Variables of type double occupy 8 bytes
Variables of type long double occupy 12 bytes
```

代码的说明

因为 sizeof 运算符的结果是一个无符号整数，所以用%u 说明符输出它。注意，使用表达式 sizeof var_name 也可以得到变量 var_name 占用的字节数。显然，在关键字 sizeof 和变量名之间的空格是必不可少的。

现在已经知道编译器给每个数值类型指定的极限值和占用的字节数了。

注意：

如果希望把 sizeof 运算符应用于一个类型，则该类型名必须放在括号中，例如 sizeof (long double)。将 sizeof 运算符应用于表达式时，括号就是可选的。

2.9 选择正确的类型

必须仔细选择在计算过程中使用的变量类型，使之能包含我们期望的值。如果使用了错误的类型，程序就可能出现很难检测出来的错误。这最好用一个例子来说明。

试试看：变量的正确类型

下面的例子说明，如果给变量选择了不适当的类型，程序就会出错。

```
// Program 2.13 Choosing the correct type for the job 1
#include <stdio.h>

int main(void)
{
  const float Revenue_Per_150 = 4.5f;
  short JanSold = 23500;                          // Stock sold in January
  short FebSold = 19300;                          // Stock sold in February
  short MarSold = 21600;                          // Stock sold in March
  float RevQuarter = 0.0f; // Sales for the quarter

  short QuarterSold = JanSold + FebSold + MarSold; // Calculate quarterly total

  // Output monthly sales and total for the quarter
  printf("Stock sold in\n Jan: %d\n Feb: %d\n Mar: %d\n", JanSold, FebSold, MarSold);
```

```
    printf("Total stock sold in first quarter: %d\n", QuarterSold);

    // Calculate the total revenue for the quarter and output it
    RevQuarter = QuarterSold/150*Revenue_Per_150;
    printf("Sales revenue this quarter is:$%.2f\n", RevQuarter);
    return 0;
}
```

这些都是相当简单的计算，一季度的总销售量应是 64 400，它只是将每个月的销售量加在一起。但运行这个程序，输出如下：

```
Stock sold in
Jan: 23500
Feb: 19300
Mar: 21600
Total stock sold in first quarter: -1136
Sales revenue this quarter is:$-31.50
```

显然，结果不正确。把 3 个较大的正数加在一起，不应得到一个负值。

代码的说明

首先，代码定义了一个要在计算中使用的常量：

```
const float Revenue_Per_150 = 4.5f;
```

这个语句定义了每销售 150 个产品的收入。这没有什么错误。接着，声明 4 个变量，并给它们赋予初值：

```
short JanSold = 23500;                          // Stock sold in January
short FebSold = 19300;                          // Stock sold in February
short MarSold = 21600;                          // Stock sold in March
float RevQuarter = 0.0f;                        // Sales for the quarter
```

前 3 个变量的类型是 short，足以存储初值了。RevQuarter 变量是 float 类型，因为我们希望季度收入在小数点后有两位数。

下一个语句声明 QuarterSold 变量，并存储每月销售量的总和：

```
short QuarterSold = JanSold + FebSold + MarSold;  // Calculate quarterly total
```

事实上，结果错误的原因是 QuarterSold 变量的声明错误。该变量声明为 short 类型，其初始值指定为 3 个月销售量的总和。这个总和是 64 400，而程序输出了一个负数。这个语句一定有错误。

问题的原因是，我们试图在 QuarterSold 变量中存储对 short 类型而言过大的数字。short 变量能存储的最大值是 32 767，计算机不能正确解释 QuarterSold 的值，所以输出了一个负值。另一个考虑是季度销售量不会是负数，也许使用无符号的类型会更合适。

这个问题的解决方法是给 QuarterSold 变量使用 unsigned long 类型，来存储非常大的数字。还可以把存储每月销售量的变量指定为无符号。

解决问题

修改程序，再次运行它。只需要修改 main()函数体中的 5 行代码。修改的新程序如下:

```
// Program 2.14 Choosing the correct type for the job 2
#include <stdio.h>

int main(void)
{
  const float Revenue_Per_150 = 4.5f;
  unsigned short JanSold =23500;                    // Stock sold in January
  unsigned short FebSold =19300;                    // Stock sold in February
  unsigned short MarSold =21600;                    // Stock sold in March
  float RevQuarter = 0.0f;                          // Sales for the quarter

  unsigned long QuarterSold = JanSold + FebSold + MarSold;// Calculate quarterly total

  // Output monthly sales and total for the quarter
  printf("Stock sold in\n Jan: %d\n Feb: %d\n Mar: %d\n", JanSold, FebSold, MarSold);
  printf("Total stock sold in first quarter: %ld\n", QuarterSold);

  // Calculate the total revenue for the quarter and output it
  RevQuarter = QuarterSold/150*Revenue_Per_150;
  printf("Sales revenue this quarter is:$%.2f\n", RevQuarter);
  return 0;
}
```

运行这个程序，这次的输出是正确的:

```
Stock sold in
Jan: 23500
Feb: 19300
Mar: 21600
Total stock sold in first quarter: 64400
Sales revenue this quarter is :$1930.50
```

一季度的销售量是正确的，收入也是正确的。注意，这里使用%ld 输出总销售量，这就告诉编译器，使用 long 类型输出这个值。检查程序，用计算器计算出收入。

得到的结果应是$1 932，少了$1.50，这个数字虽然不大，但对于会计而言，这就是一个错误，必须找到少了的$1.50。程序在计算收入值时，发生了什么?

```
RevQuarter = QuarterSold/150*Revenue_Per_150;
```

这个语句给 RevQuarter 赋值，该值是等号右边表达式的结果。根据本章前面介绍的优先级规则，一步步地计算该表达式。这是一个非常简单的表达式，只需要从左向右计算，因为乘除的优先级相同。下面列出计算过程:

- QuarterSold/150 计算为 64400 / 150，结果应为 429.333。

这里有问题。QuarterSold 是一个整数，所以计算机将除法运算的结果四舍五入为一个整数，舍弃了.333。所以，在下一步计算中，结果会有出入。

- 429*Revenue_Per_150 计算为 429 * 4.5，结果为 1930.50。

知道哪里有错误后，如何更正它？可以将所有的变量都改为浮点数类型，但这违背了使用整数的初衷。输入的数字是整数，所以将它们存储到整数变量中。有较简单的解决方法吗？有两种。第一种是重写如下的语句：

```
RevQuarter = Revenue_Per_150*QuarterSold/150;
```

这个语句先执行乘法运算，因为对混合的操作数执行算术运算时，编译器会自动把整数操作数转换为浮点数，所以结果是 float 类型。对该结果除以 150，该操作也在 float 值上执行，并将 150 转换为 150f。于是，结果就是正确的。

第二种解决方法是把 150.0 作为除数。于是在除法运算执行之前，把被除数转换为浮点数。

我们不仅需要理解在不同类型的操作数上如何执行算术运算，还要理解如何控制类型的转换。在 C 语言中，可以将一种类型显式地转换为另一种类型。

2.10 强制类型转换

在程序 2.14 计算季度收入的表达式中，可以控制操作的执行，得到正确的结果：

```
RevQuarter = QuarterSold/150*Revenue_Per_150;
```

要使结果正确，必须修改这个语句，以浮点数的方式计算表达式。如果可以把 QuarterSold 的值转换为 float 类型，该表达式就会以浮点数的方式计算，问题就解决了。要把变量从一种类型转换为另一种类型，应把目标类型放在变量前面的括号中。因此，正确计算结果的表达式应如下所示：

```
RevQuarter = (float)QuarterSold/150*Revenue_Per_150;
```

这就是我们需要的表达式：在正确的地方使用正确类型的变量。当希望保留除法结果的小数部分时，不应使用整数运算。一种类型显式转换为另一种类型的过程称为强制类型转换(cast)。

当然，可以把表达式的结果从一种类型强制转换为另一种类型。此时，应把表达式放在括号中，例如：

```
double result = 0.0;
int a = 5;
int b = 8;
result = (double)(a + b)/2 - (a + b)/(double)(a*a + b*b);
```

把(a + b)的计算结果转换为 double，就确保除 2 运算用浮点数的方式进行。值 2 会转换为 double 类型，与除法运算的左操作数类型相同。把除数(a*a + b*b)的整数结果强制转换为 double，其效果与第二个除法运算类似；在执行除法运算之前，把左操作数的值转换为 double 类型。

2.10.1 自动转换类型

该程序的第二个版本的输出如下：

```
Sales revenue this quarter is :$1930.50
```

即使表达式中没有显式转换类型，结果也是浮点数形式，但它仍是错误的。结果是浮点数，是因为二元运算符要求其操作数有相同的类型。编译器在处理涉及不同类型的值操作时，会自动把其中一个操作数的类型转换为另一个操作数的类型。在二元算术运算中使用不同类型的操作数，编译器就会把其中一个值域较小的操作数类型转换为另一个操作数的类型，这称为隐式类型转换(implicit conversion)。再看看前面计算收入的表达式：

```
QuarterSold / 150 * Revenue_Per_150
```

它计算为 64400 (int) / 150 (int),，结果是 429 (int)，再将 429(int 转换为 float)乘以 4.5 (float)，得到 1930.5 (float)。

当二元运算符处理不同类型（包括不同的整数类型）的操作数时，总是会进行隐式类型转换。对于上述第一个操作，两个数字都是 int 类型，所以结果也是 int 类型。对于上述第二个操作，第一个值的类型是 int，第二个值的类型是 float。而 int 类型的值域小于 float 类型，所以自动将 int 类型的值转换为 float 类型。只要算术表达式中有混合类型的变量，C 编译器就会使用一组特殊的规则，确定表达式如何计算。下面就介绍这些规则。

2.10.2 隐式类型转换的规则

确定二元运算中的哪个操作数要转换为另一个操作数的类型时，其机制相当简单。其基本规则是，将值域较小的操作数类型转换为另一个操作数类型，但在一些情况下，两个操作数都要转换类型。

为了准确地表述这些规则，需要比上述更复杂的描述，所以可以忽略一些细节，在以后需要时再考虑它们。如果读者想了解全部规则，应继续阅读下去。

编译器按顺序采用如下规则，确定要使用的隐式类型转换：

(1) 如果一个操作数的类型是 long double，就把另一个操作数转换为 long double 类型。

(2) 否则，如果一个操作数的类型是 double，就把另一个操作数转换为 double 类型。

(3) 否则，如果一个操作数的类型是 float，就把另一个操作数转换为 float 类型。

(4) 否则，如果两个操作数的类型都是带符号的整数或无符号的整数，就把级别较低的操作数转换为另一个操作数的类型。无符号整数类型的级别从低到高为：

```
signed char, short, int, long, long long
```

每个无符号整数类型的级别都与对应的带符号整数类型相同，所以 unsigned int 类型

的级别与 int 类型相同。

(5) 否则，如果带符号整数类型的操作数级别低于无符号整数类型的级别，就把带符号整数类型的操作数转换为无符号整数类型。

(6) 否则，如果带符号整数类型的值域包含了无符号整数类型所表示的值，就把无符号整数类型转换为带符号整数类型。

(7) 否则，两个操作数都转换为带符号整数类型对应的无符号整数类型。

2.10.3 赋值语句中的隐式类型转换

赋值运算符右边的表达式值与其左边的变量有不同的类型时，也可以进行隐式类型转换。在一些情况下，这会截短数值，丢失数据。例如，如果赋值操作将 float 或 double 类型的值存储在 int 或 long 类型的变量中，float 或 double 的小数部分就会丢失，只存储整数部分。如下面的代码所示：

```
int number = 0;
float value = 2.5f;
number = value;
```

存储在 number 中的值是 2。这几行代码把 decimal 的值(2.5)赋予 int 类型的变量 number，就丢失了小数部分.5，只存储了 2。

赋值语句可能丢失信息，因为必须进行隐式类型转换，而编译器通常会为此发出一个警告。但是，代码仍可以编译，所以程序可能会得到不正确的结果。当需要在代码中进行可能导致丢失信息的类型转换时，最好使用显式类型转换。

下面的例子将说明赋值操作中的类型转换规则，代码如下：

```
double price = 10.0;                          // Product price per unit
long count = 5L;                              // Number of items
float ship_cost = 2.5F;                       // Shipping cost per order
int discount = 15;                            // Discount as percentage
long double total_cost = (count*price + ship_cost)*((100L - discount)/100.0F);
```

这些语句声明了 4 个变量，并根据给这些变量设置的值计算某个订单的总价。这里选择的类型主要用于演示隐式类型转换，它们不表示正常环境下的正确类型选择。下面看看最后一个语句如何计算 total_cost 的值：

(1) 先计算 count*price，再将 count 隐式转换为 double 类型，以进行乘法运算，结果是 double 类型，这源于第 2 个规则。

(2) 接着将 ship_cost 加到前一个操作的结果中。为此，要将 ship_cost 的值转换为前一个结果的类型 double。这个转换也源于第 2 个规则。

(3) 然后计算表达式 100L - discount，为此，要将 discount 的值转换为减法操作中另一个操作数的类型 long。这源于第 4 个规则，结果是 long 类型。

(4) 之后把上一个操作的结果(long 类型)转换为 float 类型，再除以 100.0F(float 类型)。这源于第 3 个规则，结果是 float 类型。

(5) 将第 2 步的结果除以第 4 步的结果，为此，要将上一个操作的 float 值转换为

double 类型，这源于第 3 个规则，结果是 double 类型。

(6) 最后，将上述结果存储在 total_cost 变量中，作为赋值操作的结果。当操作数的类型不同时，赋值操作总是要把右操作数的结果转换为左操作数的类型，所以上述操作的结果会转换为 long double 类型。编译器不会发出警告，因为 double 类型的所有值都可以表示为 long double 类型。

警告：

如果在代码中必须进行许多强制类型转换，存储数据的类型就可能选错了。

2.11 再谈数值数据类型

为了完整论述数值数据类型，下面讨论一些前面未提及的内容。第一个未涉及的类型是 char。char 类型的变量可以存储单个字符的代码。它只能存储一个字符代码(即一个整数)，所以被看为整数类型。可以像其他整数类型那样处理 char 类型存储的值，因此可以在算术运算中使用它。

2.11.1 字符类型

在所有数据的类型中，char 类型占用的内存空间最少。它一般只需一个字节。存储在 char 类型变量的整数可以表示为带符号或无符号的值，这取决于编译器。若表示为无符号的类型，则存储在 char 类型变量的值可以是 0~255。若表示为带符号的类型，则存储在 char 类型变量的值可以是 - 128~127。当然，这两个值域对应相同的位模式：0000 0000 到 1111 1111。对于无符号的值，这 8 位都是数据位，所以 0000 0000 对应于 0，1111 1111 对应于 255。对于带符号的值，最左边的 1 位是符号位，所以 - 128 的二进制值是 1000 0000，0 的二进制值是 0000 0000，127 的二进制值是 0111 1111。值 1111 1111 是一个带符号的二进制值，其对应的十进制值是 - 1。

从表示字符代码(位模式)的角度来看，char 类型是否带符号并不重要。重要的是何时对 char 类型的值执行算术运算。

可以给 char 类型的变量指定字符常量，作为其初始值。字符常量是一个放在单引号中的字符。下面是一些例子：

```
char letter = 'A';
char digit = '9';
char exclamation = '!';
```

也可以使用转义序列指定字符常量，例如：

```
char newline = '\n';
char tab = '\t';
char single_quote = '\'';
```

当然，上面的每个语句都把变量设置为单引号内的字符代码。实际的代码值取决于

计算机环境，但最常见的是美国标准信息交换码(ASCII)。ASCII 字符集参见附录 B。

还可以用整数值初始化 char 类型的变量，只要该值在编译器许可的 char 类型的值域内即可，如下面的例子：

```
char character = 74;              // ASCII code for the letter J
```

char 类型的变量有双重性：可以把它解释为一个字符，也可以解释为一个整数。下面的例子对 char 类型的值进行算术运算：

```
char letter = 'C';                // letter contains the decimal code value 67
letter = letter + 3;              // letter now contains 70, which is 'F'
```

因此，可以对 char 类型的值进行算术运算，同时仍把它当做一个字符。

注意：

无论 char 类型是实现为带符号还是不带符号的类型，char、signed char 和 unsigned char 类型都是不同的，需要进行转换，才能把一种类型映射到另一种类型。

2.11.2 字符的输入输出

使用 scanf()函数和格式说明符%c，可以从键盘上读取单个字符，将它存储在 char 类型的变量中，例如：

```
char ch = 0;
scanf("%c", &ch);                 // Read one character
```

如前所述，在使用 scanf()函数的源文件中，必须给<stdio.h>头文件添加#include 指令：

```
#include <stdio.h>
```

要使用 printf()函数将单个字符输出到命令行上，也可以使用格式说明符%c：

```
printf("The character is %c\n", ch);
```

当然，也可以输出该字符的数值：

```
printf("The character is %c and the code value is %d\n", ch, ch);
```

这个语句会把 ch 的值输出为一个字符和一个数值。

试试看：字符的建立

编程新手可能想知道，计算机如何知道它处理的是字符还是整数？事实是计算机并不知道。这就好像 Alice 使用 Humpty Dumpty(矮胖的人)时，会说，“我使用这个单词时，就意味着我给它赋予了“矮胖的人”这个含义。”同样，内存中的一个数据项的含义是我们赋予它的。包含值 70 的字节是一个整数，把 70 看作字母 J 的代码也是正确的。

下面的例子会说明这一点。这个例子使用转换说明符%c，它指定将 char 类型的值输出为一个字符，而不是一个整数。

```
// Program 2.15 Characters and numbers
#include <stdio.h>

int main(void)
{
  char first = 'T';
  char second = 63;

  printf("The first example as a letter looks like this - %c\n", first);
  printf("The first example as a number looks like this - %d\n", first);
  printf("The second example as a letter looks like this - %c\n", second);
  printf("The second example as a number looks like this - %d\n", second);
  return 0;
}
```

这个程序的输出如下:

```
The first example as a letter looks like this - T
The first example as a number looks like this - 84
The second example as a letter looks like this - ?
The second example as a number looks like this - 63
```

代码的说明

这个程序首先声明了两个 char 类型的变量:

```
char first = 'T';
char second = 63;
```

把第一个变量初始化为一个字符处理，第二个变量初始化为一个整数。接下来的 4 个语句以两种方式输出每个变量的值:

```
printf("The first example as a letter looks like this - %c\n", first);
printf("The first example as a number looks like this - %d\n", first);
printf("The second example as a letter looks like this - %c\n", second);
printf("The second example as a number looks like this - %d\n", second);
```

%c 转换说明符将变量的内容解释为单个字符，%d 说明符把它解释为一个整数。输出的数值是对应字符的代码。这个例子中的这些代码都是 ASCII 码。在大多数情况下字符代码都是 ASCII 码，所以本书都使用 ASCII 码。

提示:

如前所述，并不是所有的计算机都使用 ASCII 字符集，所以可能会得到与上述不同的值。但只要给字符常量使用了符号字符，无论采用什么字符编码，都会得到所需的字符。

用格式说明符%x 替代%d，就可以把 char 类型变量的整数值输出为十六进制值。

试试看：用字符的对应整数值进行算术运算

下面的例子将算术运算应用于 char 类型的值:

```
// Program 2.16 Using type char
#include <stdio.h>

int main(void)
{
  char first = 'A';
  char second = 'B';
  char last = 'Z';

  char number = 40;

  char ex1 = first + 2;                          // Add 2 to 'A'
  char ex2 = second - 1;                         // Subtract 1 from 'B'
  char ex3 = last + 2;                           // Add 2 to 'Z'

  printf("Character values %-5c%-5c%-5c\n", ex1, ex2, ex3);
  printf("Numerical equivalents %-5d%-5d%-5d\n", ex1, ex2, ex3);
  printf("The number %d is the code for the character %c\n", number, number);
  return 0;
}
```

运行这个程序，输出如下:

```
Character values      C    A    \
Numerical equivalents 67   65   92
The number 40 is the code for the character (
```

代码的说明

这个程序说明了如何对初始化为字符的 char 变量进行算术运算。main()函数体中的前 3 个语句如下:

```
char first = 'A';
char second = 'B';
char last = 'Z';
```

这些语句把变量 first、second 和 last 初始化为字符值。这些变量的数值是各个字符对应的 ASCII 码。它们可以看做数值和字符，所以可以对它们执行算术运算。

下一个语句用一个整数值初始化 char 类型的变量:

```
char number = 40;
```

初始值必须在单字节变量可以存储的值域内。对于笔者的编译器，char 是一个带符号的类型，所以其值必须在 128~127 之间。当然，也可以将该变量的内容解释为字符。在这个例子中，它是一个 ASCII 码为 40 的字符，即左括号。

接下来的 3 个语句又声明了 3 个 char 类型的变量:

```
char ex1 = first + 2;            // Add 2 to 'A'
char ex2 = second - 1;           // Subtract 1 from 'B'
char ex3 = last + 2;             // Add 2 to 'Z'
```

这些语句根据变量 first、second 和 last 中存储的值计算出新值，也就计算出了对应的新字符。这些表达式的结果存储在变量 ex1、ex2 和 ex3 中。

之后的两个语句以两种不同的方式输出 3 个变量 ex1、ex2 和 ex3:

```
printf("Character values      %-5c%-5c%-5c\n", ex1, ex2, ex3);
printf("Numerical equivalents %-5d%-5d%-5d\n", ex1, ex2, ex3);
```

第一个语句使用%5c 转换说明符把所存储的值解释为字符。它指定把值输出为字符，且左对齐，字符宽度为 5。第二个语句又输出了这些变量，但这次使用%5d 说明符把这些值解释为整数。对齐方式和字符宽度与第一个语句相同，但%5d 中的 d 指定输出是一个整数。在这两行输出中，第一行显示 3 个字符，第二行显示它们的 ASCII 码。

最后一行代码将 number 变量输出为一个字符和一个整数:

```
printf("The number %d is the code for the character %c\n", number, number);
```

变量要输出两次，只需要编写两次即可——printf()函数的第二和第三个参数。它先输出一个整数，再输出一个字符。

对字符执行算术运算的功能是很有用的。例如，要把大写字母转换为小写，只要给大写字母加上‘a’-‘A’的结果(ASCII 码 32)即可。要把小写字母转换为大写，只要减去‘a’-‘A’。附录 B 列出了字母字符的十进制 ASCII 值。当然，这个操作要求 a~z 和 A~Z 的字符代码是连续的整数。如果计算机使用的字符编码不是连续的整数，就不能这么做。

注意：

标准库 ctype.h 头文件提供的 toupper()和 tolower()函数可以把字符转换为大写和小写。

2.11.3 枚举

在编程时，常常希望变量存储一组可能值中的一个。例如一个变量存储表示当前月份的值。这个变量应只存储 12 个可能值中的一个，分别对应于 1~12 月。C 语言中的枚举(enumeration)就用于这种情形。

利用枚举，可以定义一个新的整数类型，该类型变量的值域是我们指定的几个可能值。下面的语句定义了一个枚举类型 Weekday：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

这个语句定义了一个类型，而不是变量。新类型的名称 Weekday 跟在关键字 enum 的后面，这个类型名称称为枚举的标记。Weekday 类型的变量值可以是类型名称后面的大括号中的名称指定的任意值。这些名称叫做枚举器(enumerator)或枚举常量(enumeration constant)，其数量可任意。每个枚举器都用我们赋予的唯一名称来指定，编译器会把 int 类型的整数值赋予每个名称。枚举是一个整数类型，因为指定的枚举器对应不同的整数值，这些整数默认从 0 开始，每个枚举器的值都比它之前的枚举器大 1。因此在这个例子中，Monday 到 Sunday 对应 0~6。

可以声明 Weekday 类型的一个新变量，并初始化它，如下所示：

```
enum Weekday today = Wednesday;
```

这个语句声明了一个变量 today，将它初始化为 Wednesday。由于枚举器有默认值，所以 Wednesday 对应 2。用于枚举类型变量的整数类型是由实现代码确定的，选择什么类型取决于枚举器的个数。

也可以在定义枚举类型时，声明该类型的变量。下面的语句就定义了一个枚举类型和两个变量：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
                          Friday, Saturday, Sunday} today, tomorrow;
```

这个语句声明了枚举类型 Weekday，定义了该类型的两个变量 today 和 tomorrow。还可以在同一个语句中初始化变量，如下所示：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
                      Friday, Saturday, Sunday} today = Monday, tomorrow = Tuesday;
```

这个语句把变量 today 和 tomorrow 初始化为 Monday 和 Tuesday。枚举类型的变量是整数类型，所以可以在算术表达式中使用。前面的语句还可以写为：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
              Friday, Saturday, Sunday} today = Monday, tomorrow = today + 1;
```

tomorrow 的初始值比 today 大 1。但是，在执行这个操作时，要确保算术运算的结果是一个有效的枚举值。

注意：

可以给枚举类型指定一组可能的值，但没有检查机制来确保程序只使用这些值。所以程序员要确保只为给定的枚举类型使用有效的枚举值。一种方式是只给枚举类型的变量赋予枚举常量名。

1. 选择枚举值

可以给任意或所有枚举器明确指定自己的整数值。尽管枚举器使用的名称必须唯一，但枚举器的值不要求是唯一的。除非有特殊的原因让某些枚举器的值相同，否则一般应确保这些值也是唯一的。下面的例子定义了 Weekday 类型，使其枚举器的值从 1 开始：

```
enum Weekday {Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

枚举器 Monday 到 Sunday 的对应值是 1~7。在明确指定了值的枚举器后面，枚举器会被赋予连续的整数值。这可能使枚举器有相同的值，如下面的例子所示：

```
enum Weekday {Monday = 5, Tuesday = 4, Wednesday,
              Thursday = 10, Friday = 3, Saturday, Sunday};
```

Monday、Tuesday、Thursday 和 Friday 明确指定了值，Wednesday 设置为 Tuesday+1，所以它是 5，Monday 与它相同。同样，Saturday 和 Sunday 设置为 4 和 5，所以它们的值也是重复的。完全可以这么做，但除非有很好的理由使一些枚举常量的值相同，否则这容易出现混淆。

只要希望变量有限定数量的可能值，就可以使用枚举。下面是定义枚举的另一个例子：

```
enum Suit{clubs = 10, diamonds, hearts, spades};
enum Suit card_suit = diamonds;
```

第一个语句定义了枚举类型 Suit，这个类型的变量可以有括号中的 4 个值的任意一个。第二个语句定义了 Suit 类型的一个变量，把它初始化为 diamonds，其对应的值是 11。还可以定义一个枚举，表示扑克牌的面值，如下所示：

```
enum FaceValue {two=2, three, four, five, six, seven,
                eight, nine, ten, jack, queen, king, ace};
```

在这个枚举中，枚举器的整数值匹配扑克牌的面值，其中 ace 的值最高。

在输出枚举类型的变量值时，会得到数值。如果要输出枚举器的名称，必须提供相应的程序逻辑，详见下一章的内容。

2. 未命名的枚举类型

在创建枚举类型的变量时，可以不指定标记，这样就没有枚举类型名了。例如：

```
enum {red, orange, yellow, green, blue, indigo, violet} shirt_color;
```

这里没有标记，所以这个语句定义了一个未命名的枚举类型，其可能的枚举器包括从 red 到 violet。该语句还声明了未命名类型的变量 shirt_color。

可以用通常的方式给 shirt_color 赋值：

```
shirt_color = blue;
```

显然，未命名枚举类型的主要限制是，必须在定义该类型的语句中声明它的所有变量。由于没有类型名，因此无法在代码的后面定义该类型的其他变量。

2.11.4 存储布尔值的变量

_Bool 类型存储布尔值。布尔值一般是比较的结果 true 或 false；第 3 章将学习比较操作，并使用其结果做出判断。_Bool 类型的变量值可以是 0 或 1，对应于布尔值 false 和 true。由于值 0 和 1 是整数，所以_Bool 类型也被看为整数类型。声明_Bool 变量的方式与声明其他整数类型一样，例如：

```
_Bool valid = 1;                // Boolean variable initialized to true
```

_Bool 并不是一个理想的类型名称。名称 bool 看起来更简洁、可读性更高，但布尔类型是最近才引入 C 语言的，所以选择类型名称_Bool，可以最大限度地减少与已有代码冲突的可能性。如果把 bool 选作类型名称，则在将 bool 作为一种内置类型的编译器上，使用 bool 名称的程序大都不会编译。

尽管如此，仍可以使用 bool 作为类型名称，只需在使用它的源文件中给<stdbool.h>标准头文件添加#include 指令即可。除了把 bool 定义为_Bool 的对应名称之外，<stdbool.h>

头文件还定义了符号 true 和 false，分别对应 1 和 0。因此，如果在源文件中包含了这个头文件，就可以将上面的声明语句改写为：

```
_Bool valid = 1;                // Boolean variable initialized to true
```

这似乎比上面的版本清晰得多，所以最好包含<stdbool.h>头文件，除非有特殊的理由。本书的其余部分使用 bool 表示布尔类型，但需要包含相应的头文件，其基本类型名称是_Bool。

可以在布尔值和其他数值类型之间进行类型转换。非零数值转换为 bool 类型时，会得到 1(true)，0 就转换为 0(false)。如果在算术表达式中使用 bool 变量，编译器就会在需要时插入隐式类型转换。bool 类型的级别低于其他类型，所以在涉及 bool 类型和另一个类型的操作中，bool 值会转换为另一个值的类型。这里不详细介绍如何使用布尔变量，具体内容详见下一章。

2.12 赋值操作的 op=形式

C 语言是一种非常简洁的语言，提供了一些操作的缩写形式。考虑下面的代码：

```
number = number + 10;
```

这类赋值操作是给一个变量递增或递减一个数字，它非常常见，所以有一个缩写形式：

```
number += 10;
```

变量名后面的+=运算符是 op=运算符家族中的一员。这个语句等价于上面的语句，但输入量少了许多。op=中的 op 可以是任意算术运算符：

```
+ - * / %
```

如果 number 的值是 10，就可以编写如下语句：

```
number *= 3;              // number will be set to number*3 which is 30
number /= 3;              // number will be set to number/3 which is 3
number %= 3;              // number will be set to number%3 which is 1
```

op=中的 op 也可以是其他几个运算符：

```
<< >> & ^ |
```

第 3 章将介绍这些运算符。op=运算符的工作方式都相同。如果有如下形式的语句：

```
lhs op= rhs;
```

其中 rhs 表示 op=运算符右边的表达式，该语句的作用与如下形式的语句相同：

```
lhs = lhs op (rhs);
```

注意 rhs 表达式的括号，它表示 op 应用于整个 rhs 表达式的计算结果值。为了加强理解，下面看几个例子。下面的语句：

```
variable *= 12;
```

等价于：

```
variable = variable * 12;
```

现在给一个整数变量加 1 有两种方式。下面的两个语句都给 count 加 1：

```
count = count + 1;
countd += 1;
```

下一章将介绍这个操作的另一种方式。有这么多选择，使编写 C 程序的人数无法统计。op=运算符中的 op 应用于 rhs 表达式的计算结果，所以如下语句：

```
a /= b + 1;
```

等价于：

```
a = a/(b + 1);
```

到目前为止，我们的计算能力比较受限。现在只能使用一组非常基本的算术运算符。而使用标准库的功能可以大大提升计算能力。所以在进入本章的最后一个例子之前，先看看标准库提供的一些数学函数。

2.13 数学函数

math.h 头文件包含各种数学函数的声明。为了了解这些数学函数，下面介绍最常用的函数。所有的函数都返回一个 double 类型的值。

表 2-11 列出了各种用于进行数值计算的函数，它们都需要 double 类型的参数。

表 2-11 用于进行数值计算的函数

函 数	操 作
floor(x)	返回不大于 x(double 类型)的最大整数
ceil(x)	返回不小于 x(double 类型)的最小整数
fabs(x)	返回 x 的绝对值
log(x)	返回 x 的自然对数(底为 e)
log10(x)	返回 x 的对数(底为 10)
exp(x)	返回 e^x 的值
sqrt(x)	返回 x 的平方根
pow(x)	返回 x^y 的值

给函数名的末尾添加 f 或 l，就得到处理 float 和 long double 类型的函数版本，所以 ceilf()应用于 float 值，sqrtl()应用于 long double 值。下面是使用这些函数的一些例子：

```
double x = 2.25;
double less = 0.0;
double more = 0.0;
double root = 0.0;
less = floor(x);              // Result is 2.0
more = ceil(x);               // Result is 3.0
root = sqrt(x);               // Result is 1.5
```

还有一些三角函数，如表 2-12 所示。给函数名的末尾添加 f 或 l，就得到处理 float 和 long double 类型的函数版本，参数和返回值的类型也是 float、double 或 long double，角度表示为弧度。

表 2-12 三 角 函 数

函 数	操 作
sin(x)	x(弧度值)的正弦
cos(x)	x 的余弦
tan(x)	x 的正切

如果使用三角法，这些函数的用法非常简单。下面是一些例子：

```
double angle = 45.0;                // Angle in degrees
double pi = 3.14159265;
double sine = 0.0;
double cosine = 0.0;
sine = sin(pi*angle/180.0);         // Angle converted to radians
cosine = sin(pi*angle/180.0);       // Angle converted to radians
```

180° 等于 1 弧度，所以以度数表示的角度除以 180，再乘以 PI 的值，就得到其弧度值，这些函数都要求使用弧度值。

还可以使用反三角函数：asin()、acos()和atan()，以及双曲线函数 sinh()、cosh()和 tanh()。如果要使用这些函数，必须在程序中包含 math.h 头文件。如果不需要使用这些函数，就可以跳过本节。

2.14 设计一个程序

下面设计本章末的一个真实例子，来试用一些数值类型。这里将从头开始编写一个程序，涉及编程的所有基本要素，包括问题的初始描述、问题的分析、解决方案的准备、编写程序、运行程序，以及测试它，确保它正常工作。该过程的每一步都会引入新问题，而不仅仅是纸上谈兵。

2.14.1 问题

许多人都对树的高度很感兴趣。如果将树砍倒，量出它的高度，就可以确定离树多远才是安全的。这对于患有神经衰弱的人来说非常重要。问题是如何不使用非常长的梯子，就可以确定树的高度，因为长梯也会对人和树枝带来危险。为了确定树的高度，可以向朋友求助，最好找一个个子比较矮的朋友，除非自己比较矮，此时需要一个个子比较高的朋友。假定要测量的树比自己和朋友都高。比自己还矮的树很容易测量出其高度，除非这棵树长满了刺。

2.14.2 分析

现实问题很少能用适合于编程的方式来表达。在编写代码之前，需要确保完全理解了问题及其解决方式。只有这样，才能估计出创建解决方案所需的时间和精力。

分析阶段应增强对问题的理解，确定解决它的逻辑过程。一般这需要大量的工作，这包括找出问题阐述中模糊或遗漏的细节。只有全面理解了问题，才能开始以适合编程的形式表达解决方案。

我们打算用一个简单的图形和两个人(一高一矮)的身高来确定树的高度。首先给高个子命名为 Lofty，矮个子命名为 Shorty。为了得到比较精确的结果，高个子应明显高于矮个子。否则高个子可以考虑站在一个箱子上。图 2-3 给出了解决这个问题的思路。

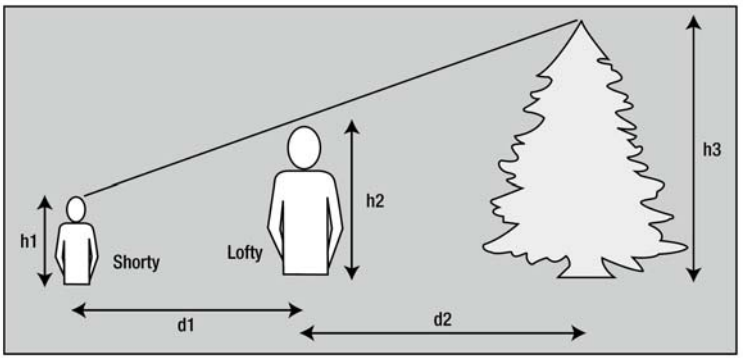

图 2-3 树的高度

确定树的高度是很简单的。如果知道图中 h1 和 h2 的值(它们分别是 Shorty 和 Lofty 的高度)以及 d1 和 d2(它们分别是 Shorty 与 Lofty 之间的距离和 Lofty 与树之间的距离)，就可以计算出树的高度。使用相似三角形的特性就可以求出树的高度，如图 2-4 所示。

因为三角形是相似的，所以 height1:distance1=height2:distance2。使用这个关系，就可以通过 Shorty 和 Lofty 的身高，以及他们与树之间的距离求出树的高度，如图 2-5 所示。

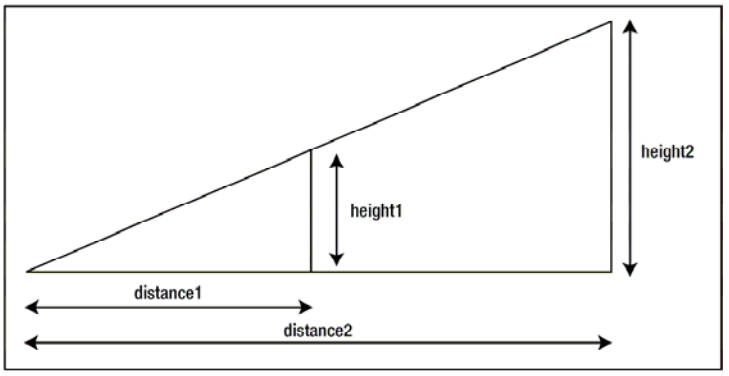

图 2-4 相似三角形

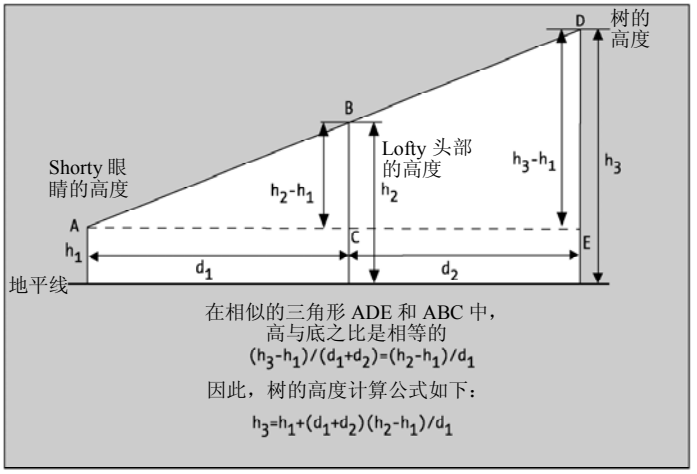

图 2-5 计算树的高度

三角形 ADE 和 ABC 与图 2-4 相同。由于这两个三角形相似，则一个三角形任意一边的长度除以另一个三角形的对应边长度，结果总是相等的。所以可以使用图 2-5 底部的等式计算出树的高度。

这说明，在程序中，可以使用如下 4 个值计算出树的高度：

- Shorty 与 Lofty 之间的距离，即图中的 d1。用 shorty_to_lofty 变量存储这个值。
- Lofty 与树之间的距离，即图中的 d2。用 lofty_to_tree 变量存储这个值。
- 从地平线到 Lofty 头部的高度，即图中的 h2，用 lofty 变量存储这个值。
- 从地平线到 Shorty 眼睛的高度，即图中的 h1，用 shorty 变量存储这个值。

接着，把这些值放在计算树高的等式中。首先要把这 4 个值输入计算机。接着使用其比值计算出树的高度，最后输出答案。步骤如下：

(1) 输入需要的值。

(2) 使用图 2-5 中的等式计算树的高度。

(3) 显示答案。

2.14.3 解决方案

本节列出解决问题的步骤。

1. 步骤 1

第一步获取计算树高需要的值。这意味着必须包含 stdio.h 头文件，因为需要使用 printf()和 scanf()函数。接着确定存储这些值的变量。之后，就可以使用 printf()提示输入数字，使用 scanf()从键盘上读取值。

为了方便用户，把高个子和矮个子的身高输入为英尺英寸值。但在程序中，高度和距离使用相同的单位会更方便，所以应将所有的数字都转换为英寸值。我们需要两个变量存储 Shorty 和 Lofty 的身高(英寸值)，还需要一个变量存储 Shorty 和 Lofty 之间的距离，需要另一个变量存储 Lofty 与树之间的距离，当然，这两个距离值都以英寸为单位。

在输入过程中，首先将 Lofty 的身高输入为一个整数英尺值和一个英寸值，在此过程中要提示用户输入每个值。为此可以使用另外两个变量，一个存储英尺值，另一个存储英寸值。接着把它们转换为英寸值，将结果存储在为 Lofty 身高保留的变量中。对 Shorty 的身高进行相同的处理(但只输入从地平线到 Shorty 眼睛的高度)，最后处理他们之间的距离。对于 Lofty 与树之间的距离，可以只使用整数英尺值，因为这已经足够准确了——还要把距离转换为英寸值。对于每个输入的英尺值和英寸值，可以使用相同的变量。所以下面是程序的第一部分：

```
// Program 2.17 Calculating the height of a tree
#include <stdio.h>

int main(void)
{
  long shorty = 0L;             // Shorty's height in inches
  long lofty = 0L;              // Lofty's height in inches
  long feet = 0L;
  long inches = 0L;
  long shorty_to_lofty = 0L;    // Distance from Shorty to Lofty in inches
  long lofty_to_tree = 0L;      // Distance from Lofty to the tree in inches
  const long inches_per_foot = 12L;

  // Get Lofty's height
  printf("Enter Lofty's height to the top of his/her head, in whole feet: ");
  scanf("%ld", &feet);
  printf("             ...and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  // Get Shorty's height up to his/her eyes
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;
```

```
  // Get the distance from Shorty to Lofty
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                        ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  // Get the distance from Lofty to the tree
  printf("Finally enter the distance from Lofty to the tree to the nearest foot: ");
   scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  // The code to calculate the height of the tree will go here

  // The code to display the result will go here
  return 0;
}
```

注意，代码进行了缩进，以便于阅读。这不是必须的，但如果要在未来修改程序，这么做更便于确定程序的工作方式。应总是给程序添加注释，以帮助理解程序。至少要清楚地说明变量的用途，解释程序的基本逻辑。

使用一个声明为 const 的变量将英尺转换为英寸。该变量的名称是 inches_per_foot，说明了它在代码中使用时会发生什么。这要比明确使用 12 这个数字好得多。这里处理的是英尺和英寸，大多数人都知道，12 英寸是 1 英尺。但在其他环境下，数值常量的重要性没有这么明显。如果在计算薪水的程序中使用 0.22，它的含义就不是很明显。因此，这个计算相当难理解。如果创建一个 const 变量 tax_rate，把它初始化为 0.22，就不会有理解障碍了。

2. 步骤 2

有了需要的所有数据后，就可以计算树的高度了。只需利用变量的值，实现计算树高的等式即可。这里需要声明另一个变量来存储树的高度。

为此，添加如下粗体的代码：

```
// Program 2.18 Calculating the height of a tree
#include <stdio.h>

int main(void)
{
  long shorty = 0L;                 // Shorty's height in inches
  long lofty = 0L;                  // Lofty's height in inches
  long feet = 0L;
  long inches = 0L;
  long shorty_to_lofty = 0L;        // Distance from Shorty to Lofty in inches
  long lofty_to_tree = 0L;          // Distance from Lofty to the tree in inches
  long tree_height = 0L;            // Height of the tree in inches
  const long inches_per_foot = 12L;

  // Get Lofty's height
  printf("Enter Lofty's height to the top of his/her head, in whole feet: ");
  scanf("%ld", &feet);
```

```
  printf("                                              ...and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  // Get Shorty's height up to his/her eyes
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                       ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  // Get the distance from Shorty to Lofty
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                        ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  // Get the distance from Lofty to the tree
  printf("Finally enter the distance from Lofty to the tree to the nearest foot: ");
  scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  // Calculate the height of the tree in inches
  tree_height = shorty + (shorty_to_lofty + lofty_to_tree)*(lofty-shorty)/
                                                              shorty_to_lofty;

  // The code to display the result will go here
  return 0;
}
```

计算树高的语句与图中的等式相同。这有点繁琐，但直接转换为程序中的语句，以计算树高。

3. 步骤 3

最后，输出答案。为了以最容易理解的形式显示结果，应把存储在 tree_height 中的结果(英寸值)转换为英尺和英寸值：

```
// Program 2.18 Calculating the height of a tree
#include <stdio.h>

int main(void)
{
  long shorty = 0L;              // Shorty's height in inches
  long lofty = 0L;               // Lofty's height in inches
  long feet = 0L;
  long inches = 0L;
  long shorty_to_lofty = 0L;     // Distance from Shorty to Lofty in inches
  long lofty_to_tree = 0L;       // Distance from Lofty to the tree in inches
  long tree_height = 0L;         // Height of the tree in inches
  const long inches_per_foot = 12L;

  // Get Lofty's height
  printf("Enter Lofty's height to the top of his/her head, in whole feet: ");
```

```
  scanf("%ld", &feet);
  printf("                                              ... and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  // Get Shorty's height up to his/her eyes
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                        ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  // Get the distance from Shorty to Lofty
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                          ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  // Get the distance from Lofty to the tree
  printf("Finally enter the distance from Lofty to the tree to the nearest foot: ");
  scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  // Calculate the height of the tree in inches
  tree_height = shorty + (shorty_to_lofty + lofty_to_tree)*(lofty-shorty)/
                                                                shorty_to_lofty;
  // Display the result in feet and inches
  printf("The height of the tree is %ld feet and %ld inches.\n",
                    tree_height/inches_per_foot, tree_height% inches_per_foot);
return 0;
}
```

程序的输出如下：

```
Enter Lofty's height to the top of his/her head, in whole feet first: 6
                                                ... and then inches: 2
Enter Shorty's height up to his/her eyes, in whole feet: 4
                                    ... and then inches: 6
Enter the distance between Shorty and Lofty, in whole feet : 5
                                          ... and then inches: 0
Finally enter the distance to the tree to the nearest foot: 20
The height of the tree is 12 feet and 10 inches.
```

2.15 小结

本章介绍了许多基础知识，讨论了 C 程序的构建方式、各种算术运算、如何选择合适的变量类型等。除了算术运算之外，还学习了输入输出功能，通过 scanf()将值输入变量，通过 printf()函数把文本、字符值和数值变量输出到屏幕上。读者可能不能第一次就掌握所有这些内容，但可以在需要时复习本章。

下一章将开始学习如何根据输入值做出判断，控制程序的执行。这是创建有趣且专

业化程序的关键。

表 2-13 总结了前面介绍的变量类型。在学习本书的过程中，可以随时复习这些内容。

表 2-13 变量类型和值域

类 型	字 节 数	值 域
char	1	－128~+127 或 0~+255
unsigned char	1	0~+255
short	2	－32 768~+32 767
unsigned short	2	0~+65,535
int	4	－32 768~+32767 或－2 147 438 648~+2 147 438 647
unsigned int	4	0~+65 535 或 0~+4 294 967 295
long	4	－2 147 438 648~+2 147 438 647
unsigned long	4	0~+4 294 967 295
long long	8	－9 223 372 036 854 775 808 到+9 223 372 036 854 775 807
unsigned long long	8	0~+18 446 744 073 709 551 615
float	4	±3.4E±38(6 位)
double	8	±1.7E±308(15 位)
long double	12	±1.2E±4932(19 位)

本章还介绍并使用了 printf()函数的数据输出格式说明符，完整的说明符列表请参见附录 D。附录 D 还描述了输入格式说明符，它们用于控制使用 scanf()函数从键盘上读取数据时这些数据的解释方式。当无法确定如何处理输入或输出数据时，可以参阅附录 D。

2.16 练习

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 2.1 编写一个程序，提示用户用英寸输入一个距离，然后将该距离值输出为码、英尺和英寸的形式(12 英寸是 1 英尺，3 英尺是 1 码)。

习题 2.2 编写一个程序，提示用户用英尺和英寸输入一个房间的长和宽，然后计算并输出面积，单位是平方码，精度为小数点后有两位数。

习题 2.3 一个产品有两种版本：其一是标准版，价格是$3.5，其二是豪华版，价格是$5.5。编写一个程序，使用学到的知识提示用户输入产品的版本和数量，然后根据输入的产品数量，计算并输出价格。

习题 2.4 编写一个程序，提示用户从键盘输入一个星期的薪水(以美元为单位)和工作时数，它们均为浮点数，然后计算并输出每个小时的平均薪水，输出格式如下所示：

```
Your average hourly pay rate is 7 dollars and 54 cents.
```