O'REILLY®

异步图书
www.epubit.com.cn

第2版

C程序设计新思维

21st Century C

[美] 本·克莱蒙 (Ben Klemens) 著
赵岩 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

目　　录

版权信息

书名：C程序设计新思维（第2版）

ISBN：978-7-115-46095-0

- 著　　[美] 本·克莱蒙(Ben Klemens)

  译　　赵　岩

  责任编辑　胡俊英

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

版权声明

内容提要

C语言已经有几十年的历史了。经过长时间的发展和普及，C语言的应用场景有了很大的变化，一些旧观念应该被淡化或者不再被推荐。

本书展现了传统C语言教科书所不具有的最新的相关技术。全书分为开发环境和语言两个部分，从编译、调试、测试、打包、版本控制等角度，以及指针、语法、文本、结构、面向对象编程、函数库等方面，对C程序设计的核心知识进行查缺补漏和反思。本书鼓励读者放弃那些对大型机才有意义的旧习惯，拿起新的工具来使用这门与时俱进的简洁语言。

本书适合有一定基础的C程序员和C语言学习者阅读，也适合想要深入理解C语言特性的读者参考。

译者序

最近非常有幸地接受了人民邮电出版社的邀请来翻译本书，这是一本非常经典的C语言著作，目前已经是第2版了。计算机图书出了很多年，大家对其自有判断，最简单的办法就是根据书名，20世纪90年代末期出版过几本比较经典的计算机图书，书名为：《**入门到精通》《21天学会**》等。不过很快大家就开始借用这种书名，最后搞得有些良莠不齐。更有甚者，最近出现了好多类似《**从入门到放弃》《**从入门到入院》的图书，彻底颠覆了以前程序员心目中这么神圣的书名。好在还有O’Reilly出版社的动物丛书，目前还都是品质和经典的象征。这套书经典到有时圈内的人们都忘了书名，只记得动物的名字，例如Perl语言的 “骆驼书”以及Git的“蝙蝠书”等。

当完成最后一个字的录入，作为本书的译者，我认为应该系统地给这本书做个总结了。首先，这是一本经典的C语言图书，亚马逊上有50多条评论，评分达到4分。这个傲人的成绩主要来自于本书的两个优点：第一个优点就是系统性和大局观。C语言最开始作为开发UNIX操作系统的工具，它和UNIX操作系统有着不可分割的关系。无论UNIX派生的POSIX标准以及GNU运动，C语言都是其核心的开发语言和工具。所以如果想真正发挥C语言的威力，那必须要把这个语言放到一个更大的生态环境中去。本书通过对POSIX标准库、GNU编译器、Shell 脚本、Make、Git、文档和测试，以及各种常用的函数库等一系列内容的介绍，建立了一个高效整合的C语言开发环境。C语言作为这个环境的核

心开发语言，通过各种开发工具和函数库的配合，将开发环境的优点淋漓尽致地发挥出来，从而能显著地提高你的开发效率。

第二个优点就是新思维和反规则。作为物理学的爱好者，我用物理来做一个类比 。牛顿创立了经典力学和万有引力。正当我们认为物理学已经完胜的时候，爱因斯坦在一边幽幽地说的那句“光会拐弯”。在爱因斯坦的结论在观测日全食方面得到验证以后，这位天才自信心爆棚并宣称“一切都是可以通过计算来确定的”。这个时候研究量子力学的波尔却传给了他一个纸条说：“上帝掷骰子！”也许，我是说也许，我们一直都相信的规则或者答案过于片面。 就像有一天我6岁的女儿小米粒问我：“我们人类是从哪里来的啊？”我说：“有人说是猴子变的，有人说是神创造的，你信哪种说法都可以。你告诉爸爸，你信哪种啊？”我的女儿想都没想就回答到：“是神把猴子变成人的！”

我们人类总是有一种倾向，一旦形成了自己的某些规则，那么就会自然地排斥和否定另外的反规则。而本书的难能可贵之处就在于，它不仅提出了C语言的一些反规则，而且通过一些例子证明这些反规则是合理的。例如，我们可以建造高效和准确的宏，可以不需要对内存的使用斤斤计较，哪怕有点内存泄漏，我们可以用goto，但是对switch却完全可以放弃，等等。现代物理有一个反物质学说，当物质和反物质相遇时，二者会立即湮没，并爆发出巨大的能量。这里我借用一下：当你熟悉了规则，同时也理解了反规则，这个时候你的心中就没有了规则。剩下的就是巨大的能力。此时小李飞刀已经不带刀，此时无招已经胜有招。

俗话说：“优点不说没不了，缺点不说不得了。”下面说说本书的缺

点，那就是：对每一部分的内容并没有详细地介绍！所以你不要指望着阅读完本书就能熟练地使用shell脚本，写出复杂的makefile并通过Git高效地与人协作。坦白地说，这也并不算是缺点。本书的目的就是告诉你，当你想做什么事情的时候，有哪些工具你可以用，而这些工具的最常见的用法又是什么。当你发现这些工具并不够用的时候，你可以去找专门的介绍相关内容的书。这个时候你会发现完整介绍某些工具的书，它的厚度足以挡住狙击步枪射出的子弹。同时，POSIX体系内的东西，有臭名昭著的学习曲线。你要是不服就下载一个VIM编辑器试试看！

从安装VIM到现在，我一直在使用它，但我一直搞不懂如何退出VIM。大家都说C语言难，客观地来讲，这个锅不能让C语言和本书来背。但是，你想要“会当凌绝顶，一览众山小”，那你的学习曲线必须要很陡才行！

另外，本书的第3.3节“用Autotools打包代码”，我个人认为这个技术有些过时。目前流行的build system 是CMake系统，它在移植性和易用性上都要比Autotoolson工具要好。所以CMake是新开发系统的首选build system，也是未来的潮流。

最后说说本书面向的读者对象。首先，本书并不是教材，虽然书的后面有一个简短的介绍C语言的附录，我没有骗你，它确实很简短。请注意我的用词，我只是说“简短”，并没有说“简单”。所以，如果你是一个C语言的初学者或者是零基础读者，那么本书并不适合你。

本书面向的读者对象是有一定C语言基础的高年级学生，或者是一些使用C语言作为主要开发语言的工程师。对于大学高年级的学生，它们缺乏的是一种对大的编程环境的认识。而对于使用C语言的从业人

员，本书会让你对C语言有不一样的认识。它对你多年使用C语言形成的习惯和风格提出了挑战，让你有一种“原来C语言也可以这么用”的赞叹！然后让这些反规则去湮没你心中存在多年的规则，从而爆发出巨大的能量！

最后再提醒一句，本书真的不适合初学者！否则，你看不懂会说我翻译得不好。我已经把$E=MC^2$翻译成了“能量等于质量乘以光速的平方”，如果你还看不懂，就别怪译者了。

举个简单的例子，书中8.2节最后一段是：

这意味着如果两个文件中的两个变量有共同的名字，但是你想它们应该彼此独立的。这个时候如果你忘记static关键字，编译器可以把有外部链接变量的链接成一个变量。这种细微的bug非常容易发生，所以对于那些有内部链接的变量不要忘了使用static。

如果你不了解tentative definition(临时定义)这一C语言中特有的现象，就不能充分理解“这种细微的bug”究竟是什么。所以为了让读者能更容易地阅读本书，我为本书做了一个网站: http://zhaoyan.website/xinzhi/c21/book.php，里面有我对每一章的观点，同时推荐了一些帮助读者理解的补充内容等。例如，对于上面的tentative definition的问题，我给出了非常好的介绍，同时还有我自己编写的一段程序对这一概念进行了说明。对于本书的每部分内容过于简短和艰深这一缺点，我也做了一些有益的补充和修正。

最后我想说的是：这本书真的很好！这本书真的很难！你有勇气挑战一下吗？

——赵岩

2017年5月

前言

C就是Punk Rock

虽然C仅有为数不多的关键词，并且没有那么多细节修饰，但是它很棒[1]！你可以用C来做任何事情。它就像一把吉他上的C、G和D弦，你很快就可以掌握其基本原理，然后就得用你的余生来提高。不理解它的人害怕它的威力，并认为它粗糙得不够安全。虽然没有企业和组织花钱去推广它[2]，但是实际上它在所有的编程语言排名中一直被认为是最流行的语言。

这门语言已经有几十年的历史了，可以说已经进入了中年。创造它的是少数对抗管理阶层并遵从完美的punk rock精神的人；但那是20世纪70年代的事情了，现在这门语言已经历尽沧桑，并且成为主流的语言。

当punk rock变成主流的时候人们会怎样？在其从20世纪70年代出现后的几十年里，punk rock已经从边缘走向中心：The Clash、The Offspring、Green Day和The Strokes等乐队已经在全世界卖出了几百万张唱片（这还只是一小部分），我也在家附近的超市里听过被称为grunge的一些精简乐器版本的punk rock分支。Sleater-Kinney乐队的前主唱还经常在自己那个很受欢迎的喜剧节目中讽刺punk rocker音乐人[3]。对这种持续的进化，一种反应是划一条界限，将原来的风格称为punk rock，而将其余的东西称为面向大众的粗浅的punk。传统主义者还是可以播放20

世纪70年代的唱片，但如果唱片的音轨磨损了，他们可以购买数码版本，就像他们为自己的小孩购买Ramones牌的连帽衫一样。

外行是不明白的。有些人听到punk这个词时脑海里就勾画出20世纪70年代特定的景象，经常的历史错觉就是那个时代的孩子们真的在做什么不同的事情。喜欢欣赏1973年Iggy Pop 的黑胶唱片的传统主义者一直是那么兴趣盎然，但是他们有意无意地加强了那种punk rock已经停滞不前的刻板印象。

回到C的世界里，这里既有挥舞着ANSI’89标准大旗的传统主义者，也有那些拥抱变化，甚至都没有意识到如果回到20世纪90年代，他们写的代码都不可能被成功编译与运行的人。外行人是不会知道个中缘由的。他们看到从20世纪80年代起至今还在印刷的书籍和20世纪90年代起至今还存于网上的教程，他们听到的都是坚持当年的软件编写方式的、死硬的传统主义者的言论，他们甚至都不知道语言本身和别的用户都在一直进化。非常可惜，他们错过了一些好东西。

这是一本打破传统并保持C语言punk精神的书。我对将本书的代码与1978年Kernighan和Ritchie出版的书[4]中的C标准进行对比毫无兴趣。既然连我的电话都有512MB内存，为什么还在我的书里花费章节讲述如何为可执行文件减少几千字节呢？我正在一个廉价的红色上网本上写这本书，而它却可以每秒运行3 200 000 000条指令，那为什么我还要操心8位和16位所带来的操作的差异呢？我们更应该关注如何做到快速编写代码并且让我们的合作者们更容易看懂。毕竟我们是在使用C语言，所以我们那些易读但是并没有被完美优化的代码运行起来还是会比很多烦琐的语言明显更快。

Q&A（本书的参考引用）[5]

问题：这本书与其他书有什么不同？

答案：有些书写得好，有些书写得有趣，但是大部分C语言的教科书都非常相像（我曾经读过很多这样的教科书，包括[Griffiths，2012]、[Kernighan，1978]、[Kernighan，1988]、[Kochan，2004]、[Oualline，1997]、[Perry，1994]、[Prata，2004]和[Ullman，2004]。多数教材都是在C99标准发布并简化了很多用法之后写成的，你可以看到现在出版的这些教材的第N版仅仅在一些标注上做了一点说明，而不是认真反思了如何使用这门语言。它们都提到你可以拥有一些库来编写你自己的代码，但是书籍完成时，缺少了保障库的可靠性和可移植性的安装与开发环境。那些教科书现在仍然有效并且具有自己的价值，但是现代的C代码已经看起来和那些教科书里面的不太一样了。

这本书与那些教科书的不同之处，在于对这门语言及其开发环境进行了拾遗补漏。书中讲解的方式是：直接使用提供了链表结构和XML解析器的现成的库，而不是把这些从头再写一次。这本书也体现了如何编写易读代码和用户友好的函数接口。

问题：这本书的目标读者是谁？我需要是一个编程大师吗？

答案：你必须有某种语言的编程经验，或许是Java，或者是类似于Perl的某种脚本语言。这样我就没有必要再向你讲为什么你不应该写一个很长的没有任何子函数的函数了。

本书的内容假设你已经有了通过写C代码而获得的C语言的基本知

识。附录A提供了一个简短的有关C语言基础的教程，那些以前写Python和Ruby等脚本语言的读者可以阅读它。

请允许我介绍我写的另一本关于统计和科学计算的教科书*Modeling with Data* [Klemens，2008]。那本书不仅提供了很多关于如何处理数值和统计模型的内容，它还可以用作一本独立的C语言的教材，并且我认为那本书还避免了很多早期教材的缺点。

问题：我是个编写应用软件的程序员，不是一个研究操作系统内核的人。为什么我应该用C而不是像Python这类可以快速编程的脚本语言呢？

答案：如果你是一个应用软件程序员的话，这本书就是为你准备的。我知道人们经常认定C是一种系统语言，这让我觉得真是缺少了点punk的反叛精神——他们是谁啊？要他们告诉我们要用什么语言？

像“我们的语言几乎和C一样快，但更容易编写”这样的言论很多，简直成了陈词滥调。好吧，C显然是和C一样快，并且这本书的目的是告诉你C也像以前的教科书所暗示的那样容易使用。你没必要使用malloc，也没必要像20世纪90年代的系统程序员那样深深卷入内存管理，我们已经有处理字符串的手段，甚至核心语法也进化到了支持更易读的代码的境界。

我当初正式学习C语言是为了加速一个用脚本语言R编写的仿真程序。和众多的脚本语言一样，R具有C接口并且鼓励用户在宿主语言[6]太慢的时候使用。最终我的程序里有太多的从R语言到C语言的调用，最后我索性放弃了宿主语言。随后发生的事情你已经知道，就是我在写

这本关于现代C语言技术的书。

问题：如果原本使用脚本语言的应用软件程序员能喜欢这本书当然好，但我是一名内核黑客。我在五年级的时候就自学了C语言，有时做梦都在正确编译。那这本书还有什么新鲜的吗？

答案：C语言在过去的几年里真的进化了很多。就像我下面要讨论的那样，各个编译器对新功能的支持的时间也不一样，感谢自从ANSI标准发布后，又发布了两个新的C语言标准。也许你应该读一下第10章，找找有什么能叫你感到惊讶的。本书的一部分，如讲解指针的经常被人错误理解的一些概念（第6章），也覆盖了自从1980年以后变化的内容。

并且，开发环境也升级了。很多我提到的工具，如make和debugger，你已经很熟悉了，但是我发现别人可能还不知道。Autotools已经改变了代码发布的方式，Git也改变了我们合作编程的方式 。

问题：我实在忍不住要问，为什么这本书中有差不多三分之一的篇幅都没有C代码？

答案：这本书本意就是讲述一些其他C语言教科书没有讲到的内容，排在首位的就是工具和环境。如果你没有使用调试器（独立的或者集成在你的IDE中），你就是在自讨苦吃。教科书经常把debugger放到最后面，有的根本就不提。与他人共享代码也需要另外的工具集，如Autotools和Git。代码并不存在于真空里，其他的教科书都在假设读者只需要了解C语言语法就会有生产力了，那就让我写一点不同于这些教科书的内容吧。

问题：有太多的用于C开发的工具，你在本书中如何取舍呢？

相比大部分语言，C语言社区有更高的内部互通性。GNU提供了太多的C语言扩展，还有那些只工作在Windows平台的IDE，只存在于LLVM中的编译器扩展等。这就是为什么过去的教科书不去讲解工具的原因。但是现在，有些系统应用得非常普遍。很多工具来自于GNU；LLVM和相关工具虽然不是主流，但是也打下了坚实的基础。不管你用什么，Windows、Linux或者你从你的云计算提供商那里取得的任何东西，这里我介绍的工具全部是容易并且可以快速地安装的。我提到了一些平台相关的工具，但是仅限那么几例。

我并没有介绍集成开发环境（IDEs），因为很少的集成开发环境能跨平台工作（尝试建立一个Amazon Elastic Computer Cloud实例，然后在上面安装Eclipse和它的C插件），而且IDEs的选择大部分被个人的喜好所左右。IDE有一个项目建造系统，它通常与别的IDE的项目建造系统不兼容。IDE的项目文件在你分发到外面的时候就不能用了。除非你硬性规定所有的人（在教室、特定办公室或者某些计算平台上）都必须使用相同的IDE。

问题：我能上网，一两秒的功夫就能找到命令和语法的细节。那么说真的，为什么我还要读这本书？

答案：的确。在Linux或Mac机器上你只要用一个带有 man operator 的命令行就能查到运算符优先级表，那么我为什么还要把它放在这本书里？

我可以和你上同样的Internet，我甚至花了很多的时间阅读网上的内

容。所以我有了一个之前没谈到的、准备现在讲的好主意：当介绍一个新工具的时候，如gprof或者GDB，我给你那些你必须知道的方向，然后你可以去自己习惯的搜索引擎中查找相关问题。这也是其他教科书所没有的（这样的内容还不少呢）。

标准：难以抉择

除非特别地说明，本书的内容遵从ISO C99和C11标准。为了使你明白这意味着什么，下面给你介绍一点C语言的历史背景，让我们回顾一下主要的C标准（而忽略一些小的改版和订正）。

K&R（1978前后）

Dennis Ritchie、Ken Thompson以及一些其他的贡献者发明了C语言，并编写了UNIX操作系统。Brian Kernighan和Dennis Ritchie最终在他们的书中写下了第一版关于这个语言的描述，同时这也是C语言的第一个事实上的标准[Kernighan，1978]。

ANSI C89

后来Bell实验室向美国国家标准协会（ANSI）交出了这个语言的管理权。1989年，ANSI出版了他们的标准，并在*K&R*的基础上做出了一定的提高。*K&R*的书籍的第2版包含了这个语言的完整规格，也就是说在几万名程序员的桌子上都有这个标准的印刷版[Kernighan，1988]。1990年，ANSI标准被ISO基本接受，没有做重大的改变，但是人们似乎更喜欢用ANSI，89这个词来称呼这个标准（或者用来做很棒的T恤衫标语）。

10年过去了。C成为了主流，考虑到几乎所有的PC、每台Internet服务器的基础代码或多或少都是用C编写的，C语言已经成为了主流，这已经是人类的努力可以达到的最大的极限了。

在此期间，C++分离出来并大获成功（虽然也不是那么大）。C++是C身上发生的最好的事情了。当所有其他的语言都在试图添加一些额外的语法以跟随面向对象的潮流，或者跟随其作者脑袋里的什么新花招的时候，C就是恪守标准。需要稳定和可移植性的人使用C，需要越来越多的人把大量的金钱投入到了C++语言上，这样的结果就是：你好我好，大家过年，每个人都高兴。

ISO C99

10年之后，C标准经历了一次主要的改版。为数值和科学计算增添了一些附加功能，如复数的标准数据类型以及泛型(type-generic)函数。一些从C++中产生的便利措施被采纳，包括单行注释（实际上起源于C的前期语言，BCPL），以及可以在for循环的开头声明变量。因为一些新添加的关于如何声明和初始化的语法，以及一些表示法上的便利，使用泛型函数变得更加容易。出于安全考量以及并不是所有人都说英语原因，一些特性也被做了调整。

当你想着单单C89的影响其实并不大，以及全球大范围地运行着C代码时，你就理解了ISO做出的任何改变都是会被广泛批评的——甚至你不做任何改变，别人还想找茬骂你呢[7]。的确，这个标准是有争论的。有两种常用的方式来表达一个复数（直角坐标和极坐标）——那么ISO会采用哪一个？既然所有的好代码都没采用变长的宏输入机制来编写，为什么我们还需要这个机制？换句话说，纯洁主义者批评ISO是屈

服于外界的压力才给C语言增加了更多的特性。

当我写这本书的时候，多数的编译器在支持C99的同时都增加或减少了一些特性；如long double 类型看起来就引发了很多问题。然而，这里还是有一个明显的特例：Microsoft至今拒绝在其Visual Studio C++编译器中添加C99支持。在本书第6页“1.2 在Windows下编译C”一节中讲述了几种在Windows环境中编译C的方法，所以不能使用Visual Studio最多也就是有点不方便，这好比一个行业奠基人告诉我们不能使用ISO标准的C，这样标准就更有punk rock风格了。

C11

觉察到了对所谓背叛行业趋势的批评后，ISO组织在第三版的标准中做出了为数不多的几个重大改变。我们有了可以编写泛型函数的方法，并且对安全性和非英语支持做出了进一步的改进。

C11标准在2011年12月发布后不久，编译器的开发者以惊人的速度完成了对新标准的支持。目前一些主流编译器已经声称做到了几乎全部的标准兼容。但是标准定义了编译器的行为，也定义了标准库和库支持。如线程和原子性等，有些系统上实现了，有些系统上还在开发。

POSIX标准

事物的规律就是这样，伴随着C语言的进化，这门语言同时也和UNIX操作系统协同发展，并且你将会从本书中看到，这种相互协同的发展对日常工作是有意义的。如果某件事情在UNIX命令行中很容易利用，那么很有可能是因为这件事情在C中也很容易实现；某些UNIX工具之所以存在，也是为了帮助C代码的编写。

UNIX

C和UNIX都是在20世纪70年代由Bell实验室设计的。在20世纪的多数时间里，Bell一直面临垄断调查，并且Bell有一项与美国联邦政府达成的协议，就是Bell将不会把自身的研究扩张到软件领域。所以UNIX被免费发放给学者们去研究和重建。UNIX这个名字是一个商标，原本由Bell实验室持有，但随后就像一张棒球卡一样在数家公司之间转卖。

随着其代码被不断研究、重新实现，并被黑客们以不同的方式改进，UNIX的变体迅速增加。因此带来了一点不兼容的问题，即程序或脚本变得不可移植，于是标准化工作的迫切性很快就变得显而易见。

POSIX

这个标准最早由电气和电子工程师协会（IEEE）在1988年建立，提供了一个类UNIX操作系统的公共基础。它定义的规格中包括shell脚本如何工作，像ls、grep之类的命令行应该如何工作，以及C程序员希望能用到的一些C库等。举个例子，命令行用户用来串行运行命令的管道机制被详细地定义了，这意味着C语言的popen（打开管道）函数是POSIX标准，而不是ISO C标准。POSIX标准已经被改版很多次了；本书编写的时候是POSIX:2008标准，这也是当我谈到POSIX标准的时候所指代的。POSIX标准的操作系统必须通过提供C99命令来提供C编译器。

这本书用到POSIX标准的时候，我会告诉大家。

除了来自Microsoft的一系列操作系统产品，当前几乎所有你可以列举出的操作系统都是建立在POSIX兼容的基础上：Linux、Mac OS X、iOS、WebOS、Solaris、BSD——甚至Windows Servers也提供POSIX子

系统。对于那些例外的操作系统，1.2“在Windows下编译C程序”一节将告诉你如何安装POSIX子系统。

最后，有两个POSIX的实现版本因为有较高的流行度和影响力，值得我们注意。

BSD

在Bell实验室发布UNIX给学者们剖析之后，加州大学伯克利分校的一群好人做了很多明显的改进，最终重写了整个UNIX基础代码，产生了伯克利软件发行版（Berkeley Software Distribution，BSD）。如果你正在使用一台Apple公司生产的电脑，你实际上在使用一个带有迷人图形界面的BSD。BSD在几个方面超越了POSIX，因此我们还会看到，有一两个函数虽然不属于POSIX，但是如此有用而不容忽略（其中最重要的救命级函数是asprintf）。

GNU

GNU这个缩写代表GNU’s Not UNIX，代表了另一个独立实现和改进UNIX环境的成功故事。大多数的Linux发行版使用GNU工具。有趣的是，你可以在你的POSIX机器上使用GNU编译器组合（GNU Compiler Collection，gcc）——甚至BSD也用它。并且，gcc对C和POSIX的几个方面做了一点扩充并成为事实上的标准，当本书中需要使用这些扩充的时候我会加以说明。

从法律意义上说，BSD授权比GNU授权稍微宽容。由于很多群体对这些授权的政治和商业意义深感担心，实际上你会经常发现多数工具同时提供GNU和BSD版本。例如，GNU 的编译器组合（gcc）和BSD的clang都可以说是顶级的C编译器。来自两个阵营的贡献者紧密跟随对方

的工作，所以我们可以认为目前存在的差异将会随着时间逐渐消失。

> 法律解读
>
> 美国法律不再提供版权注册系统：除了很少的特例，只要某人写下什么就自然获得了该内容的版权。
>
> 发行某个库必然要通过将其从一个硬盘复制到另一个硬盘这样的操作，而且即便带有一点争议，现实中还是存在几种常用机制允许你有权利复制一个有版权的内容。
>
> - GNU公共许可证：其允许无限制地复制和使用源代码和可执行文件。不过有一个前提：如果你发行一个基于GPL许可证的源代码程序或库，你也必须将你的程序的源代码伴随程序发行。注意，如果你是在非商业环境下使用这样的程序，你不需要发行源代码。像用gcc编译你的源代码之类的运行GPL许可证的程序本身并不会使你具有发行源代码的义务，因为这个程序的数据（比如你编译出的可执行文件）并不认为是基于或派生于gcc的。例如：GNU科学计算库。
> - 次级GPL许可证：与GPL有很多相似，但是具有一个明显的区别：如果你以共享库的方式连接一个LGPL库，你的代码不算作派生的代码，也没有发行源代码的义务。也就是说，你可以采用不暴露源代码的方式发行一个与LGPL库连接的程序。例如：Glib。
> - BSD许可证：要求使用者维持BSD授权原始码原有的版权声明和免责声明，但不要求同时提供你的原始码。

请注意以下的免责声明：笔者不是律师，这段小常识只是完整法律文件的简单声明，读者如果无法判断自身所处的状况或者相关细节，请阅读原始文件或者请教律师。

附加内容

第2版

我以前是一个愤世嫉俗主义者，认为如果你写了第2版，那么你的主要目的就是让那些卖你第1版二手书的人不开心。本书的第2版如果没有第1版被发表的话，是不会，也不可能这么快的就出版的。（反正现在很多读者都在阅读电子版本了。）

与第1版相比，最大的增加就是并发线程，也就是并行计算部分了。它集中描述了OpenMP和原子变量和结构。OpenMP并不是C语言标准，但它是C生态系统中非常可靠的一部分，所以它应该在本书的范围内。原子变量是在2011年12月发布的C标准修订版中加入的，一年以后本书第1版出版的时候，还没有编译器去支持它。现在好了，我们不仅可以在理论上进行讲解，同时还可以有现实的实现和测试代码了。参考第12章。

第1版本得到了很多细心读者的反馈。他们发现了很多可能导致bug的内容，从一些我在命令行上使用的斜杠，到句子中的一些可能会引起误会的词。这个世界上没有什么东西是没有错的，但是有了读者的反馈，这本书现在更加的正确和有用了。

本版中添加的其他内容：

- 附录A对于从其他语言转过来的读者，提供了一个C语言的简单的教程。我本来不想在第1版中包含这一部分内容，因为有太多的C语言教程了，但是包括了这部分内容让本书更加有用了。
- 应大家的要求，我扩展了关于使用调试器的内容。见本书第32页“2.1使用调试器”。
- 第 1 版有一部分讲述了如何写一个接受变长参数的函数，如sum(1,2.2)和sum(1,2.2,3,8,16)都是合法的。但是如果你想传送多个

变长列表的时候该怎么办呢？例如点积函数把两个变长向量相乘，dot((2,4),(−1,1)和dot((2,4,8,16), (−1,1,−1,1)) 10.4“多列表”介绍这一部分内容。

- 我重写了第11章，利用新函数来扩展对象。主要添加的部分是虚函数表的实现。
- 关于预处理器，我也多写了一些，主要是在8.1.2“测试宏”中介绍了一些测试宏的概念和用法。也包括了_Static_assert关键字。
- 我一直坚守诺言，本书中不包括关于正则表达式解析的内容（因为网上和其他书籍有太多这一部分内容了）。但是我在13.2.1“解析正则表达式”部分加上了一个演示，展示了如何使用POSIX的正则表达式解析函数。比起其他的语言，这些函数还处在比较原始的形态上。
- 第1版中关于字符串的讨论主要依赖于asprintf函数，它是一个sprintf类似的函数，当写一个字符串的时候，会自动分配需要的内存。这是一个被GNU广泛分发的版本，但是很多读者却被限制使用，所以在本版本中，我加入了例子9-3，演示了如何利用C语言标准的部分去实现这样一个函数。
- 第7章最大的主题是精细地去管理那些会造成麻烦的数值类型，所以第1版并没有提到很多在C99标准中新加入的数值类型。如int_least32_t，uint_fast64_t等[C99，§7.18；C11，§7.20]。很多读者鼓励我至少提一些有用的类型，如intptr_t和intmax_t，好吧，我从善如流。

本书使用的排版约定

本书使用如下排版约定：

斜体*(italic)*

用来表示新术语、URL、E-mail地址、文件名、文件扩展名等。

等宽字体（`Constant width`）

用来表示程序列表，同时在段落中引用的程序元素（例如变量、函数名、数据库、数据类型、环境变量、声明和关键字等）也用该格式表示。

等宽斜体（*`Constant width italic`*）

用于表示应该以用户提供的值或根据上下文决定的值加以替换的文本。

这个图标代表诀窍、建议和一般性的说明。

这个图标表示警告或错误。

使用示例代码

这是一本试图帮助你解决实际问题的书。总的来说，你可以在你的程序和文档中用本书的代码。除非你复制了太多的部分，你并不需要得到我的许可。例如，你的程序里使用了几段本书的代码并不需要得到许可。但是销售或发行含有O’Relly出版书籍中的源代码的确需要许可。通过引用本书及其源代码的方式来回答一个问题不需要得到许可。在你的文档中合并大量来自本书的代码不需要得到许可。

本书中用到的示范代码可以在以下地址找到：https://github.com/b-k/21st-Century-Examples。

我们感谢，但并不要求，您在引用时注明出处。一个引用通常包括书名、作者、出版商以及ISBN书号。例如，“C程序设计新思维（第2版）Ben Klemens(O’Reilly)。版权2014 Ben Klemens, 978-1-491-90389-6”。

如果你感觉你对示范代码的使用可能超出了上面列举的合理情况，你可以随时通过以下邮件地址联系我：permissions@oreilly.com。

网上书店

Safari网上书店（www.safaribooksonline.com）是一个点播方式的数字图书馆，可以下载世界顶级技术和商业作家的专业书籍和视频。

专业技术人员、软件开发者、Web设计者、商业和创意人士使用Safari网上书店作为他们首选的研究、解决问题、学习和认证培训的信息资源。

Safari网上书店为组织、政府机关和个人提供了一系列的产品组合和定价套餐。订阅人可以从一个可统一检索的出版商的数据库中找到几千本书籍、培训视频和预发布的手稿O’Reilly Media，Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redboks、Packet、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology。请登录其网站了解Safari网上书店的详情。

如何联系我们

如有对本书的评论或问题，请联系出版商：

美国

O’Reilly Media, Inc。
1005 Gravenstein Highway North。
Sebastopol, CA 95472。

中国

北京市西城区西直门南大街2号成铭大厦C座807室（100035）。
奥莱利技术咨询（北京）有限公司。

致谢

Nora Albert：慷慨的支持，豚鼠。

Jerome Benoit：Autoconf技巧。

Bruce Field、Dave Kitabjian、Sarah Weissman：严谨和彻底的审阅。

Patricj Hall：Unicode知识。

Nathan Jepson和Shawn Wallace：社论。

Andreas Klein：指出intptr_t的价值。

Rolando Rodriguez：测试、试用和调查。

Rachel Steely：出品。

Ulrik Sverdrup：指出我们可以使用重复指定初始化来设定默认值。

---

[1] “it rocks”，此处原文为双关语，借用英语中rock的不同含义，即“摇滚乐”和“很棒”。标题中的“punk rock”为流行于20世纪70年代的一种摇滚乐风格，以狂野反叛为特色，国内也称为“朋克”。——译者注

[2] 这篇前言明显地，而且是必须向*Punk Rock Language: A Polemic*致敬，作者是Chris Adamson。

[3] 像“can’t get to heaven with a three-chord song”这样的歌词，可能会

让Sleater-Kinney被归类在后punk时期？不幸的是，没有ISO punk标准为各种音乐提供精确的定义。

[4] 从下文可以看到，该书的出版被认为是C语言诞生的标志性事件，业内常称为*K&R*。——译者注

[5] 这里作者将问题与解答比喻为C语言函数入口的参考引用。——译者注

[6] 这里指调用C语言的语言。——译者注

[7] 译者注：世界上就两种语言，没人用的和大家骂的。

第1部分　开发环境

在脚本语言花园围墙外的旷野里，有大量解决C语言的那些烦恼的工具，当然你必须自己去寻找。我之所以这么说，因为其中一些工具对于你轻松编写代码是完全必要的。如果你不使用调试器（无论是独立存在的还是集成于IDE环境的），你简直就是自找苦吃。

有很多已经存在的库，你可以在你的代码中去使用。这样你可以集中精力去处理手头的问题，而不是重新实现什么链表、解析器和其他一些基础性的东西。当使用外部库的时候，要保证你的程序的编译也尽量简单。

本书第1部分的内容简述如下。

第1章讲述如何设定基本开发环境，包括找到一个包管理器并利用这个工具安装所用的工具。这些背景知识足够我们体会有趣的内容，比如用从别处得到的库来编译程序。整个过程非常标准化，包括一小部分环境变量的设定和配置。

第2章介绍调试、文档管理和测试工具，因为直到调试、编档和测试都完成的时候，你的代码才能显示出良好的一面。

第3章讨论Autotools，这是一个用于打包并发布你的程序的工具。这一章选择了一种比较详尽的介绍方法，因此还包含了编写shell脚本和makefile的方法。

我们可不能像某些人那样把生活搞得太复杂。第4章介绍Git，一个用来追踪你和同事们的硬盘文件版本的微小改变的工具，以便使你尽可能简单地融合不同的版本。

其他的语言也是现在C语言开发环境的重要因素，因为太多的语言提供了C接口。第5章提供了一些如何编写这些接口的建议，并给你一个基于Python的扩展例子。

第1章　准备方便的编译环境

对于完成一些实际的项目，C语言的标准库是不够的。

事实上，C语言的生态系统已经延伸到了C语言标准之外。也就是说，如果你不想局限于只完成作业本上的习题，那么你必须知道如何方便地调用那些非ISO标准库以外的函数。如果你想处理XML文件、JPEG文件或者TIFF文件，那么你需要那些不是标准但是可以免费使用的libxml、libjpeg以及 libtiff库。不幸的是很多教科书都把这部分内容忽略了，这就使得读者不得不自己进行探索。这就是为什么很多批评C语言的人会说出不公平的言论：C语言是一门40岁的老语言了，所以你必须从头开始写很多有用的函数。这些读者根本就不知道如何使用外部的库。

以下是本章的主题。

设定必要的工具集

比起需要自行准备各种组件的黑暗时期，如今已经轻松很多了。你只需要10～15分钟就可以建立起完整的开发环境（当然得加上下载所需要的时间）。

编译一个C程序

你已经知道怎么做了，但是我们还需要设定一下需要链接哪些函数

库，以及那些函数库所在的位置；只是输入cc myfile.c已经不够了。Make几乎是最简单的编译工具程序，我们就以它作为切入点开始介绍，先介绍一个最简单的makefile，虽然它很简单，但是有非常大的空间可以继续改进和扩充。

设定一些变量并加入一些新的库

无论我们使用什么系统，它们都会利用一小部分环境变量对自身进行定制。所以我们首先介绍这些环境是什么以及如何设定它们。一旦我们完成了基本的设定工作，只要稍微调整这些环境变量，就可以使用新加入的函数库了。

建立一个编译系统

作为回报，我们可以利用以上介绍的这些知识，建立一个简单的编译系统。在这个系统中，我们可以在命令行编译那些复制来的代码了。

对于使用IDE（集成开发环境）的用户来说，有一点值得特别说明：即使你不使用make，但是本部分内容依然和你有关。make使用的很多步骤，在IDE中都有对应的步骤。如果你知道make的内部机理，你就有能力对IDE做出更适合你的一些调整和设定。

1.1 使用包管理工具

如果你没有使用过包管理工具，那你太落伍了。

这里介绍包管理工具有以下几个原因：首先，有些读者可能还没有安装过程序。对于这部分读者，我把这部分内容放到本书的开始，就是

为了让你尽快地获得这些工具。一个好的包管理工具会让你很快地建立起一个POSIX子系统，其中包含很多你听说过的语言的编译器、很多的游戏、一些常见的办公应用软件以及几百个C语言的库等。

其次，对于C语言的开发者，包管理工具是日后我们安装C语言函数库的一个重要工具。

最后，当你从一个包的使用者成长为一个包的开发者的时候，本书会教你如何让自己的包更易于安装，这样当包存储库的管理员决定把你的代码包含进存储库的时候，管理员可以无障碍地建立最后的包。

如果你是一个Linux用户，并已经通过包管理器配置了电脑，你就已经知道了软件获取的过程非常简单。对于Windows系统的用户来说，我会在下面介绍Cygwin。对于使用Mac的用户有一些选项，例如Fink和Homebrew或者Macports，所有的这些选项都依赖于Apple的Xcode包。你可以在系统的安装光盘上找到它，或者到苹果的APP商店获取，或者通过注册成为苹果的开发者而获取（依赖于你的Mac电脑的具体生产年份）。

你需要什么包呢？这是C开发过程中最基本的问题。因为每一个系统都有不同的组织方式，所以包可能被包装在不同的地方，或者在基本包中默认安装，或者被命名为其他古怪的名字。如果你也不是很确定，那就先安装它再说，毕竟我们的电脑不会因为安装过多软件就变得不稳定或者运行慢了。但是你可能没有那么大的带宽来下载，或者没有那么大的硬盘来安装所有的包，所以你还需要做一些选择。如果你发现自己没有安装某些包，还可以回过头来重新安装。以下这些包是必须安装的。

- 编译器，必须安装gcc；也可以选择安装Clang。
- GDB调试器。
- Valgrind，用于测试C内存使用错误。
- Gprof，用于运行效率评测。
- Make，使得你不用直接调用编译器。
- Pkg-config，用于查找库。
- Doxygen，用于生成程序文档。
- Text editor, 目前有几百个文本编辑器可以选择，这里给出几个主观的推荐。

  ——Emacs 和Vim是计算机老手的最爱。Emacs几乎涵盖了很多的功能（E代表extensible，可扩展的）；Vim更小型化，并且有很多友好的功能键。如果你是一个编辑器的重度用户，那么你需要从这两个编辑器中选择一个来进一步学习。

  ——Kate非常友好，也有吸引人的界面，能提供我们开发程序时需要的一些功能，例如语法高亮。

  ——最后的一个选项是Nano，使用上最简单，是基于文本模式的编辑器。即使你没有GUI界面，也可以使用它。
- 如果你是IDE的爱好者，那就安装一个（或者几个）。同样有很多的选择。这里只是给出一些推荐。

  ——Anjuta：属于GNOME家族，可以方便地使用 GNOME GUI builder工具Glade。

  ——KDevelop：属于KDE家族。

  ——XCode：苹果公司OS X的IDE。

  ——Code::blocks：相对简单，在Windows下工作。

  ——Eclipse：拥有很多按钮和杯架的“豪华跑车”，跨平台。

在本章的后面，我还会介绍一些强力工具。

- Autotools：Autoconf、Automake、libtool。
- Git。
- 其他的shell，例如Z shell。

当然，还有一些C库能够避免你重新发明“轮子”（更准确的类比是，重新发明“火车头”）。你可以获得更多的库，下面这些库是我们要在本书中用到的。

- libcURL。
- libGLib。
- libGSL。
- libSQLite3。
- libXML2。

C语言库没有一致的命名标准，你必须要了解你的包管理器是如何将一个单独的包分解成不同的子部分的。一般有一个供用户使用的包，同时还有一个供开发者使用的包，所以请在选择基本的包的同时，选择带有-dev或者-devel的包。某些系统将文档分拆在独立的包中。也有的要求你单独下载带有调试符号表的包。如果你在没有调试符号表的包上运行GDB，那么GDB会要求你下载带有调试符号表的包。

如果你使用POSIX系统，在安装完所有前面提到的工具后，你就有了完整的开发环境，现在你可以开发你的程序了。对于Windows的用户，下面介绍如何在Windows操作系统下设置开发环境以便和Windows主系统进行沟通。

1.2 在Windows下编译C程序

在大多数的系统下，C语言是主要的开发语言，其他的工具都以C语言为参照进行开发。但是Windows操作系统很奇怪地忽略了C语言。

因此我们需要一点时间来说明如何在Windows环境下设置开发环境。如果你不使用Windows系统，可以忽略这部分内容，直接跳到1.3“链接函数库的方式”一节。

1.2.1 Windows中的POSIX环境

由于C语言起源于UNIX，并与UNIX一起演变，我们很难将两者分开来讨论。我想从POSIX开始介绍应该比较简单。如果想在Windows平台下编译其他平台下开发的程序，这是最自然的方法。

就作者所知，所有的操作系统可以分为两大类。

- POSIX兼容系统。
- Windows操作系统家族。

POSIX兼容并不代表系统的外观和UNIX相像，例如大部分Mac使用者完全不知道自己使用的是一个带有华丽界面的BSD系统。但是了解这部分知识的人却可以从应用程序→工具文件夹中启动终端（Terminal）应用程序，然后在其中运行ls、grep或者make等各种工具。

另外，并不是所有的系统都100%符合标准（例如Fortran 77编译器）。就本书来说，我们需要有一个基本的类似POSIX shell的shell、一些工具（sed、grep、make等）、一个C99编译器，以及fork和iconv等标

准C语言库以外的函数库。这些工具和库可以作为主系统的补充。包管理工具相关的底层脚本、Autotools以及所有开发可移植代码的工具都在某种程度上依赖上面提到的那些工具和库。所以即使你不愿意与命令行打交道，你依然需要安装这些工具和库。

在作为服务器的操作系统以及完整的 Windows 7系统中，微软公司提供以前称为INTERIX，现在称为Subsystem for UNIX-based Applications（SUA）的子系统，包含常用的POSIX系统调用、Korn shell以及gcc。这个子系统默认不安装，需要你另行下载。目前版本的Windows不再提供SUA，Windows 8也不提供了，因此无法依赖微软自己提供的POSIX子系统了。

所以我们需要使用Cygwin。

如果想要从头建立Cygwin，可以参考下面的介绍。

1．为Windows撰写C函数库，提供所有的POSIX函数。这需要调整Windows和POSIX系统间的差异，例如Windows系统使用C:代表硬盘，而POSIX使用统一文件系统（Unified Filesystem）。对这种情况，可以为C:建立cygdrive/c，为D:建立cygdrive/d等别名。

2．现在可以编译POSIX标准程序，链接到上一步介绍的函数库，产生新的Windows版本的ls、bash、grep、make、gcc、x、rxvt、libglib、perl、python等。

3．建立好很多的程序和函数库以后，接着建立包管理工具，让使用者可以自己选择安装软件。

但是作为Cygwin的使用者，你不需要完成上面介绍的那些麻烦的步骤，只需要从Cygwin网站（http://cygwin.com）下载包管理工具，选择要安装的包，当然包括上面清单列出来的那些程序，再加上一个合适的终端（Terminal）（可以试试Mintyy，或者安装X系统中的Xterm，这两者都比Windows 自带的cmd.exe友好）。安装完后，你就可以看到开发系统所需要的各种豪华工具都在其中了。

在“路径”部分，我讨论了影响编译的各种环境变量，包括搜索文件的包含路径。这一部分并不是POSIX所独有的，Windows也有环境变量，你可以在控制面板的环境设置部分找到它们。如果你把Cygwin的bin路径（C：\cygwin\bin）加到Windows的PATH变量中，那么使用Cygwin的时候会更加方便。

现在你可以编译C代码了。

1.2.2 在POSIX环境中编译C语言

微软公司在Visual Studio中提供了C++编译器，提供了与C89相容的模式（通常称为ANSI C，虽然C11是ANSI的当前标准）。这是目前微软公司提供的编译C语言的唯一方式，微软公司的很多代表都表示不会去支持C99标准（更别提C11标准了），Visual Studio是唯一一个还使用C89标准的编译器，因此我们需要其他替代方案。

当然，Cygwin提供了gcc，如果按照前面介绍的步骤安装好了Cygwin，那么你就有了完整的开发环境。

在Cygwin下编译的程序会依赖于Cygwin1.dll库中所提供的POSIX函

数（不论你的程序是否使用任何POSIX相关的调用）。在有安装Cygwin的机器上执行这些程序不会有任何问题，使用者可以通过双击执行这些程序，系统能够找到cygwin1.dll。如果要在没有安装cygwin1.dll的机器上运行这个程序，你必须与程序同时提供cygwin1.dll文件。在作者的机器上，路径是/bin/Cygwin1.dll，Cygwin1.dll采用类似GPL的授权方式（参考前言部分的“法律解读”），如果将dll从Cygwin分离出来而单独与你的应用程序一起分发，那你必须提供程序的源代码[1]。

如果有困难，你需要用不依赖于Cygwin1.dll的方式来重新进行编译。也就是要你的程序中使用MinGW，而不去使用POSIX相关的函数（像fork和popen函数），就像我们后面介绍的那样。利用cygcheck可以发现你的应用程序依赖于那些DLL，从而验证你的执行程序是否连接到了Cygwin1.dll。

查看一个程序或者动态链接库依赖于哪些库，用下面的命令：

- Cygwin: cygcheck libxx.dll。
- Linux: ldd libxx.so。
- Mac: otool -L libxx.dylib。

1.2.3 在非POSIX环境中编译C语言

如果你的程序不需要调用POSIX函数，你可以使用MinGW（Minimalist GUN for Windows），它提供了一个标准的C编译器和一些基础工具。MSYS是与MinGW伴生的，提供了另外的一些工具和shell。

MSYS提供了一个POSIX shell（你可以使用Mintty或者RXVT终端来运行你的shell）。或者干脆不使用任何命令行，只是使用Code::blocks，这是一个在Windows上使用MinGW的IDE集成环境。Eclipse是一个功能更加丰富的可以配置成与MinGW一起工作的IDE集成环境，虽然这需要更多一点的配置工作。

不过如果POSIX命令行能让你感觉更舒服些，那就安装Cygwin，下载提供MinGW版本的gcc，用这些编译环境而不是用与POSIX连接的、默认版本的Cygwin gcc。

如果你还没有遇到过Autotools，那你会很快遇到。通过Autotools建立软件包的三个标志性命令是：./configure、make和make install。MSYS提供了足够的机制以保证可能让包在MinGW环境下安装；否则你就必须从Cygwin的命令行来安装这个包，但是你可以用下面的命令来配置这个包，使用Cygwin的MinGW32编译器来产生与POSIX无关的代码：

```
/.configure –host=ming32
```

然后像通常那样运行make和make install。

在MinGW下编译后，不管是通过命令行还是Autotools，你都会得到一个Windows本地的二进制代码。因为MinGW并不知道cygwin1.dll的存在，你的程序并不会有任何POSIX调用。你得到的将是一个真正的

Windows程序，没有人会知道你是从POSIX环境编译出来的。

然而，MinGW的真正问题是缺乏预编译库[2]。如果你想摆脱cygwin1.dll，那么你也不能使用与Cygwin一起发行的libglib.dll版本。你将必须从源代码重新将GLib编译成一个Windows 动态链接库——但是Glib在国际化方面依赖GNU的gettext，所以你需要先把那个库编译一遍。现代代码依赖于现代的库，所以你可能会发现自己花费了很多时间来处理类似的工作，而在别的系统上可能只是一行包管理器的命令。如果这样，我们就真的如某些人所说，C已经40岁了，你需要从头写每样东西。

所以，下面就是几句忠告。微软公司拒绝在C语言标准的支持上与别人沟通，任由其他人实现后grunge时代的C编译器。Cygwin完成了这些工作并提供了全功能的包管理器，带有可以满足你的多数或全部工作的足够多的库，但这些都伴随着POSIX风格的代码和Cygwin的DLL。如果你觉得这是一个问题，那么你将不得不多做很多工作，以重建那些优雅的代码所需要的整个环境和库。

1.3 链接函数库的方式

有了编译器，有了POSIX的工具包，还有一个可以用来方便地安装几百个库的包管理器。现在我们开始用这些工具来编译程序。

我们必须从编译器命令行开始，这会很快给我们带来很多麻烦，好在还可以用三个（有时候是三个半）相对简单的步骤结束。

1．设置一个变量，代表编译器使用的编译选项。

2．设置一个变量，代表要链接的那些库。所谓的半个步骤是指，有时你不得不设定一个唯一的变量，指定编译时使用的函数库；或者有时不得不设定两个变量，分别用在编译时和运行时的链接。

3．设置一个使用这些变量来协调编译的系统。

为了使用一个库，你必须告诉编译器你将从库中两次导入函数：一次是为了编译，一次是为了链接。对于一个在标准位置的库，这两次导入一次发生在通过程序中的#include指令时，另外一次发生在通过编译选项−l进行编译时。

例1-1展示了一个小例子，可以用来做一些神奇的计算（至少对于我来说是有趣的；如果统计学术语对你来说就像希腊文一样，你也不用太在意）。erf（*x*）是C99标准的误差函数，是与平均数为0、均方差为的从0到*x*的正则分布的积分紧密相关。这个例子中，我们用erf来验证一个在统计学家中流行的领域（一个标准大样本假设的95%置信区间）。我们把这个文件命名为erf.c。

例1-1　一个使用标准库的只有一行的程序（erf.c）

```
#include <math.h>    //erf, sqrt
#include <stdio.h>   //printf

int main(){
    printf("The integral of a Normal(0, 1) distribution "
           "between -1.96 and 1.96 is: %g\n", erf(1.96*sqrt(1/2.)));
}
```

你应该已经很熟悉#include行了。编译器将把math.h和stdio.h文件的内容添加在源文件的这个地方，并因此导入了printf、erf和sqrt的声明。

在math.h中的声明并没有具体指定erf函数做什么，只是说这个函数接收一个double型参数，也返回一个double类型的值。这些已经足够编译器去检查我们使用的合法性并产生一个目标文件了。这个目标文件中带着一个给计算机的标记，这个标记告诉计算机，一旦你看到这个标记，就去找erf函数，并用erf的返回值替代这个标记。

而链接器的任务是确实找到erf这个函数来取代目标库中的标记，这个函数就在你硬盘的某个库里。

在math.h中声明的数学函数分散在它们自己的库中，你需要通过一个-lm编译选项告诉链接器。这里，-l是一个选项，用来指示某个库需要被链接进来。而本例中的库有一个用单个字母表示的名字：m。你不用设定任何选项就可以使用printf，因为在链接命令行的末尾，有一个隐含的-lc选项来要求链接器链接标准libc库。随后，我们将看到Glib 2.0通过-lglib-2.0被链接进来，GNU科学计算库也通过-lgsl被链接进来，依此类推。

所以，如果文件名为erf.c，那么完整的gcc编译器命令行应该如下所示（这里包括几个选项，在后面将会详细介绍）。

```
gcc erf.c -o erf -lm -g -Wall -O3 -std=gnu11
```

这样就能告诉编译器通过程序中的#include包含数学函数，并告诉链接器通过命令行中的-lm链接数学库。

-o选项用来给出输出文件的名字；否则将得到一个默认的可执行文件名a.out。

1.3.1 一些我喜欢的选项

在后面你将看到我几乎每次都用到一些编译器选项，并且我建议你也使用它们。

- -g，表示加入调试符号。如果没有这个选项，调试器就不会显示变量或者函数的名字。这些调试信息并不会把程序拖慢，而且我们也不在乎程序增加1KB，所以看起来没啥理由不去用它。该选项对gcc、Clang和icc（Intel的编译器）都有效。
- -std=gnu11，这是gcc特有的选项，允许你使用符合C11和POSIX标准的代码。否则，gcc可能将一些目前有效的语法判为非法。本书撰写的时候，一些系统是在C11之前发布的，这个时候你就得使用-std=gnu99。这是gcc独有的；而其他的编译器都从很久以前就将C99当作默认配置。POSIX标准规定你的系统中必须有C99，因此以上这一命令行的与编译器无关的版本应该是：

```
c99 erf.c -o erf -lm -g -Wall -O3
```

在后文将介绍的makefile中，我通过设定一个变量CC=c99实现这个效果。

警告

在Mac系统中，c99是一个特别修改的gcc版本，可能并不是使用者预期的版本。如果你有一个不符合要求的c99版本，或者它根本就不存在，那就自己建立一个。把一个叫作c99的文件放在你的

搜索路径中的目录里：

```
gcc --std=gnu99 $*
```

或者如果你愿意，就用

```
clang $*
```

并通过chmod +x c99让它成为可执行的文件。

- -O3显示出这里的优化等级是三级，也就是尝试已知的所有方式去建立更快的代码。如果你运行调试器，你会发现太多的变量被优化掉了，以至于你都没办法跟踪执行情况，如果这样影响了你调试程序，你可以换成-O0，这是常见的CFLAGS参数调整。一般情况下，CFLAGS都是设置成-O3的。该选项对gcc、Clang和icc都有效。
- -Wall添加编译器警告。该选项对gcc、Clang和icc都有效。对于icc用户，你可能更喜欢用-w1，它显示编译器警告，并不显示注意（remarks）。

提示

要坚持使用编译器警告。即使你对你的代码质量已经非常挑剔了，即使你可能已经熟知C语言标准了，你也不可能比你的编译器更挑剔和更熟知C。旧的C教材连篇累牍地警告你注意=和==的差别，或者检查是否所有的变量在使用前都被初始化了。作为一

> 本更加现代的书的作者，我可以轻松一点了，因为我可以把所有的警告总结为一点：永远都要用你的编译器警告。
>
> 如果编译器建议你做一个改变，不要怀疑或试图碰运气而放弃修改。尽可能去：（1）理解你为什么得到了警告；（2）修改代码直到不产生任何警告和错误。编译器信息是出了名的难懂，所以如果你在第（1）步有困难，把警告信息贴在搜索引擎上，就能看到有多少人在你之前也面对了类似的问题。你可能想加上-Werror编译选项，这样你的编译器将把警告当作错误来处理。

1.3.2 路径

在笔者的硬盘中有超过70万个文件，声明sqrt和erf函数的头文件只是其中之一，而且还有一个是包含了这些函数对应的被编译后的目标文件[3]。编译器需要知道在哪个目录中去查找正确的头文件和目标文件，当开始使用非标准C库的时候，这个问题就会变得更加严重。

在一个典型的安装中，库可能存放的地方至少有三个。

- 操作系统的厂家可能预定义了一两个自己用来安装库文件的目录。
- 可能存在为本地系统管理员准备的用于安装包的目录，并且不会被来自厂家的下一次操作系统更新所覆盖。系统管理员也可能用一个特殊的破解版本的库覆盖系统缺省的版本。
- 用户一般来说没有向这些路径写操作的权限，但是有从他们的主目录利用那些库的权限。

操作系统标准的路径一般不会引发什么问题，编译器也应该知道如何查找那些路径，并找到标准C库以及伴随其安装的任何文件。POSIX

标准用“通常位置”来指代上面所说的目录。

但是对于其他的东西，你必须告诉编译器如何查找。这使得状况变得有点复杂：没有一个统一的方法去寻找那些不按标准位置安装的库。这一点是人们对C比较恼火的地方。不过令人感到欣慰的是，编译器知道如何在通常位置查找，而库的提供者也倾向于将库安放在通常位置，所以你可能从来没有真正手工去指定这些路径。再者，也有几种方法使你可以指定路径。最后，一旦你把非标准库安装在系统上，你可以在shell脚本或makefile中的变量中设定这个路径，然后就再也不会有路径的烦恼了。

假设你在计算机上安装了一个叫作Libuseful的库，并且你知道与之相关的文件放在/usr/local/目录下，也就是你的系统管理员安装本地函数库的位置。你已经把#include <useful.h>放在了你的代码里，现在你必须把下面一行放在你的命令行中：

```
gcc -I/usr/local/include use_useful.c -o use_useful -L/usr/local/lib –luse
ful
```

- -I添加指定的路径到头文件的搜索路径范围内，编译器会在头文件搜索路径范围内搜索你放在代码里面的用#included指定的那个头文件。
- -L添加指定的路径到库的搜索路径范围内。
- 注意顺序问题。如果你有一个叫作specific.o的文件依赖于Libbroad库，而Libbroad库依赖于Libgeneral，那么你应该输入：

```
gcc specific.o -lbroad –lgeneral
```

任何其他顺序，比如gcc-lbroad-lgeneral specific.o，都可能失败。你可以这样理解链接器的工作方式，链接器首先查看第一个目标——specific.o，将无法解析的函数、结构和变量名记入一个列表。然后链接器查看下一个目标——lbroad，并在这个目标内搜索列表中仍然缺失的项目，同时有可能在列表中添加新的项目；接着在-lgeneral查找仍然缺失的项目。如果直到搜索完最后的目标仍然存在未解析的符号（包括在最后的隐含的-lc），链接器将终止运行并向用户给出最后剩下的未解析项目。

现在回到路径问题：要链接的库到底在哪里呢？如果安装库的包管理器和安装操作系统其他部分的包管理器相同，那么库最可能在通常路径，你也不用去担心这个。

你可能想不清你自己的本地库应该放在何处，如/usr/local，还是/sw或者/opt。你无疑可以用硬盘搜索的方式来查找，如在你的机器或POSIX环境中使用：

```
find /usr -name 'libuseful*'
```

来搜索/usr 中以libuseful开头的文件。当你发现Libuseful库的共享目标文件在/some/path/lib中，那么几乎可以肯定对应的头文件一定在/some/path/include中。

在硬盘里到处找库文件是一件很恼人的事情，为了解决这个问题，pkg-config维护了一个包含配置信息和位置信息的资料库，然后pkg-config会报告编译时需要的这些信息。在命令行中输入pkg-config；如果你得到一个错误提示说“没有指定包名字”，那么很好，说明你有pkg-

config命令了，你可以用它来做研究了。例如，在我个人计算机的命令行上输入以下两行命令：

```
pkg-config --libs gsl libxml-2.0
pkg-config --cflags gsl libxml-2.0
```

得到下面两行输出：

```
-lgsl -lgslcblas -lm -lxml2
-I/usr/include/libxml2
```

这些正是我用来编译GSL和LibXML2所需要的所有选项。-l选项揭示出GNU科学计算库依赖于基本线性代数子程序库（BLAS），而GSL的BLAS库依赖于标准数学库。看起来所有这些库都在通常路径，因为这里没有-L选项，但是-I选项表明LibXML2的头文件的特殊位置。

回到命令行，shell提供了一个方法，就是当你把一个命令行用单引号包围时，这个命令行会被其自身的输出替代。就是说，输入：

```
gcc 'pkg-config --cflags --libs gsl libxml-2.0' -o specific specific.c
```

编译器看到的是：

```
gcc -I/usr/include/libxml2 -lgsl -lgslcblas -lm -lxml2 -o specific specifi
c.c
```

所以pkg-config会为我们做很多工作，但是这并不足以使它成为一个标准，不是所有平台都有pkg-config命令，也不是每个库都用它注册。如果你没有pkg-config，你就必须自己研究，比如读这个库的手册，或者像前面那样搜索。

警告

有很多与路径有关的环境变量，比如CPATH、LIBRARY_PATH或者C_INCLUDE_PATH。你可以在.bashrc或别的用户定义的环境变量列表中设定它们。它们都不是标准的——连Linux和Mac中的gcc都分别使用不同的变量，别的编译器自然也使用自己的变量。我发现在每个项目的makefile或类似机制的基础上，用-I和-L来设定这些路径相对更容易。如果你喜欢这些路径变量，可以在你的编译器的帮助文件的末尾查找符合你的情况的相关环境变量。

即便用pkg-config，我们也显然需要某种工具帮我们把所有这些自动执行。每个元素都很容易理解，但是组合起来却是一个冗长且重复的琐碎工作。

1.3.3 运行时连接

编译器连接静态库的时候，是将库里的相关内容直接复制到最终的可执行文件中的。所以程序本身或多或少是一个独立的系统。而共享库与你的程序是在运行时链接的，就是说我们在运行时会遇到像编译器在编译时寻找库的路径那样的问题。甚至更糟的是，你的程序的用户也存在同样的问题。如果你的库在一个非标准的路径，那你需要找到一个修改运行时搜索库的路径的方法。有以下选择。

- 如果你用Autotools打包你的程序，Libtools知道如何添加合适的选项，也就是说你不用再操心什么。

- 当用gcc、Clang或者icc来基于一个在libpath中的库编译程序的时候，添加：

```
LDADD=-Llibpath -Wl,-Rlibpath
```

到相应的makefile中。-L选项告诉编译器到哪里去找到库以解析符号；-Wl选项从gcc/Clang/icc传递这个选项到链接器，而链接器将给定的-R嵌入所链接的库的运行时搜索路径。不幸的是，pkg-config经常不知道运行时路径，所以你可能必须手工输入这些信息。

- 运行的时候，除了通常的位置，以及你通过wl,R…这个命令显示标注在可执行程序中的路径以外，链接器还会搜索其他的路径，这个路径一般是在shell（.bashrc、.zshrc，或者别的什么对应物等）启动脚本中设定。为了确保在运行的时候搜索动态库时使用这个路径，可使用命令：

```
export LD_LIBRARY_PATH=libpath:$LD_LIBRARY_PATH      #Linux, Cygwin
export DYLD_LIBRARY_PATH=libpath:$DYLD_LIBRARY_PATH  #OS X
```

有些人反对过度使用LD_LIBRARY_PATH（万一有人把恶意伪装的库放到那个路径，在你没有察觉的情况下代替了真正的库怎么办？），但是如果你所有的库都放在一个路径，将这个路径加入进来也是合理的。

1.4 使用makefile

makefile提供一个解决所有以上这些麻烦的方案。它基本上可以看作结构化的变量和一系列一行的shell脚本。POSIX标准的make程序读入

makefile的内容作为指令和变量，然后自动化处理那些冗长、烦琐的命令行。在这部分讲解之后，就没有什么必要去直接调用编译器了。

在“makefile与shell脚本”中，会讲述关于makefile的更多细节；这里，先给出一个最小的、实用的并且能够编译一个依赖于一个库的基本程序的makefile，它只有6行：

```
P=program_name
OBJECTS=
CFLAGS = -g -Wall -O3
LDLIBS=
CC=c99

$(P): $(OBJECTS)
```

用法：

- 第一次：将这段脚本存为一个名为makefile的文件，与.c文件放在同一个目录中。如果你在使用GNU Make，你可以将首字母大写，即命名为Makefile，这样就将该文件与其他文件区别开了。把你的程序的名字放在第一行（用progname，而不是progname.c）。
- 每次你需要重新编译的时候，输入make命令。

自己动手：下面是编程世界著名的hello.c程序，就两行：

```
#include

int main(){ printf("Hello, world.\n"); }
```

把这个源文件和前面的makefile存在同一个目录中，然后试着按前面的步骤编译和运行程序。

1.4.1 设定变量

很快我们会介绍makefile的实际应用，但你可能注意到前述makefile里6行中有5行是关于变量设定的（目前多数被设定为空），这意味着我们还需要多花一点时间在环境变量的细节上。

警告

历史上，曾经出现过两种主流的shell语法：一种基本上是基于Bourne shell的，另一种主要基于C shell。C shell的变量语法稍有不同，例如，采用set CFLAGS="-g-Wall-O3"来设定CFLAGS的值。但是POSIX标准是写成Bourne类型的语法，也就是我这本书余下部分所采用的。

shell和make用$来指代变量的值，但是shell用$var，而make要求任何多于一个字母的变量必须用括号括起来：$（var）。所以，前面的makefile中，$（P）:$（OBJECTS）将相当于：

```
program_name:
```

有以下几种办法来让make识别变量：

- 在调用make之前从shell中设定变量，并且使用export命令导出这些变量，也就是说当shell产生一些子进程的时候，它有自己的环境变量列表。从一个POSIX标准的命令行中设定CFLAGS，可以这样：

```
export CFLAGS='-g -Wall -O3'
```

我自己经常忽略这个makefile中的第一行，P=program_name，取而代之的是在每个会话中通过export P=program_name来设定，这样我就不用重复编辑 makefile了。

- 你也可以将那些export命令放在你的shell启动脚本中，比如.bashrc或.zshrc。这可以确保每次你登录或开始一个新的shell，这个变量都会被设定并导出。如果你很确信CFLAGS每次都会是一样的，你可以在启动脚本中设定它们，然后就再也不用惦记这个问题了。
- 你可以通过在一个命令前放一个赋值操作，这就为这个命令直接设置了这个环境变量。env命令会列出它所知道的所有环境变量，所以当你运行：

```
PANTS=kakhi env | grep PANTS
```

你将看到正确的变量及其他的值。这是为什么shell不让你在等号附近放空格的原因：空格是用来分割命令行中的赋值操作和后续的命令行的。

在这个方法中设定和导出变量应该在一行实现。如果你在命令行中执行了上面这条命令，再次运行env | grep PANTS，就会发现PANTS不再是一个被导出的变量了。

只要你愿意，你可以指定任意数量的变量：

```
PANTS=kakhi PLANTS="ficus fern" env | grep 'P.*NTS'
```

这个技巧出现在shell规范的“简单命令”描述部分，也就是说赋值必

须在一个实际的命令之前。这在你使用非命令的shell结构时非常重要。编写：

```
VAR=val if [ -e afile ] ; then ./program_using_VAR ; fi
```

将失败并伴随一个晦涩的语法错误。正确的方式是：

```
if [ -e afile ] ; then VAR=val ./program_using_VAR ; fi
```

- 就像前面介绍的那个makefile一样，你可以在makefile的头部设定变量，类似：CFLAGS=。在makefile中，你可以在等号两边放上空格，这不会引发任何错误。
- make可以让你在命令行中设定变量，并独立于shell。所以下面两行基本是对等的：

```
make CFLAGS="-g -Wall" Set a makefile variable.
CFLAGS="-g -Wall" make   Set an environment variable visible to make and i
ts children.
```

对makefile而言，上面所有这些手段都是相等的；例外之处在于，被make调用的子程序只知道新的环境变量，而不知道任何makefile中的变量。

C中的环境变量

在C代码中，可以用getenv函数来得到环境变量。getenv非常简单、易用，所以你可以从命令行中尝试设定不同的选项。

例1-2是一个打印示例程序，只要用户需要，随时打印一个信息到屏幕。环境变量msg用于设定要打印的信息，同时通过reps设定重复的次数。请注意我们是如何设定它们的默认值10和“Hello.”的，在调用getenv返回NULL（典型的含义是这个环境变量没有被设定）的时候，默认值才被设定。

例1-2　环境变量提供了一个改变程序细节的快速方式（getenv.c）

```
#include <stdlib.h> //getenv, atoi

#include <stdio.h>  //printf


int main(){

    char *repstext = getenv("reps");

    int reps = repstext ? atoi(repstext) : 10;


    char *msg = getenv("msg");

    if (!msg) msg = "Hello.";


    for (int i=0; i < reps; i++)

        printf("%s\n", msg);

}
```

就像之前看到的，我们可以用一行命令导出一个变量，这样可以使得向程序发送变量更加方便。用法：

```
reps=10 msg="Ha" ./getenv

msg="Ha" ./getenv

reps=20 msg=" " ./getenv
```

你可能觉得这个用法很奇怪——程序的输入应该跟在程序名后面才对，真是可恨——先不管这些奇怪的事情，你可以看到程序自身进行了一些设置工作，我们几乎没费什么精力就立即得到了来自命令行的命名参数。

当你的程序有更多的程序变量的时候，可以研究一下getopt或者是基于GNU的argp_parse，它按照通常的方法取得输入参数。

make也提供一些内置的变量。下面是其中一些（POSIX标准）变量

的介绍，你可能在随后学习“规则”一节中用到这些变量。

$@

返回完整的目标文件名。所谓目标（target）就是指需要被生成的文件，比如从一个a.c文件中编译而得到的a.o文件，或者一个通过链接.o文件生成的程序。

*$**

不带文件名后缀的目标文件。如果目标文件是prog.o，$就是*prog*，而*$*.c就成为prog.c。

$<

触发和制作该目标的文件的名称。如果我们正在制作prog.o，有可能是因为prog.c文件刚被修改，所以$<就是prog.c。

1.4.2 规则

现在让我们专注地了解一下makefile的执行过程，并了解变量是如何影响这个过程的。

先不讨论变量的事情，来看一下makefile的代码片段，一般有以下的形式：

```
target: dependencies
        script
```

假设输入命令make target，则target被调用，那么dependencies（依

赖项）将被检查。如果target是一个文件，dependencies也是文件，并且target是比dependencies时间上更新的文件，那就是说这个文件已经是最新的，所以系统也不会再做什么。否则，针对target的处理将被暂停，所有的dependencies将首先被运行或重新产生，当这一切结束后，target段落的script（脚本）部分才会被执行。

例如，在本文成书之前，我的博客上贴出了一系列的文章（在http://modelingwithdata. org）。每篇博客都是用HTML和PDF格式上传的，也都是用LaTeX产生的。为了让这个例子简单，我忽略了很多细节（如latex2html的很多配置选项），但这是一个人们经常编写和运行的makefile。

警告

如果你将这些makefile片段从屏幕或者书本上复制到makefile文件中，不要忘记每行代码开头的空白部分必须是制表符（Tab键）而不是空格（Space键）。要怪就怪POSIX标准吧。

```
all: html doc publish

doc:
    pdflatex $(f).tex

html:
    latex -interaction batchmode $(f)
    latex2html $(f).tex

publish:
    scp $(f).pdf $(Blogserver)
```

我们通过类似export f=tip-make的命令来设定f。然后在命令行输入make的时候，第一个目标all被检测到。就是说，不指定目标的make命令默认指定makefile文件中的第一个目标all。这个目标依赖于html、doc和publish，所以那些目标也被依次调用。如果我知道这还没有准备好递交给这个世界，我可以调用make html doc并仅完成其中的两个目标。

在之前那个简单的makefile中，我们仅有一组target/dependency/script。例如：

```
P=domath
OBJECTS=addition.o subtraction.o

$(P): $(OBJECTS)
```

这个和我的博客里的makefile遵循同样的支持项和脚本执行次序，但是脚本是隐含的。这里，P=domath是被编译的程序，并且依赖于目标文件addition.o和subtraction.o。因为addition.o没有被列出来当作一个处理目标，所以make用一条如下所述的隐含的规则来从.c文件编译.o文件。对subtrction.o和domath.o也是一样的操作（因为GNU make隐含假设domath依赖于这里给出设置的domath.o）。一旦所有的目标文件被建立了，因为没有在$（P）目标处指定运行的脚本，那么GNU make使用它的默认脚本来链接.o文件而生成一个可执行文件。

POSIX标准的make有一个特殊的从a.c源文件到a.o的编译方法：

```
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c
```

这里$（CC）变量代表你的C编译器；POSIX标准规定一个默认的CC=c99，但是GNU make的当前版本设定为CC=cc，并且一般就是gcc的

一个链接。在本段最早的那个最小的makefile中，$（CC）被明确设定为c99，$（CFLAGS）被按照之前的选项列表设定，$（LDFLAGS）没有被设定，因此以空值传入。所以如果make认为它必须产生your_program.o，根据以上的makefile中，会运行下面的命令行：

```
c99 -g -Wall -O3 -o your_program.o your_program.c
```

当GNU make觉得你需要从目标文件链接出一个可执行文件时，它用下面的方法实现：

```
$(CC) $(LDFLAGS) first.o second.o $(LDLIBS)
```

如果想起在链接器中存在次序问题，那么我将需要两个链接器变量。在前面的例子中，我们需要：

```
cc specific.o -lbroad -lgeneral
```

作为链接命令的对应的部分。比较实际的编译命令和预设的编译命令，我们可以看到需要设定LDLIBS=-lbroad-lgeneral。

提示

如果你想看到你的make内置的全部默认规则和变量的列表，可以尝试：

```
make -p > default_rules
```

所以，这个游戏就是：找到合适的变量并把它们设置在makefile中。你还是要探究一下正确的选项分别是什么，但是至少你可以把它们写在makefile中，之后便再也不必去为这个事情而烦心了。

> 自己动手：修改makefile以便编译erf.c文件。

如果你用一个IDE，或者CMAKE，或者任何POSIX标准make的替代品，你都可以做这个“找到合适的变量”的游戏。我还会继续讨论前面的最小makefile，在你的IDE中应该不难找到对应的变量。

- CFLAGS是根深蒂固的习惯，但是你需要为不同的系统的链接器设定不同的变量。甚至LDLIBS都不是POSIX标准的，只是被GNU make使用。
- CFLAGS和LDLIBS变量用来加入编译器所需要的信息，这些信息用来定位和识别库文件。如果你有pkg-config，可以把反引号调用放在这里。例如，因为我几乎在所有的程序中使用Apophenia和Glib库，所以我系统中的makefile看起来是这样的：

```
CFLAGS='pkg-config --cflags apophenia glib-2.0' -g -Wall -std=gnu11 -O3
LDLIBS='pkg-config --libs apophenia glib-2.0'
```

或者，手工指定-I、-L和-l变量，如：

```
CFLAGS=-I/home/b/root/include -g -Wall -O3
LDLIBS=-L/home/b/root/lib -lweirdlib
```

- 当你在LIBS和CFLAGS行中添加了一个库的路径，并确定这个配置在你的系统上起作用时，一般很少有理由去移除这个配置。你真的

在乎最终的可执行文件可能会比你为每个程序都做一个个性化makefile所配置的大10KB吗？这意味着，你可以把你机器里常用的所有库都总结在一个makefile中，并把它从一个目录复制到另一个目录中而不需要分别进行重写。

- 如果你有第二个（或者更多）C文件，就添加类似second.o，third.o的内容到makefile OBJECTS段落头部的行中（不要加逗号，名字之间仅用空格）。make将用这些来决定哪个文件需要被制作以及以何种方式制作。
- 如果你的程序只有一个.c文件，你可能压根就不需要makefile。假设目录中现在有一个erf.c文件且没有makefile，可以用你的shell：

```
export CFLAGS='-g -Wall -O3 -std=gnu11'
export LDLIBS='-lm'
make erf
```

看看make如何用它的C编译知识来做余下的工作。

> 选择什么链接器选项来建立共享库？
>
> 说实话，我根本不知道。不同的类型和不同的年份的操作系统中都是不同的，甚至在同一个系统中，规则也经常有点混乱。
>
> 不过，我们将在第3章中介绍的工具Libtool了解每个操作系统中的每个共享库的每个制作过程的细节。我建议你花点时间去了解Autotools，那么就可以一举解决共享目标文件的编译问题，而不是花时间了解每种系统的正确编译器选项和链接过程。

1.5 以源文件的方式使用库

到目前为止，我们讲的都是如何用make来编译自己的代码。而编译

别人的代码则是另外一回事了。

让我们试一个简单的包，即包含大量数值计算函数的GNU科学计算库（GSL）。

GSL是用Autotools打包的，而Autotools是一个为在任何机器上都能使用函数库的工具的集合，其原理是测试所有已知的特殊细节并加上适当的解决方法。Autotools主要关注现代的代码是如何发布的，3.3“用Autotools打包你的代码”将详细讲述如何用它来打包你自己的程序和库。但是到现在，我们可以从用户的角度，来享受快速安装有用的库文件的便利。

GSL经常是由包管理器预编译好的，但是为了达到掌握编译库的每一步的目的，这里我们先得到GSL的源代码并手工把它安装好，假设你具有计算机的root权限。

```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-1.16.tar.gz ❶
tar xvzf gsl-*gz                               ❷
cd gsl-1.16
./configure                                    ❸
make
sudo make install                              ❹
```

❶ 下载源文件的zip压缩包。如果你还没有wget，使用包管理器安装它，或者在你的浏览器里直接键入这个URL。

❷ 解压缩包。x=抽取，v=详细模式，z=用gzip解包，f=文件。

❸ 检测。如果configure步骤给你一个“缺失一个元素”的出错信息，那么用包管理器来获取这个元素，然后重新运行configure。

❹ 将GSL安装到正确的位置上——需要具备相应权限。

如果你是在自己家里尝试这个，那么你可能已经有了root权限，上面步骤执行起来会很顺利。如果你是在工作场所并是在使用一个共享的服务器，则拥有超级用户权限的概率应该不会很大，由于最后一步要求超级用户权限，而你没有密码。如果这样，你就屏住呼吸直到下一部分吧。

它安装了吗？例1-3是一个使用GSL函数来找到95%的置信空间的小程序；编译一下这个例子，看看你能否把这个例子链接起来并运行。

例1-3　利用GSL来重做一次例1-1（gl_erf.c）

```
#include
#include

int main(){
    double bottom_tail = gsl_cdf_gaussian_P(-1.96, 1);
    printf("Area between [-1.96, 1.96]: %g\n", 1-2*bottom_tail);
}
```

为了使用刚才安装的库，我们需要修改会用到这个库的程序的makefile文件，以指定库及其位置。

你可以采用下列语句的任何一种，取决于你是否安装了pkg-config：

```
LDLIBS='pkg-config --libs gsl'
#or
LDLIBS=-lgsl -lgslcblas -lm
```

如果库没有被安装在标准的位置并且pkg-config也没有被安装，你

需要把路径加在定义行的开头，例如，CFLAGS=-I/usr/local/include和LDLIBS=-L/usr/local/lib –Wl, -R/usr/local/lib。

1.6 以源文件的方式使用库（即使你的系统管理员不想叫你这么做）

你在工作场所的共享机器里可能没有root权限，或者你被有特别的权限的管理员控制。那你就必须做点地下工作，制作一个属于自己的root目录。

第一步很简单，即创建这个目录：

```
mkdir ~/root
```

由于我已经有了一个～/tech目录，用来保存我所有的技术文档、手册和源代码，所以我建立的是～/tech/root目录。名字其实无所谓，但是我还是喜欢用～/root作为本书的示范目录。

提示

shell可以将波浪线替换为你个人主目录的完整路径，节省你很多打字的时间。POSIX标准只要求shell在第一个词或者冒号后的第一个字母（路径类型的变量会使用冒号）才这样处理，但是多数的shell扩展支持了词中间的波浪线。其他的程序，比如make，或许能，或许不能识别你个人主目录的波浪线。这种情况下，使用POSIX强制要求的HOME环境变量，如下面的例子所

> 示。

第二步，把新建的root系统添加到所有相关的路径上去。修改.bashrc （或其他shell的该配置文件）的PATH变量如下。

```
PATH=~/root/bin:$PATH
```

如果把你的新的目录的bin子目录添加在你的原来的PATH前面，这个子目录就将被首先查找到，并且你放在那里的任何程序都将被率先找到。这样你就可以把标准共享目录中的任何程序的替代版本放在那里。

对于那些你想链接的C程序库，请注意将新的搜索路径加入到在前面的makefile文件中：

```
LDLIBS=-L$(HOME)/root/lib       (plus the other flags, like -lgsl -lm ...)
CFLAGS=-I$(HOME)/root/include  (plus -g -Wall -O3 ...)
```

现在你已经有了一个本地的root目录，你也可以在别的系统上使用它，比如Java的CLASSPATH。

最后一步是在新的root目录上安装程序。如果你有源代码并使用Autotools，你只需要在合适的位置添--prefix=*$HOME*/root：

```
./configure --prefix=$HOME/root && make && make install
```

这些程序和库都在你的主目录中，所使用的许可权不会超出你所具有的范围，系统管理员不能抱怨你做了任何危害他人的事情。如果你的系统管理员还是有所抱怨，那么即使有点难过，你也应该和他分手了。

我记得以前的确有印刷版本的手册，不过现在它们都存在于man命令中了。例如，用man strtok来阅读关于strtok函数的内容，一般包括需要包含什么样的头文件、输入参数，以及它的基本用法的解释。手册文档倾向于简洁、明了，有时候缺乏示例，并且假设读者已经有了一些这个函数的基本用法的知识。如果你需要一个更加基本的教程，可以用常用的搜索引擎在Internet上找到几个（对于strtok这个例子，你可以参见9.1.4 “strtok的颂歌”）。GNU C库的手册，也很容易在网上找到，对初学者而言是非常易懂的。

- 如果你无法想起要找的函数的名字，每个手册页都有一个一行长的简述，man -k searchterm将搜索那些简述。许多系统还提供apropos命令，它和man -k类似但是多了一些别的功能。为了进一步的利用，我经常把apropos命令的输出用管道导出给grep命令。
- 手册分为几段。第1段是命令行命令，第3段是库函数。如果你的系统有一个命令行程序叫作printf，那么man printf将展示这个命令的文档，而man 3 printf将展示C库函数中的printf的文档。
- 如果想了解更多关于man命令的用法（比如各段的完整列表），可以使用man man。
- 你的文本编辑器或者IDE可能有某种快速打开手册页的方法。例如，vi的使用者可以把鼠标放在一个词语上，用*K*键来打来这个词语的手册页。

1.7 通过here来编译C程序

到此，你应该已经看出编译过程的模式了。

1．设定一个描述编译器选项的变量。

2．设定一个描述链接器选项的变量，包括为你用的所有函数库加上正确的-l选项。

3．用make命令或者你的IDE的操作来把这些变量转换为完整的编

译和链接命令。

本章的余下部分将把以上步骤最后做一次，并采取一种非常简短的设置：仅仅用shell。如果你思维敏捷，可以通过摘录一些语句段落到解析器上来学习脚本，你也将可以同样地把C代码贴在你的命令行上。

1.7.1 在命令行里包含头文件

gcc和Clang有一个用于包含头文件的方便的配置。例如：

```
gcc -include stdio.h
```

这和在C源文件的开头放入下面这行命令等价的：

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

同样地，也可以用clang -include stdio.h。

通过把以上内容加入到编译器调用，最终我们可以把hello.c程序写成只有一行：

```
int main(){ printf("Hello, world.\n"); }
```

通过以下方式顺利编译：

```
gcc -include stdio.h hello.c -o hi --std=gnu99 -Wall -g -O3
```

或者利用shell命令：

```
export CFLAGS='-g -Wall -include stdio.h'
export CC=c99
make hello
```

这个关于-include的窍门是随编译器而不同的，用于把信息从代码转移到编译指令中，如果你认为这是一个坏习惯，那好，就忘了它吧。

1.7.2 统一的头文件

请允许我在下面的几段中先跑一下题，去聊聊头文件。一个有用的头文件必须包含类型定义、宏定义、函数的类型声明、宏以及包含这个头文件的代码文件中使用的那些函数。同时，它不应该包含那些代码文件中并没有使用的类型定义、宏定义以及函数声明。

为了切实满足上面的要求，你需要为每一个单独的代码文件写独立的头文件，这个头文件只包含和代码文件相关的内容。但是没有人真的这样做。

很久以前，编译器处理一个哪怕不算复杂的程序要也要花费好几秒甚至几分钟的时间，所以减少一些编译器不得不做的工作，可以产生一些显而易见的好处。我现在的stdio.h和stdlib.h每个都有1000多行（可以用wc -l /usr/include/stdlib.h验证），time.h也有400行，这就意味着下面这个仅7行的小程序，实际上是一个大约2400行的程序。

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    srand(time(NULL));       // Initialize RNG seed.
    printf("%i\n", rand());  // Make one draw.
}
```

而今，编译器已经不再认为2400行是一个大问题，这种编译也就花费不到1秒。那么我们为何还要花时间去为一个特定的程序挑选正确的

头文件呢？还不如把更多内容放入一个头文件好。

你会看到一个使用Glib的例子，在头部包含了#include<glib.h>。这个头文件包含74个子头文件，覆盖了所有的Glib库的所有方面。Glib团队设计的这个用户界面非常好，因为那些不想花时间去挑选正确的头文件的用户可以只用一行就解决了包含头文件的问题。而那些需要精确控制的人也可以不用这一行，而去74个子头文件中挑选自己需要的。如果C语言的标准库也有这种快速、简单的头文件就好了；这并不是20世纪80年代的风格，但是做一个出来也很简单。

自己动手：为自己写一个通用的头文件，让我们称之为allheads.h，然后把你用过的所有头文件都扔到里面去，那么它看起来会像：

```
#include <math.h>

#include <time.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <gsl/gsl_rng.h>
```

我其实无法告诉你它确切长得什么样，因为我不知道你每天都用什么文件。

现在你有了这个集成的头文件，只需在所有文件的开头加上这么一句：

```
#include <allheads.h>
```

> 这样你就不用再操心头文件了。的确，加上这个头文件会扩展出10000行额外的代码，而且其中大多数和手头的程序无关。但是你不会留意到，而且没用的声明也不会改变最终的可执行文件。

如果你为其他用户写一个公共的头文件，那么按照规则，一个头文件是不应该包含不必要的元素的，你的头文件也许不应该包含用来读入全部标准库的声明和定义的#include "allhead.h"——事实上，有可能你的公共头文件中没有任何公共库的元素。通常来说这是真的：你的库有一个代码片段，使用了Glib的链表，但是这意味着你需要在代码文件中包含#include<glib.h>，而不是在公共的头文件中包含它。

让我们继续讨论在命令行上进行快速编译的话题，有一个统一的头文件可以让你更快地写出程序。一旦你有了一个统一的头文件，即使#include<allheads.h>这一行也是多余的，因为你可以把-include allheads.h加到CFLAGS这个环境变量中，这样你就不需要在你的项目中包含这个文件了。

1.7.3 here文档

here文档是POSIX标准shell的一个特性，你可以用在C、Python、Perl或者别的什么语言中，它们也使得这本书更加有用和有趣。并且，如果你想要有一个多语言的脚本，here文档是一个实现的方便法门。在Perl中做解析，在C语言中做算数，然后用Gnuplot产生漂亮的图片，并且集成在一个文本文件里。

这是一个Python的例子。通常，你通过下面的方法告诉Python去运

行一个脚本：

```
python your_script.py
```

你可以使用“-”以使stdin作为输入文件：

```
echo "print 'hi.'" | python -
```

理论上，你可以通过echo把一些长脚本放到命令行上，但是你很快就会看到有很多短小、不符合期望的解析在进行——比如，你可能需要用\"hi\"而不是"hi"。

那么，here文档实际上根本就不会被解析。比如：

```
python - <<"XXXX"
lines=2
print "\nThis script is %i lines long.\n" %(lines,)
XXXX
```

- here文档是一项标准的shell特性，所以它们应该在任何POSIX系统中都可以用。
- “*XXXX*”代表你想用的任何字符串；“EOF”也很流行，即使“-----”也行，只要你保证连字符的数量在顶部和底部都相等。当shell看到你选定的字符串单独占一行，它会停止向stdin输出脚本。这就是一般脚本解析的过程。
- 还有以<<-开始的变体。这种变体去掉了所有每行开头的Tab键，所以你可以把一个渐次缩进的shell脚本文档放在这里。当然，这对Python的嵌入文档而言是一个灾难。
- 作为另一个变体，<<"*XXXX*"和<<*XXXX*是不同的。在第二个版本中，shell解析一定的元素，意味着你可以让shell为你插入

$shell_variables的值。shell非常依赖$来表达变量和其他的扩展；$是标准键盘上很少的几个在C中没有特殊含义的字符之一。这大约是那些从白手起家建立UNIX的人为了容易写出一个shell去产生C代码才这么干的……

1.7.4 从stdin中编译

现在回到C的世界中来：我们可以用here文档来通过gcc或Clang编译那些贴到命令行的C代码，或者在一个多语言的脚本插入几行C语言程序。

我们不再用makefile，所以我们需要一个单独的编译命令。为了让生活变得不那么痛苦，让我们给它起个别名。把它粘贴到你的命令行，或者把它添加在你的.bashrc、.zshrc中，或者任何适合的地方。

```
go_libs="-lm"
go_flags="-g -Wall -include allheads.h -O3"
alias go_c="c99 -xc - $go_libs $go_flags"
```

这里的allheads.h是你之前放在一起的集成的头文件。当你写C代码的时候，用-include选项是你最不应该考虑的事情，并且我也发现当C代码中出现#字符的时候，bash的history命令会受到影响。

在编译行，用'-'取代了文件名，这意味着用stdin而不是从一个命名的文件输入。-xc认为这是C代码，因为gcc代表GNU编译器组合，而不是GNU C编译器，也没有类似.c的输入文件名来提示它，所以我们必须指明这不是Java、Fortran、Objective C、Ada，或者C++（对Clang也一样，即使它的名字是有意在提示是C语言）。

无论你在makefile中对LDLIBS和CFLAGS做了多少定制化，这里仍要做。现在你已经扬帆出海了，可以在命令行中编译C代码了：

```
go_c << '---'
int main(){printf("Hello from the command line.\n");}
---
./a.out
```

我们可以使用here文档来粘贴简短的C程序到命令行，并写一点带有争议性的测试程序。不仅是你不需要一个makefile，你甚至也不需要一个输入文件。

不要把这种模式当成你的主要工作状态。但是剪切、粘贴代码段落到命令行是有趣的，并且在一个较长的shell脚本内可以用一步完成C也是非常神奇的。

---

[1] Cygwin有Red Hat, Inc.运营的项目，该公司也允许用户购买不按照GPL版权发布自己源代码的权利。

[2] 虽然MinGW有一个包管理器，可以安装一些基本的系统和一些库（大部分库都是MinGW自己需要的）。和一些其他包管理器相比，这些包的数量与典型的包管理器提供的上百个包相比显得很苍白。事实上，Linux包管理器提供的编译完的库比MinGW包管理器提供的多。写作本书的时候是这样，不过当你阅读本书的时候，有些用户也许向MinGW添加了更多的包了。

[3] 你可以在任何POSIX标准系统中试一下find /-type f | wc –l，用来得到一个粗略的文件数。

第2章　调试、测试和文档

本章讲述用于调试、测试和编写文档的工具——这些工具可以使你的代码从有用的草稿提升到大家可以依赖的产品。

C给了你对内存直接操作的自由，有的时候你会误用这种自由而对内存做一些蠢事。调试意味着不仅要针对程序逻辑问题例行检查（用GDB），还要检查包括内存滥用和泄漏等更加技术的问题（用Valgrind）。在文档编写方面，本章从界面的层次讲述了一个文档管理的工具（Doxygen），以及另外一个帮助你针对程序的每一步归档的工具（CWEB）。

本章概要介绍了测试环境，可以帮助你为自己的代码快速地写出很多测试。本章还包括一些关于错误报告的内容以及如何处理输入和用户错误。

2.1　使用调试器

先给大家一个简洁、明了的使用调试器的建议，那就是：

永远使用调试器。

可能有些读者认为这算不上什么提示，难道还有人真的不用调试器吗？本书已经是第二版了，我告诉你关于第一版读者提出的最多的请求是多介绍一下调试器，因为对很多读者来说，这一部分的内容对他们来

说是全新的。

另外，我发现有些人担心，bug 来源于我们对大的方面的问题的误解，而调试器只能给出来自底层的变量状态变化和回溯的信息。的确，当你用调试器精确定位了一个bug，是值得花时间去研究你所发现的问题根源和对错误的理解，另外就是这个bug是否会在代码中重现。一些死亡证明会包括对于死因的大胆判断：检验对象的死因为_____，因为_____，因为_____，因为_____，因为_____。当你利用调试器建立了对自身代码的更深刻的理解之后，你可以把这些理解用在更多的单元测试中。

注意前面所说的“永远”：在一个调试器的环境下运行程序实际上不会产生什么代价。调试器也不是程序崩溃的时候才去使用的东西。Linus Torvalds 解释说：“我一直都用gdb……把它作为可以破解程序编码意图的一个强化版的反编译器来使用。”你会觉得下面这些功能很不错：例如可以在任何地方暂停下来；利用一个快速的print verbose++来提升信息输出级别；通过print i=100和continue来强制一个像for （int i=0; i<10; i++）的循环超出运行范围；或者通过加入一系列的输入来测试一个函数。那些交互式语言的粉丝一直这么认为：与代码互动将持续地改进你的开发过程；不过可惜的是，他们却从来没有坚持读到C语言教科书的调试器那一章，因此也从来没有意识到C语言也支持那些互动的特性。

不管你的意图如何，你终归需要那种编译到程序里、对任何调试者都有用的人类可理解的调试信息（变量名和函数名）。为了包含调试符号，可在编译器开关（即CFLAGS变量）中加入-g选项。事实上没什么

理由不使用-g选项——它并不会降低你的程序运行速度，并且你的可执行文件所增加的1KB在多数情况下也没什么问题。当你使用-O0（O零）选项关掉编译器的优化后，调试会变得相对简单，因为优化有的时候会去掉很多的变量，或者将整个的代码结构以令人吃惊的方式进行重整和改变。

本章只介绍GDB调试工具，因为在多数POSIX系统中，这是你“唯一的选择”。（顺便说一下，一个C++编译器会将代码命名重整。使用gdb的时候，我发现在gdb输出的那些提示中进行C++的调试是一件非常痛苦的事。因为C代码不进行命名重整，所以gdb对调试C语言更加有用，而且我们也不需要一个GUI来将那些被重整的命名恢复。）LLDB（与LLVM/clang伴生的）正在流行起来，我也会介绍一些。苹果在Xcode套件中已经停止发布了gdb，但是你可以通过包管理器如Macports、Fink或者Homebrew来安装它。在Mac电脑上，你可能需要通过sudo（!）来运行调试程序，如sudo lldb stddev_bugged。

如果你是从一个IDE或者其他的图形前端中运行你的程序，你也许可以单击“run”菜单就能在调试模式下运行你的程序了。这里我会介绍GDB命令行中的命令，使用IDE的读者可以轻松地将这些命令转换成屏幕上的鼠标操作。依赖于不同的前端，你也可以使用.gdbinit中定义的宏。

在命令行中使用GDB时，你可能需要一个在另外的窗口或终端显示的文本编辑器，以便同时显示你的代码。简单的GDB/编辑器组合就已经可以带来一个IDE所能提供的很多便利，而且足够你使用的了。

堆栈帧

为了启动程序，我们要求系统运行一个叫作main的函数。此时计算机会产生一个帧，并把main函数的信息放在里面，比如输入参数（对于main函数习惯上被命名为argc和argv）和函数内部生成的变量。

假设在main的执行过程中调用另外一个函数get_agents，main的执行就会停止，并且为get_agents产生一个新的帧，用来存放各种细节和变量。也许get_agents又调用另一个函数agent_address，这样我们就得到了一个不断增长的包含很多帧的堆栈。最终，agent_address结束执行，函数对应的帧将被堆栈弹出，接下来get_agent函数继续执行。

那么如果你的问题只是：“我在哪里？”简单的回答是看源代码中的行号即可，有时这也正是你所需要的。但是更多的情况下，你可能会问：“我怎么走到这里的？”这个问题的答案就是回溯，也就是一个堆栈帧的列表。这里是一个使用backtrace命令的示例：

```
#0 0x00413bbe in agent_address (agent_number=312) at addresses.c:100

#1 0x004148b6 in get_agents () at addresses.c:163

#2 0x00404f9b in main (argc=1, argv=0x7fffffffe278) at addresses.c:227
```

堆栈的顶端是0帧，从0开始到main的时候已经是第2帧（这个值将随着堆栈的增长和缩减而变化）。帧号后的十六进制数是一个地址，被调用函数返回的时候，程序执行将返回到这个地址；我总是把它们当作视觉噪音而忽略。地址后面是函数名、它的输入参数（恰好argv也是一个十六进制地址），以及当前执行到的源代码行号。

如果你发现agent_address列出的地址信息明显是错的，那么也许agent_number输入也可能是错的，此时你不得不跳到1帧去查看是怎样的get_agents状态导致把agent_address设定为这样一个奇怪的状态。很多分析程序的技巧就是跳入堆栈中，从一个个的函数的堆栈帧中追踪因果关系。

2.1.1 调试的侦探故事

本节会进行一场虚拟的关于GDB和LLDB的问答。在本书的示例代码中，你会发现stddev-bugged.c文件，一个重新书写的有一个bug的例子

7-4代码。改动是很小的，以至于你可以参考stddev.c这个原始的文件，来了解一下哪里进行了改动。就像好的侦探故事一样，你可以利用那些能够用于识别出罪犯的线索。所有的问题都可以帮助你缩小嫌疑人范围，直到最后的嫌疑人被识别出来，并且使得bug变得清晰起来。

当你编译完这个程序后（使用CFLAGS="-g" make stddev_bugged命令），我们开始向调试器提问：

```
gdb stddev_bugged
# or
lldb stddev_bugged
```

现在我们有了调试器命令提示符，准备开始询问了。

问题：这个程序是做什么的？

回答：run命令会运行这个程序。这里GDB和LLDB的命令是相同的，在这个例子中我使用的是GDB的命令，如果使用了LLDB的命令，我会用方括号把它们括起来。与所有的GDB和LLDB命令一样，它可以被缩写。

```
(gdb) r

mean: 5687.496667 var: 194085710
mean: 0.83 var: 4.1334
[Inferior 1 (process 22734) exited normally]
```

看起来程序产生了一些平均值和方差。程序看来运行完毕了，没有产生segfault（段错误）和其他故障，主程序返回了0，说明是正常的执行。

问题：main中的代码和我们得到的那些输出结果是否一致？

回答：查看代码最简单的方法是使用一个文本编辑器。即使你登录到一个只有终端显示的远程计算机上，你也可以使得你的调试器和编辑器并排地显示。（参考第3章的复用器）GDB和LLDB可以使用list命令来显示你的代码。

```
(gdb) l main
28          }
29          return (meanvar){.mean = avg,
30                           .var = avg2 - pow(avg, 2)}; //E[x^2] - E^2[x]
31      }
32
33      int main(){
34          double d[] = { 34124.75, 34124.48,
35                         34124.90, 34125.31,
36                         34125.05, 34124.98, NAN};
37
```

以我们要求的行数为中心得到了10行的代码，重新不带参数运行list会显示下面的10行代码。

```
(gdb) l
38          meanvar mv = mean_and_var(d);
39          printf("mean: %.10g var: %.10g\n", mv.mean, mv.var*6/5.);
40
41          double d2[] = { 4.75, 4.48,
42                          4.90, 5.31,
43                          5.05, 4.98, NAN};
44
45          mv = mean_and_var(d2);
46          mv.var *= 6./5;
47          printf("mean: %.10g var: %.10g\n", mv.mean, mv.var);
```

我们看到在第38行调用了mean_and_var函数，我们传入了一个数组参数d。问题在于，数组d中的数大约是34125，但是返回的返回值是5678（方差也是不对的）。同样，第二次调用的时候，数组中的数大约

是5，但是返回的平均值却是0.83。

本节的剩余部分其实是在问同一个问题，错误开始于代码中的哪一行？为了回答这个问题，我们需要更多的细节。

问题：怎样才能知道mean_and_var函数里面发生了什么？

答案：如果想让程序在mean_and_var函数暂停，我们需要在那里放置一个断点：

```
(gdb) b mean_and_var
Breakpoint 1 at 0x400820: file stddev_bugged.c, line 16.
```

有了断点以后，重新运行这个程序，这个时候程序就会在那个断点停下来了：

```
(gdb) r
Breakpoint 1, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugg
ed.c:16
16     meanvar mean_and_var(const double *data){
(gdb)
```

我们现在在第16行，函数的头部，可以看看函数内部发生了什么。

问题：data是正确的吗？

回答：通过print命令，我们可以查看当前帧下的数据，也可以把这个命令缩写成p：

```
(gdb) p *data
$2 = 34124.75
```

有点失望了：我们只得到了第一个元素。但是GDB有一个特殊的语

法@，可以显示数组中的一系列数据。比如显示10个数据[LLDB: mem read-tdouble-c10 data]。

```
(gdb) p *data@10
$3 =   {34124.75,
  34124.480000000003,
  34124.900000000001,
  34125.309999999998,
  34125.050000000003,
  34124.980000000003,
  nan(0x8000000000000),
  7.7074240751234461e-322,
  4.9406564584124654e-324,
  2.0734299798669383e-317}
```

请注意表达式前面有一个星号，如果没有这个星号，我们将只获得10个连续的十六进制的地址值。

我输出了10个元素，因为我也不知道这个数组里面到底有多少个元素，但是前7个元素是对的：一系列数字，后面跟一个nan标志。这之后是一些没有经过初始化的噪声数据。

问题：这和我们从main中送进去的数据一致吗？

回答：我们可以通过bt命令来回退：

```
(gdb) bt
#0 mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:16
#1 0x0000000000400680 in main () at stddev_bugged.c:38
```

堆栈里面目前有两帧，包含当前的帧和它的调用者main帧。让我们查看在第1帧中的数据，首先切换到第1帧。

```
(gdb) f 1
#1 0x0000000000400680 in main () at stddev_bugged.c:38
```

```
38          meanvar mv = mean_and_var(d);
```

目前调试器在main这一帧，第38行。38行是我们期望的行数，执行的顺序看来是正确的（没有被优化器打乱执行的顺序）。在这一帧，数据数组被命名为d：

```
(gdb) p *d@7
$5 =   {34124.75,
  34124.480000000003,
  34124.900000000001,
  34125.309999999998,
  34125.050000000003,
  34124.980000000003,
  nan(0x8000000000000)}
```

这与mean_and_var的帧中的数据看起来一致，所以在数据方面看来没发生什么错误。

如果想继续运行我们的程序，我们不需要显式的回到第0帧。但是如果你想这么做，你可以通过f 0或者可以在当前的退栈结构中进行移动：

```
(gdb) down
```

注意up和down代表数字的顺序。给定用bt命令（在GDB或者LLDB中都存在）返回的列表将数值小的帧排列在列表的最上头，up命令在列表中向下走，而down命令会在列表中向上走。

问题：是多线程造成的问题吗？

回答：我们可以通过info threads[LLDB:thread list]来获得线程的列表：

```
(gdb) info threads
  Id   Target Id         Frame
  * 1    Thread 0x7ffff7fcb7c0 (LWP 28903) "stddev_bugged" mean_and_var
                      (data=data@entry=0x7fffffffe180)at stddev_bugged.c:16
```

本例中，只有一个活跃的线程，所以这不是一个多线程的问题。*代表目前我们的调试器在哪个线程。如果有两个线程，我们可以通过GDB的命令 thread 2 或者是LLDB的命令 thread select 2来切换过去。

提示

现在你的程序并没有产生大量的新线程，当你读完第12章后才会产生大量的线程。GDB用户可以把这一行加到.gdbinit中来关掉那些为每一个新线程产生的讨厌的通知。

```
set print thread-events off
```

问题：mean_and_var函数是做什么的？

答案：我们可以重复单步运行程序。

```
(gdb) n
18          avg2 = 0;
(gdb) n
16      meanvar mean_and_var(const double *data){
```

没有什么输入，只是单击回车键代表着重复上一个命令，所以我们不需要键入n。

```
(gdb)
18                avg2 = 0;
(gdb)
20          size_t count= 0;
(gdb)
16      meanvar mean_and_var(const double *data){
(gdb)
21          for(size_t i=0; !isnan(data[i]); i++){
(gdb)
21          for(size_t i=0; !isnan(data[i]); i++){
(gdb)
22              ratio = count/(count+1);
(gdb)
26              avg += data[i]/(count +0.0);
```

行数代表着程序正在跳转。这是因为在每一步，调试器在执行机器层次的指令，这个指令和产生他们的C语言并不一定对应。即使你关掉了优化器，这也是正常的。跳转指令也会影响变量的值，它们的值也许现在并不可靠，需要我们对这一行第二次或者第三次的单击才行。

有另外的选项进行单步执行，典型的是snuc这四个命令中的一个；参考表2-1。但是这样单步运行也许需要花费一天的时间，我们看到有一个for循环用来处理数据，所以我们在循环的中间设置一个断点。

```
(gdb) b 25
Breakpoint 2 at 0x400875: file stddev_bugged.c, line 25.
```

现在我们有两个断点了，我们可以通过GDB info break或者是LLDB的break list命令来查看它们：

```
(gdb) info break
Num     Type          Disp Enb Address            What
1       breakpoint    keep y   0x0000000000400820 in mean_and_var
                                                  at stddev_bugged.c:16
        breakpoint already hit 1 time
2       breakpoint    keep y   0x0000000000400875 in mean_and_var
                                                  at stddev_bugged.c:25
```

我们不再需要在mean_and_var头部的那个断点了，所以可以使它处于不活跃状态[LLDB: break dis 1]：

```
(gdb) dis 1
```

这样，info break 命令输出的显示中，Enb列中断点1将来会是n。如果需要，你可以以后重新用GDB enable 1命令或者是LLDB的break enable 1来重新使得它们有效。或者你知道再也不需要它了，那么你可以通过GDB的del或者是LLDB的break del 1来彻底删除这个断点。

问题：在循环的中间，变量的情况怎么样？

回答：我们可以通过r来重新开始，或者通过c来继续运行我们的程序。

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugg
ed.c:25
25           avg2 *= ratio;
```

现在我们停在了第25行，我们可以查看本地的变量表[LLDB:frame variable]：

```
(gdb) info local
i = 0
avg = 0
avg2 = 0
ratio = 0
count = 1
```

我们也可以通过GDB的命令 info args来检查输入的参数，虽然我们已经查看过了数据。LLDB的命令 frame variable 包含局部数据和输入的

参数。

问题：我们知道输出的mean是错误的，那么在每次运行的过程中avg是如何变化的？

答案：我们可以在程序每次停留在断点处的时候键入 p avg，不过显示命令可以自动化地帮我们完成这个工作：

```
(gdb) disp avg
1: avg = 0
```

现在我们继续，调试器会遍历这个循环，而且每次停留在断点的时候，我们都能看见变量avg的值了。

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugg
ed.c:25
25              avg2 *= ratio;
1: avg = 0

(gdb)
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugg
ed.c:25
25              avg2 *= ratio;
1: avg = 0
```

这就不对了，代码有下面的一行：

```
avg *= ratio;
...
avg += data[i]/(count +0.0);
```

所以avg应该在每次循环的时候都会改变才对，但是它一直为零。现在知道那里出了问题，所以我们不再需要显示avg这个变量了，我们可以通过undisp 1来关闭自动显示。

问题：如何检查输入给avg的那些变量？

答案：我们验证了data是对的，那么变量ratio 和count呢?

```
(gdb) disp ratio
2: ratio = 0

(gdb) disp count
3: count = 3
```

遍历循环几次，我们可以看到count在正常增加，但是ratio没有改变。

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugg
ed.c:25
25              avg2 *= ratio;
3: count = 4
2: ratio = 0
```

问题：ratio是在哪里被设置的？

回答：通过代码编辑器或者l命令检查代码，发现ratio只在第22行中被设置：

```
ratio = count/(count+1);
```

我们已经验证了count是按照我们的预期递增了。但是这一行一定有别的地方错了。目前错误很明显，如果count是一个整数，那么count/（count+1）是一个整数除法，这样会返回一个整数（3/4=0），而不是我们在小学里面学过的浮点数的除法（3/4=0.75）。为了修改这个错误，我们需要保证分子或者分母其中有一个是浮点数，可以通过把整形的常数1换成浮点型常数1.0来达到这个目的：

```
ratio = count/(count+1.0);
```

调试器不会就这种一般错误提醒我们，但是它可以帮我们发现在代码中第一个出现错误的地方，在一行中发现错误比在一个50行的程序块中发现错误更简单。利用这个方法，我们检查并验证了关于代码的各种细节，更好地理解了程序的流程以及程序的堆栈结构。

表2-1提供了更多的常用的调试命令，GDB和LLDB都有很多调试命令，但是90%的时间你只会用到其中的10%。命令中的变量名字大多来源于“New York Time”头条下载器程序，这个程序在13.5“libxml和cURL”一节。

表**2-1**

组	命令	含义
运行	run	从开始运行程序
	run args	从开始运行程序，并带有指定的命令行参数
停止	b get_rss	在特定的函数暂停
	b nyt_feeds.c:105	在代码的特定行前面暂停
	break 105	就像b nyt_feeds.c:105命令一样，如果你已经停留在bnyt_ feeds.c文件了，那么你可以用break 105。这将使程序在执行105行之前停下来
	info break [GDB] break list [LLDB]	列出断点
	watch curl [GDB] watch set var curl [LLDB]	如果给定的变量的值改动了，那么暂停
	dis 3 / ena 3 / del 3 [GDB] break dis 3 / break ena 3 / break del 3	比如disable 3（这里3是你从info break中得到的断点号），随后可用enable 3来重新打开它。如果你有很多断点集合，直接使用disable可以把它们全部关掉，然后可以在需要的时候把它们中的一两个打开 用del 3来彻底删除一个断点

	[LLDB]	
检查变量	p url	显示url变量的值。你也可以指定表达式，包括函数调用
	p *an_array@10 [GDB]	显示数组an_array的前10个元素，接下来的10个元素可以使用*（an_array+10）@10来显示
	mem read -tdouble -c10 an_array	从数组an_array中读入10个类型为double的元素，接下来的10个元素是：mem read -t double -c10an_array+10
	info args / info vars [GDB]	得到一个函数所有参数的值和全部局部变量
	frame var [LLDB]	得到一个函数所有参数的值和全部局部变量
	disp url	每次程序停止的时候，显示url的值
	undisp 3	关闭显示元素3，GDB：如果没有数字，那么就关闭所有的显示
线程	info thread [GDB] thread list [LLDB]	列出活跃的线程
	thread 2 [GDB] thread select 2 [LLDB]	切换到线程2
帧	bt	列举出堆栈帧
	f 3	查看帧3
	up / down	在堆栈的帧中上下移动
单步	s	step：单步执行一行，甚至包括进入另一个函数
	n	next：下一行，但是不进入子函数，并且可能跳回一个循环的开头
	u	Until：从当前行开始运行到指定的行 （这样可以让一个已经进入循环的循环体直接运行到循环体外面的语句）
	c	continue：继续运行直到下一个断点或者程序的结尾
	ret or ret 3 [GDB]	从当前的函数立即返回，并且给出返回值
	j 105 [GDB]	跳到任何你喜欢的一行（需要合理的一行才行）
查看代码	l	list（或者l）来得到一个当前行附近10行的代码
重复	Enter	按Enter键将重复最近的一条命令，这使得单步执行变得更容易，或者在l命令后使用，以列出下一个10行

编译	make [GDB]	不用离开GDB而运行make，你也可以指定一个目标，如make myprog
获取帮助	help	查看调试器的帮助文件

2.1.2 GDB变量

本节介绍GDB中一些有用的特性，这些可以帮助你尽可能方便地查看数据。下面所有的命令都来自于GDB命令行；基于GDB的IDE调试器也经常会提供可以调用这些特性的方法。

这里有个什么都不做的程序例子，可以用来满足你审视变量的目的。因为这是一个什么都不做的程序，要确保把编译器优化选项设定为-O0，否则x就会整个消失。

```
int main(){
    int x[20] = {};
    x[0] = 3;
}
```

第一个技巧只有那些从来不去读GDB手册[Stallman, 2002]的人才不知道，但是可能所有的人都没有读过这个手册。你可以生成一个方便的变量，来节省你的键入次数。例如，如果你想查看一个在很深层次结构中的一个元素，你可以这样做。

```
(gdb) set $vd = my_model->dataset->vector->data
p *$vd@10

(lldb) p double *$vd = my_model->dataset->vector->data
mem read -tdouble -c10 $vd
```

第一行产生了一个方便的变量用来代替那一长串路径。遵循shell的风格，美元符号代表一个变量。同时与shell又不同，GDB在第一次使用变量的时候使用set和美元符号，LLDB使用clang’s解析器来计算表达式，所以LLDB的声明是一个典型的C语言声明。两个调试器的第二行都演示了一个简单的应用。这里我们没有节省很多键盘的键入，但是如果你怀疑一个变量有错误的行为，那么给它一个短的名字，在以后彻底调查这个变量的时候就显得很有用了。

它们不仅是名字，还是你可以修改的变量。在我们的什么都不做的程序的第3行或者第4行中断，然后尝试：

```
(gdb) set $ptr=&x[3]
p *$ptr = 8
p *($ptr++)  #print the pointee, and step forward one

(lldb) p int *$ptr = &x[3]
p *$ptr = 8
p *($ptr++)
```

第二行在给定的位置改变了变量的值。参见6.3.4“你需要知道的各种指针运算”，将指针加1后指向了数组中的下一个元素，所以第三行以后，$ptr指向x[4]。

给指针加1这种形式特别有用，我们只要不停地按Enter键就可以重复上一条命令而无须再次输入。因为指针单步向前移动，你每次敲击Enter键都可以得到下一个值，直到你查看了全部的数组。在你处理一个链表时，这也是很有用的。假设你有一个函数show_structure，用来显示链表中的一个元素，并设定$list等于一个给定的元素，而且我们有链表的头list_head。那么：

```
p $list=list_head
show_structure $list->next
```

单击Enter键将单步走过整个链表。随后我们用假想函数来显示一个现实中的数据结构。

但是，首先这里有另一个关于那些$开头的变量的技巧。让我从其他地方剪切复制几行代码来演示与调试器的互动吧：

```
(gdb|lldb) p x+3
$17 = (int *) 0xbffff9a4
```

你可能根本不想再看它了，但是请注意，如何使得print输出的信息从$17开始。的确，每个输出都被分配了一个变量名，以至于我们可以像这样用：

```
(gdb|lldb) p *$17
$18 = 8
(gdb|lldb) p *$17+20
$19 = 28
```

为了能更简洁，GDB用一个单独的$代表一个简化变量，这个简化变量就是最后输出的那个变量。所以，如果当你得到一个十六进制的地址并且想下一步得到这个地址上的值的时候，就把p *$放在下一行来得到这个值。通过这个方式，上面的步骤可以为：

```
(gdb) p x+3
$20 = (int *) 0xbffff9a4
(gdb) p *$
$21 = 8
(gdb) p $+20
$22 = 28
```

2.1.3 打印结构

GDB允许你定义简单的宏，这一点在显示常见数据结构时非常有用——这也恰好是调试器最常做的工作。即使是一个简单的二维数组，当它显示成一长串数字的时候，也会刺伤你的双眼。在完美的世界中，每个你常用到的结构都会有一个对应的调试器命令，用来以某种对你真正有用的方式来快速查看这个结构。

这些工具是非常原始的，但是你可能已经写了一个C侧的函数用于输出你需要处理的复杂结构，那么利用宏就可以通过简单地敲几下键盘来调用那个函数。

在GDB的命令提示后你不能使用任何C预处理器宏，因为它们在调试器能看到你的代码的时候就已经被替换了。所以如果你在代码中有个有价值的宏，你可能还需要在GDB中重新实现一次。

这里有一个函数，你可以试一下在parse函数（参见13.5“libxml和cURL”）中间设置一个断点，在那个断点处你会得到一个doc结构，这个结构代表一个 XML树。现在你把这些宏放在.gdbinit中。

```
define pxml
    p xmlElemDump(stdout, $arg0, xmlDocGetRootElement($arg0))
end
document pxml
Print the tree of an already opened XML document (i.e., an xmlDocPtr) to t
he
screen. This will probably be several pages long.
E.g., given: xmlDocPtr doc = xmlParseFile(infile);
use: pxml doc
end
```

请注意在这个函数之后的文档；可以用help pxml或help user-defined

来查看。宏的作用只是为了节省一点键盘输入，但是因为调试器的原始用途就是为了查看数据，如果你反复地查看一些数据，那么这些小的好处就积少成多了。

我下面会讨论LLDB版本中对应的宏。

GLib有一个链表结构，所以我们应该有一个链表查看器。例2-1的代码实现了这个查看器，查看器中包括两个用户可见的宏（phead用来查看链表的头，而pnext用来单步向前查看），还有一个用户不会看到的宏（plistdata，实现了在phead和pnext中一些共同的功能）。

例2-1　一组用于在GDB中方便地显示链表的宏——这是你可能会用到的最善于阐述的调试宏（gdb_showlist）

```
define phead
    set $ptr = $arg1
    plistdata $arg0
end
document phead
Print the first element of a list. E.g., given the declaration
    Glist *datalist;
    g_list_add(datalist, "Hello");
view the list with something like
gdb> phead char datalist
gdb> pnext char
gdb> pnext char
This macro defines $ptr as the current pointed-to list struct,
and $pdata as the data in that list element.
end

define pnext
    set $ptr = $ptr->next
    plistdata $arg0
end
document pnext
You need to call phead first; that will set $ptr.
This macro will step forward in the list, then show the value at
that next element. Give the type of the list data as the only argument.
```

```
This macro defines $ptr as the current pointed-to list struct, and
$pdata as the data in that list element.
end

define plistdata
    if $ptr
        set $pdata = $ptr->data
    else
        set $pdata= 0
    end
    if $pdata
        p ($arg0*)$pdata
    else
        p "NULL"
    end
end
document plistdata
This is intended to be used by phead and pnext, q.v. It sets $pdata and pr
ints its value.
end
```

例2-2是一段简单的程序，在代码中使用双向链表GList来存储char*。我们可以在程序的第8或第9行执行断点，然后调用上一个宏。

例2-2　一些尝试调试的示例代码，或者是一个对Glib链表的快速介绍（glist.c）

```
#include
#include

GList *list;

int main(){
    list = g_list_append(list, "a");
    list = g_list_append(list, "b");
    list = g_list_append(list, "c");

    for ( ; list!= NULL; list=list->next)
        printf("%s\n", (char*)list->data);
}
```

提示

你可以定义一个函数在某个命令执行之前或之后运行。例如：

```
define hook-print

echo <----\n

end



define hookpost-print

echo ---->\n

end
```

将在任何你打印的东西前后加上可爱的括号。最令人兴奋的钩子函数是hook-stop。display命令在每次程序停止的时候会显示某个表达式的值，不过如果你想在程序停止的时候使用某个宏或者是某个GDB命令，需要重新定义hook-stop：

```
define hook-stop

pxml suspect_tree

end
```

当你分析完某个可疑变量后，重新把hook-stop定义为空函数即可：

```
define hook-stop

end
```

LLDB用户：see target stop-hook add。

自己动手：

GDB宏可以包含while循环，它与例2-2中的if语句看起来很像（以类似while $ptr开始，并且以end结尾）。请以此为例来写一个宏，实现一次性打印整个链表。

LLDB在实现上有点不一样。

首先，你可能已经注意到LLDB命令描述性更好一些，因为作者期望你给你经常使用的命令起一个专属的别名。例如你可以给print double或者int 类型的数组起下面的别名：

```
(lldb) command alias dp memory read -tdouble -c%1
command alias ip memory read -tint -c%1

# Usage:
dp 10 data
ip 10 idata
```

别名机制用来将存在的命令进行简写。没有办法给一个别名命令赋予一个帮助字符串，因为LLDB会回收完整命令关联的字符串。为了能像GDB那样在LLDB里面写宏，LLDB使用了正则表达式。

这是放到.lldbinit里面的LLDB版本：

```
command regex pxml
          's/(.+)/p xmlElemDump(stdout, %1, xmlDocGetRootElement(%1))/'
          -h "Dump the contents of an XML tree."
```

关于正则表达式的完整讨论超出了本书的范围（线上有上百个正则表达式的教程），这里你只需要知道在第一个斜杠和第二个斜杠之间的括号的内容会被插入到第二个斜杠和第三个斜杠之间的%1标志的那个位置。

代码分析

不管程序现在运行多快，我们仍然期望它更快一点。在多数语言中，头号建议是用C重写一遍，但是现在你已经是在用C语言写程序了。下一步就是找到最花费时间的函数，对它优化得越多，我们的回报也越多。

首先，在gcc或icc的CFLAGS中添加-pg（是的，这是存在编译器差异的；gcc将为gprof预处理你的程序；Intel的编译器将为prof预处理你的程序，并且是和我在这里给出的gcc特有的细节很类似的流程）。通过这个选项，你的程序将每几毫秒就暂停一下，并记下当前是在运行在哪个函数中。这个记录以二进制的方式记录在gmon.out中。

只有可执行文件被分析，连接的库不会被分析。因此，如果你需要在运行一个测试程序时分析一个库，必须把所有库和程序中的代码复制到一个地方并且把所有的一切编译成一个大的可执行文件。

运行程序之后，调用gprof *your_program* > *profile*，然后在文本编辑器里打开profile文件。这里有一个适合阅读的列表，包括函数、它们的调用，以及程序在每个函数上所花费的时间的百分比。当你发现瓶颈在哪里的时候，也许你会觉得很惊讶。

2.2 利用Valgrind检查错误

我们花在调试上的时间主要用来找到程序中第一个出现问题的点。好的代码和系统会为你找到那个点。也就是说，一个好的系统会让失败尽早地出现。

在这方面C的表现毁誉参半。在某些语言中，类似conut=15的笔误

（正确的应该是count=15）会产生一个新的变量，而且和你想要的那个count完全没有什么关系；而在C中，在编译的那一步就失败了。另一方面，C却会允许你把一个只含有9个元素的数组的第10 个元素赋值，然后你可能花了很久的时间才发现第10个元素完全是一堆垃圾。

管理内存时发生的失误实在非常让人恼火，所以已经有很多工具来处理它们。这其中的大赢家就是Valgrind。它在POXIS系统上（包含OS X）是通用的，用你的包管理器安装一个吧。Windows的用户可以试一下Dr. Memory。

Valgrind运行一个虚拟机，可以比真实的机器更好地标识内存，所以它可以知道你使用了一个含有9个元素的数组的第10个元素。

在编译完程序（当然，你需要使用gcc或Clang的-g选项来包含调试符号）后，运行下面的命令：

```
valgrind your_program
```

如果程序存在错误，Valgrind将给你两个回溯信息，看起来非常像调试器给你的那样。第一个是内存滥用被首次探测到的地方，第二个是Valgrind尽量猜测的这个内存滥用发生的代码行数，比如被重复释放的地方，以及最近的采用malloc分配内存的地方。这些错误经常是非常微妙的，因此，给你一个确切的代码行来研究其实已经帮助你在寻找bug的征途中走过了好长的一段路了。Valgrind的开发很活跃——对程序员而言没有比写一个编程工具更喜欢的了——所以令我惊奇的是，随着时间的推移，越来越多的信息被汇报出来，而且看起来未来还会变得更好。

为了演示Valgrind的回溯功能，我在例9-1的代码中插入一个错误。也就是重复了第14行，free（cmd），这样会促使cmd指针在14行和15行分别被释放一次。下面是我得到的回溯信息：

```
Invalid free() / delete / delete[] / realloc()
   at 0x4A079AE: free (vg_replace_malloc.c:427)
   by 0x40084B: get_strings (sadstrings.c:15)
   by 0x40086B: main (sadstrings.c:19)
Address 0x4c3b090 is 0 bytes inside a block of size 19 free'd
   at 0x4A079AE: free (vg_replace_malloc.c:427)
   by 0x40083F: get_strings (sadstrings.c:14)
   by 0x40086B: main (sadstrings.c:19)
```

两个回溯信息的帧的顶部是标准库中释放指针的代码，但是我们可以很自信地认为标准库是被很好地调试过的。重点考察在堆栈中我写的代码部分，回溯信息指引我到sadstring.c的14行和15行，我修改过的代码中确实曾两次调用free（cmd）。

提示

Valgrind对检测那些根据没有初始化的值而进行条件转移的语句也很在行。用它来精确追踪何时一个变量初始化了，或者没有被初始化：

```
if(suspect_var) printf(" ");
```

在你的代码中插入下面的行，看看Valgrind是否在这一行开始抱怨了。

你也可以在你第一个出错的地方开始运行你的调试器，通过运行：

```
valgrind --db-attach=yes your_program
```

随着这种形式的启动，你在为每个探测到的错误运行调试器的时候，就有一个明确的行来研究了，然后你可以像平常那样检查隐含变量的值。此时，我们得到了程序第一次发生错误就失败的这种比较理想的情况了。

Valgrind也检测内存泄露：

```
valgrind --leak-check=full your_program
```

这个过程一般会比较慢，所以你也许不想每次都运行它。当它结束的时候，你会得到每个泄露的指针位置的回溯信息。

对于某些代码来说，追踪内存的泄露可能是非常费时间的。一个库函数的内存泄露，或者在用户程序的循环里被用到了100万次，或者在一个运行了几个月的程序中才可能最终造成一个大的问题。但是你也可以发现很多运行得很可靠的程序（在我的机器上，如doxgen、git、tex、vi等），Valgrind报告丢失了几千字节的内存。对于这种情况，如果一个bug没有造成不正确的结果或者用户能察觉到的速度的变慢，那么它可能不是一个高优先级的、需要马上解决的bug。

2.3 单元测试

你当然会为你的代码写测试。你将为比较小的组件写单元测试，为了确认不同的组件之间可以协同工作而进行集成测试。你甚至会是这样

的一类人：先写单元测试，然后编译链接程序以通过这些测试。

现在你会遇到如何组织那些测试的问题，这也就是测试框架（Test Harness）该出场的地方了。一个测试框架是一个为每个测试配置环境、运行测试，以及报告是否得到期望结果的系统。像调试器一样，我估计你们中的一些人也会想知道谁是那个不用测试框架的人，而对于另一些人，他们从来没有考虑用过测试框架。

实际上有很多选择。很容易就能写一两个宏来调用测试函数，并将它的返回值和期望值进行比较，也不止一个作者把那个简单的基础变成另一种可以提供完整测试框架的实现。其中提到：“Microsoft的内部共享工具库包括40多种测试框架。”为了和本书的余下部分保持一致，我将只为你展示Glib提供的测试框架。因为这些测试框架都很相似，也因为我也不想钻到太多的细节中，否则我不得不把Glib手册都给你，我在这里的讲述的内容也适用于其他的测试框架。

测试框架有几个比典型的自制测试宏的好得多的特点：

- 你需要测试程序失败的情况。如果一个函数异常终止了，或者伴随着一个错误信息非正常地退出了，你需要一种方法去测试程序的确退出，并传回了你期望的错误信息。
- 每个测试应该彼此独立，所以你不需要担心测试3会影响测试4的结果。如果你想确定这两个过程不会过度影响对方，在分别运行它们之后，在集成测速中按照顺序运行它们。
- 在你能运行测试之前，你可能需要制作一些数据结构。建立测试环境的场景是非常费时间的，如果能用同样的测试场景来运行多个测试就再好不过了。

例2-3展示了“实现一个字典”中字典对象几个基本的单元测试，示范了以上提到的3个测试框架的特性。它演示了最后一个特性如何很大程度上影响了测试框架的使用：一个新的结构类型在程序运行开始的地方被定义出来，然后有一些函数设置或者破坏这个结构类型的实例，一旦有了这些，我们就可以使用建造环境来写出几个测试了。

这个字典是一个键-值对，所以大部分的测试就是用一个特定的键值去取值并确定它能运转正常。请注意，一个NULL的键是不能被接受的，所以我们需要检查词典程序在被送入这样一个键的时候是否停止。

例2-3　一个对字典的测试，参见11.1.1“实现一个字典”（dict_test.c）

```
#include <glib.h>
#include "dict.h"

typedef struct {                                          ❶
    dictionary *dd;
} dfixture;

void dict_setup(dfixture *df, gconstpointer test_data){   ❷
    df->dd = dictionary_new();
    dictionary_add(df->dd, "key1", "val1");
    dictionary_add(df->dd, "key2", NULL);
}

void dict_teardown(dfixture *df, gconstpointer test_data){
    dictionary_free(df->dd);
}

void check_keys(dictionary const *d){                     ❸
    char *got_it = dictionary_find(d, "xx");
    g_assert(got_it == dictionary_not_found);
    got_it = dictionary_find(d, "key1");
    g_assert_cmpstr(got_it, ==, "val1");
    got_it = dictionary_find(d, "key2");
    g_assert_cmpstr(got_it, ==, NULL);
}
```

```
void test_new(dfixture *df, gconstpointer ignored){
    check_keys(df->dd);
}

void test_copy(dfixture *df, gconstpointer ignored){
    dictionary *cp = dictionary_copy(df->dd);
    check_keys(cp);
    dictionary_free(cp);
}

void test_failure(){                                          ❹
    if (g_test_trap_fork(0, G_TEST_TRAP_SILENCE_STDOUT |
    G_TEST_TRAP_SILENCE_STDERR)){
        dictionary *dd = dictionary_new();
        dictionary_add(dd, NULL, "blank");
    }
    g_test_trap_assert_failed();
    g_test_trap_assert_stderr("NULL is not a valid key.\n");
}

int main(int argc, char **argv){
    g_test_init(&argc, &argv, NULL);
    g_test_add ("/set1/new test",dfixture, NULL,              ❺
                                dict_setup, test_new, dict_teardown);
    g_test_add ("/set1/copy test",dfixture, NULL,
                                dict_setup, test_copy, dict_teardown);
    g_test_add_func ("/set2/fail test", test_failure);        ❻
    return g_test_run();
}
```

❶ 在一组测试中使用的元素称为一个fixture。Glib要求每个fixture都是一个结构，所以我们在开始的时候构造一个一次性使用的结构传递给setup，然后再传递给teardown。

❷ 这里就是setup和teardown函数，建立测试使用的数据结构。

❸ 定义setup和teardown函数后，测试本身就是一系列对fixture结构的操作并判断运算结果是否符合预期。Glib测试框架提供了一些附加的判断宏，比如这里用到的字符串比较宏。

❹ Glib通过POSIX fork系统调用来测试失败（这就意味着它不能在没有POSIX子系统的Windows上运行）。fork系统调用产生一个新的程序来运行if中的语句，这些语句应该失败并去调用abort。本测试程序观察fork产生的运行版本并检查它是否按照预期那样失败，并将预期的信息写到了stderr。

❺ 测试通过类似于路径字符串形式组织成不同的组。NULL参数是一个指向测试使用的数据集合的指针，这个数据集合不是系统建立和析构的。请注意new测试组和copy测试组是怎么用同样的setup和teardown函数的。

❻ 如果在测试前并不需要setup和teardown，可以用这个简单的形式来运行测试。

2.3.1 把程序用作库

函数库和程序的唯一区别就是，程序中包含了一个main函数，作为开始执行的入口。

现在我有一个虽然不大，但是足以被用作创立一个独立的共享库的文件。它仍然需要测试，我可以通过一个预编译条件把测试代码放在同一个文件中。在下面的代码段落中，如果Test_operations被定义了（通过随后讨论的各种方法），那么这个代码段落就是一个运行测试的程序；如果Test_operations没有被定义（即通常的情况），那么代码段落就被编译成没有main函数的、可以被其他程序调用的库。

```
int operation_one(){
    ...
}
```

```
int operation_two(){
    ...
}

#ifdef Test_operations

    void optest(){
        ...
    }

    int main(int argc, char **argv){
        g_test_init(&argc, &argv, NULL);
        g_test_add_func ("/set/a test", test_failure);
    }
#endif
```

有几种不同的方式可以定义Test_operations变量。用通常的配置选项，比如在你的makefile中，可以加上：

```
CFLAGS=-DTest_operations
```

-D选项是POSIX标准的编译器选项，相当于在每个.c 文件中加入#define Test_operations。

当你看到第 3 章中的Automake，你将看到其提供了一个+=运算符，因此在AM_CFLAGS设定通用选项后，你还可以用check_CFLAGS加上-D选项：

```
check_CFLAGS = $(AM_CFLAGS)
check_CFLAGS += -DTest_operations
```

依靠预编译条件包含main函数从某种角度来说也是很方便的。例如，我经常会基于一些怪异的数据来做分析。在写下最终的分析之前，我首先不得不写一个函数来读取和净化这些数据，然后写几个函数用来产生摘要统计，以检查完整性和进度。这些函数放到modelone.c中。也

许下一周，我可能有一个描述模型的新的想法，它将很自然地充分应用现有的函数来清理数据和展示基础的统计。通过有条件地把main包含在modelone.c中，我可以把原来的程序迅速地变成一个库。下面是modelone.c的框架：

```
void read_data(){
    [database work here]
}

#ifndef MODELONE_LIB
int main(){
    read_data();
    ...
}
#endif
```

这里我用#ifndef而不是#ifdef，因为一般是把modelone.c用作一个程序，而且这个选择跳转所做的，与为了测试目的而有条件包含main是类似的。

2.3.2 测试覆盖

什么是你的测试覆盖？你所写的测试是否执行到你所写的所有代码行？gcc有一个伴随的工具——gcov，用来计算每行代码被程序执行的次数。这个过程是这样的：

- 在你的gcc的CFLAGS中加入-fprofile-arcs-ftest-coverge。你也许需要采用-O0选项，这样就没有哪一行被优化掉了。
- 当程序运行的时候，每个源代码文件yourcode.c会产生一个或两个数据文件，yourcode.gcda和yourcode.gcno。
- 运行gcov　yourcode.gcda将向stdout输出你的程序中可执行代码的

被执行次数的百分比（声明、#include行等不被计算在内），并产生yourcode.c.cov文件。

- yourcode.c.cov的第一栏将展示在你的测试中每个可执行行的被执行的频率，并且将没有被执行的行用一堆#####来标识。那些是你应该考虑再写一个测试的部分。

例2-4展示了一个把所有步骤连接起来的shell脚本。我用here文档来产生makefile，这样我就可以把所有的步骤放在一个脚本中，并且在编译、运行和用gcov检查这个程序后，我用grep来找出被#####标识的行。GNU grep的-C3选项要求展示匹配行前后的3行。这不是POSIX标准的，但是，pkg-config和别的测试覆盖选项也都不是。

例2-4　一个编译脚本，用于代码覆盖测试、运行测试和检查未被执行的程序行的测试代码（gcov.sh）

```
cat > makefile << '------'
P=dict_test
objects= keyval.o dict.o
CFLAGS = 'pkg-config --cflags glib-2.0' -g -Wall -std=gnu99 \
          -O0 -fprofile-arcs -ftest-coverage
LDLIBS = 'pkg-config --libs glib-2.0'
CC=gcc
$(P):$(objects)
------
make
./dict_test
for i in *gcda; do gcov $i; done;
grep -C3 '#####' *.c.gcov
```

2.4　错误检查

一个完整的编程教科书必须至少给读者提供一节，用来讲解处理由

你调用的函数所返回的错误信息的重要性。

好吧，现在换个角度。让我们考虑你写的函数将怎样以及何时返回错误。在各种不同的上下文里，有很多不同类型的错误，所以我们必须把这个问题分成几种情况来讨论：

- 出现这个错误信息的时候，用户将怎么做？
- 接受这个错误信息的是一个人还是另一个函数？
- 这个错误用什么方式和用户沟通？

我将把第三个问题留到以后讨论（10.10“从函数中返回多个数据项”），但是头两个问题已经给我们足够的话题来讨论了。

2.4.1 在错误中的用户的角色？

如果缺乏对错误处理的仔细考虑，程序员们就会在他们的代码中像撒胡椒面一样间杂很多错误检查的代码，他们认为错误检查永远不嫌多，但是这并不是一个正确的方法。你需要像维护其他代码那样维护你的错误处理代码，并且你的函数的每个用户都已经被“每个错误都应该被处理”的无尽的说教洗脑了，所以如果你抛出了一个没有合理解决方案的错误，这个函数的用户将感到内疚和不确定。这就是所谓的“信息过剩”问题。

为了解决如何处理一个错误的问题，我们先考虑一个现代的问题：当一个错误开始的时候，用户会怎么参与其中？。

- 有时用户在调用这个函数前是不知道一个输入是否有效的。
  一个经典的例子就是，在一个键-值链表中寻找一个键，结果发现

这个键并不在链表中。在这个例子里，你可以把这个函数想象为一个提供检索的函数，并且在列表中没有键的时候抛出错误，或者你可以把它想象为一个双重目的的函数，就是它或者搜索键，或者告诉调用者这个键是否存在。

或者给你一个高中代数的例子：二次方程需要计算sqrt（b*b-4*a*c），如果括号里的值是负的，平方根就不是一个实数。期望函数的用户先计算b*b-4*a*c来判断可行性是不现实的，所以下面的考虑就是合理的：要么使二次方程函数返回二次等式的根，要么报告其根是否为实数。

在这些常见的输入检查的例子中，坏的输入并不是一个错误，而是函数的一个例行的和自然的使用。遇到错误就退出或者破坏性地停止，这样的错误处理方式（如同后续的介绍 ）不应该在这样的情形下被调用。

- 用户传递了一个显著的错误输入，比如一个NULL指针或者别的什么格式错误的数据。

  你的函数不得不检查这样的问题，以防止其引发段错误或者别的程序崩溃，你其实是很难想象调用者会怎么回应这个错误信息的。

  yourfn准备文档告诉用户指针不能为NULL，那么当他们忽略了这一点而调用int* indata=NULL; yourfn（indata），函数就会返回一个类似Error：Null pointer input的错误提示，但是很难预测调用者对这个错误提示会采取什么样的处理方式。

  一个函数开头通常有类似这样的几行：if（input1==NULL）return -1; …if（input20==NULL）return -1;。我发现在这种情形下，就像我在文档中那样一一列举调用者可能忽略的那些基本要求，这时仍

然发生一个“信息过剩”问题。

- 错误本身完全是内部过程引发的错误。
  包括“不应该发生”的错误，比如一个内部的计算不知怎地得到了一个不可能的答案——好比Hair（喜爱摇滚音乐的美洲部落）所称的“血肉模糊的错误”，比如没有反应的硬件，或者断掉的网络和数据库连接等。
  这个“血肉模糊的错误”一般可以被接收者处理（例如，晃动一下网线）。或者，如果用户要求1GB的数据被存储在内存里，但是内存不足1GB，那就最好报告一个“内存不足”错误。可是，当一个含有20个字符的字符串的分配失败的时候，这机器要么是已经超负荷运转而变得不稳定，要么就是被放在火上烤，这种情况下调用系统是很难利用你返回的错误信息而优雅地恢复的。根据你工作场景的不同，返回“机器着火了”这种信息可能并不准确，并且有“信息过剩”的嫌疑。
  内部过程错误（例如，与外界条件无关的错误，以及在某种程度上无效的输入值无关的错误）是不能被调用者处理的。在这种情况下，详细地报告给用户发生了什么错误可能是一个“信息过剩”问题。调用者需要知道此时的输出是不可靠的，而枚举出许多可能的错误条件仅仅为调用者（其有义务去处理所有的错误）带来更多的工作。

2.4.2 用户工作的上下文环境

像之前看到的那样，我们经常用一个函数来检查一系列输入的有效性，这样的用法本身并不是一个错误，如果这个函数为每种情况返回一

个有意义的值而不是简单调用一个错误处理函数，就会让这个函数更加有用。本节的余下部分考察一个真正的错误。

- 如果程序的用户有使用调试器的经验，而且工作在一个可以使用某种调试器的情况下，那么最快的失败方式就是调用abort来引发程序停止工作。那么这个用户在“犯罪现场”就有本地变量和回溯。abort函数从来就是符合C标准的（你需要用#inckude <stdlib.h>）。
- 如果程序的用户实际上是一个Java程序，或者完全不了解调试器是怎么回事，那么abort是非常讨厌的，此时正确的反应是返回某种错误代码来指示一个错误。

以上两种情形都有道理，所以最好是有一个if/then的跳转来让用户根据自己工作的上下文环境选择合适的回应模式。

我很久没有看到一个常见的库没有实现它自己的错误处理宏了。虽然C语言标准并没有提供，但是用C提供的功能实现起来很容易，所以就出现了每个人都写一个新的这种情况。标准的assert宏（提示：#include <assert.h>）会检查你做出的声明，并在你的声明出现false的时候停下来。每种实现会有一点不同，但是要点是：

```
#define assert(test) (test) ? 0 : abort();
```

assert（断言）自身的功能是测试你函数的中间步骤是否在做应该做的事情。我还喜欢把assert用作文档：它是一个针对计算机的测试，但是当我看到assert（matrix_a->size1==matrix_b->size2）的时候，作为一个人类读者，我想起这两个矩阵的维度必须符合指定的条件。然而，assert只提供第一种反应（非正常退出），所以这个断言过程应该被包

装一下。

例2-5展示了一个满足两种条件要求的宏；我将在10.2“可变参数宏”中讨论。另外也请注意，有的用户可以熟练使用stderr，而有的根本没使用过。

例2-5　一个处理错误的宏：报告或记录它们，并让用户来决定是遇到错误即停止，还是继续运行（stopif.h）

```
#include
#include  //abort

/** Set this to \c 's' to stop the program on an error.
    Otherwise, functions return a value on failure.*/
char error_mode;

/** To where should I write errors? If this is \c NULL, write to \c stderr
. */
FILE *error_log;

#define Stopif(assertion, error_action, ...) {                    \
        if (assertion){ \
            fprintf(error_log ? error_log : stderr, __VA_ARGS__); \
            fprintf(error_log ? error_log : stderr, "\n");        \
            if (error_mode=='s') abort();                         \
            else                 {error_action;}                  \
```

下面是一些假想的使用示例：

```
Stopif(!inval, return -1, "inval must not be NULL");
Stopif(isnan(calced_val), goto nanval, "Calced_val was NaN. Cleaning
up, leaving.");
...
nanval:
    free(scratch_space);
    return NAN;
```

处理错误的最常用的方式是简单地返回一个值，所以如果直接使用

这个宏，可能要非常频繁地键入return。不过，这可能也不错。一些作者经常抱怨那些精致的try-catch块配置实际上就是公认为有害的、沼泽般的goto的升级版。例如，Google内部的编码风格文档不建议使用try-catch结构，所用到的理由与不使用goto沼泽的理由完全一致。它提醒读者说，程序流可能在出错的时候被重新改变到其他位置（或者来自其他位置），因此我们应该保持错误的处理过程尽量简单。

2.4.3 如何返回错误信息

我将在后面的章节详细阐述这个问题（“从一个函数中返回多个项目”），因为如果你的函数有一定程度的复杂性，返回一个结构就更有意义，并且此时在这个结构中添加一个错误汇报变量也是简单而且合理的解决方案。例如，假设一个函数返回一个叫作out的结构，其包含一个叫作error的char*元素：

```
Stopif(!inval, out.error="inval must not be NULL"; return out
             , "inval must not be NULL");
```

Glib有一个它自己类型的错误处理系统——GError，其必须作为一个参数被（通过指针）传递进任意给定的函数。以增加复杂度为代价，这个系统提供了上面例2-5中所列出的宏中所没有的几项附加功能，包括错误域，以及从子函数更容易地向母函数传递错误信息。

2.5 编制文档

文档是必须的。我们知道这一点，也知道在每次代码改动的时候应该保持文档的更新。然而或多或少地，文档却经常是第一件被遗落的工

作。大部分程序员都会说“程序已经可以用了，我过后再补文档”！

所以你需要把文档书写过程也尽量地变得容易。这立刻让我们想到可以把文档和源代码放在一个文件里，尽可能地接近被说明的代码，不过这同时意味着你将需要一个从源文件中抽取文档的方法。

把文档放在代码旁边还意味着你更有可能读那些文档。在修改函数之前重温文档是一个好习惯，只有这样你才会明白这个函数到底是怎么回事，也才能使你注意到在改动代码的同时改动文档。

本节将介绍两种把文档编写进代码的方法：Doxygen和CWEB。通过包管理器安装它们很容易。

2.5.1 Doxygen

Doxygen是一个目的单纯的简单系统。它最擅长的工作就是为每个函数、结构或者其他代码段落添加描述。这也非常适合用来为使用接口的用户提供文档，因为用户一般也不在意代码本身。描述的内容作为注释行被放在一个函数、结构或其他被描述段落的前面。像这样先把文档作为注释写出，然后再把你所承诺的内容用代码实现出来。

Doxygen的语法非常简单，稍微了解一下就可以很快上手。

- 如果代码注释由两个星开始，就像/**…*/，那么Doxygen将解析注释。而带一个星号的注释，就像/*…*/，则被忽略。
- 如果你希望Doxygen解析一个文件，你将需要把一个/** \file */放在文件的头部；参见相关例子。如果你忘记了这点，Doxygen不会为这个文件产生输出，也不会在出错的时候给你多少提示。

- 把注释放在函数、结构等的前面。
- 你的函数描述可以（也应该）包括\param段落描述输入参数，以及一个\return行列出期望的返回值。参见相关例子。
- 用\ref来指向其他文档元素（包括函数或页）的交叉参考。
- 你可以在我上面用反斜杠的地方改用@：@file、@mainpage等。这是对JavaDoc的一个模仿，而后者看起来是在模仿WEB。作为一个LaTeX的用户，我更倾向于使用反斜杠。

为了运行Doxygen，你将需要一个配置文件，并且需要进行很多配置。Doxygen有一个非常灵巧的方式来处理这个；运行：

```
doxygen -g
```

这个命令将为你准备一个配置文件。然后你可以打开文件按需编辑；当然这个文件是有非常详尽的文档的。做完这些，运行doxygen自身来产生输出，按照你的选择，可以输出包括HTML、PDF、XML或者手册页等形式。

如果你安装了Graphviz（可以向你的包管理器查询），Doxygen还能产生调用图（call graphs）：方框—箭头模式调用图展示出函数调用和被调用的关系。如果有人向你提交了一个复杂的程序，并且期望你很快就能了解它，这种方式可以帮助你尽快地了解整体流程。

我用Doxygen注解了13.5“libxml和cURL”的程序；请浏览一下以了解它是怎么做的。你可以运行Doxygen来处理它，并看看它产生的HTML文档。

本书中所有以/**开头的代码段都遵循Doxygen格式。

具体说明

你的文档应该包含至少两部分：技术文档逐一描述了函数的细节，而且还有一个说明向用户解释这个包是怎么回事，以及如何找到它们。

说明的开始是一个用\mainpage开头的注释行。如果你产生了HTML输出，会在你的网站里有一个index.html——读者应该开始阅读的第一页。从这一页开始，添加任何你想添加的页。随后的页有一个如下形式的文件头：

```
/** \page onewordtag The title of your page
*/
```

回到主页（或者是任何页，包含函数的文档），添加\ref *onewordtag*以产生一个通往你写的页的链接。如果你需要这样，你也可以标识和命名主页。

说明页可以放在你的代码的任何地方：你应该把它们放在代码的附近，或者说明页也可以成为一个作为Doxygen注释行一部分的单独的文件，并采取类似documentation.h的名字。

2.5.2 用CWEB解释代码

TeX，一个文档格式系统，经常被当作复杂而有效的系统的典范。它已经有约35年历史，并且（按照作者的观点）仍然是现存排版系统的效果中最具吸引力的。很多最近的系统甚至根本都不敢去尝试与其对比，而是用TeX作为后台的排版引擎。它的作者，Donald Knuth，曾经为发现错误提供奖金，但是在他宣布这件事很多年后也无人领奖，所以它取消了奖金。

Knuth博士在*literate programming* 一书中解释了如何保证TeX的高质量，那就是每一个代码段落之前都用自然语言解释一遍这个段落的目的和功能。最终的产物看起来像一个自由格式的对代码的描述中不时掺杂着一点实际的代码以正式地描述计算机的行为（对比常见的带文档的代码，那些经常是代码比解释多）。Knuth用WEB来写的TeX，一个以英语为主题点缀PASCAL代码的系统。今天，代码会是用C写成的，并且现在TeX可以产生非常漂亮的文档，我们也可以用它作为说明文的标记语言。也就是CWEB。

对于输出，很容易就能找到关于用CWEB来组织甚至展现内容的教科书（例如，[Hanson，1996]）。如果有什么人想研究你的代码（可能是工作伙伴或者一个审阅团队），那么CWEB会有所帮助。

我在11.5.2“一个基于代理的组构造模型”采用了CWEB；这里是一个引导你编译它并遵循CWEB特点的工作步骤：

- 习惯上在保存CWEB文件时使用.w后缀。
- 运行cweave groups.w以产生一个.tex文件；然后运行pdftex groups.tex来产生一个PDF文件。
- 运行ctangle groups.w以产生一个.c文件。GNU make在它的内置规则中知道这一点，然后make groups将为你运行ctangle。

tangle这一步移走了注释，这意味着CWEB和Doxygen是不兼容的。也许你可以产生一个带有为使用doxygen处理的每个公共函数和结构的头文件，然后用CWEB处理你的主代码。

这里是将CWEB手册简化的七个要点。

- CWEB的特殊码都是字符@再加一个字母。要小心，在写@<titles@>的时候不要写成@<incorrect titles>@。
- 每个段落都是先有一个注释，然后是代码。注释可以是空白的，但是这个注释-代码的格式必须保持，否则会产生错误。
- 以一个@加一个空格来开始一个文本段落。然后用TeX格式开始说明的内容。

    以一个@c来开始一个未命名的代码段落。

- 以一个题目加上一个等号来开始一个命名代码段落（因为这是一个定义）：@<an operation@>=。
- 这个段落被逐字插入你用到题目的地方。就是说，每个名字的块实际上是一个展开成你指定的代码块的宏，但是并没有C预处理器宏的其他规则。
- 小节（比如示例中的关于组员资格、初始化、用Gnuplot来绘图等章节）是以@* 开始，并有一个以句号（.）结尾的标题。

以上应该足够你着手用CWEB来写自己的东西了。参考一下11.5.2“一个基于代理的组构造模型”，看看你能读懂多少。

成为一个更好的打字员

这本书的很多主题来源于我帮助我的同事写C程序的经验，在这个过程中我也掌握了是什么东西给他们造成了麻烦。例如，设置环境是一个拦路石；很多人不喜欢指针，更有甚者很多人不喜欢键盘。这可能和编程没什么关系，但是人们对自己的键盘没有足够的信心，我们的键盘用于写这种包含很多符号的文本如（i=0;i<10;i++）时，是否很合适呢？

如果下次你有一些与键盘相关的工作要做，可以找一件薄的T-Shirt并把它悬垂在键盘上。把你的手插到T-shirt下面，开始打字。

> 这么做的目的是防止在我们找不到键的时候经常去偷看。最后你会发现键盘会变得不那么漂浮，并且你离开的时候，它们也没什么变化。那种寻找某物导致的片刻停顿也可以帮助我们以安全速度使用键盘，这样你就会保持你的自信心和灵活性。
>
> 如果最开始的时候看不见令你有点恼火，你应该坚持下去并且努力去了解那些你不怎么熟悉的键。当你对键盘更加自信后，你就可以把更多的精力放到写作本身了。

第3章　打包项目

读到这里，你已经学习了解决C代码编写核心问题的一些工具——例如调试工具及编写文档工具等。要是你渴望开始接触C代码本身，那么可以直接跳到第二部分。本章及下一章将讲述一些用于与他人协作和发布工作的重量级工具：打包工具和版本控制系统。顺便我们会说很多题外话，主要是关于你如何用这些工具写更好的代码，即使你是在独自工作。

我在介绍部分提到过，但是没有人去看介绍。这里值得重复一下：C社区具有很高的内部互通性。是的，如果你去办公室或者咖啡馆，会发现每个人几乎都用同样的工具集。但是离开了本地的区域，你会看到很大的不一致。个人来说，我经常收到一些人的邮件，这些人在一些我完全没有听说过的系统上使用我的代码。我想这很好，我尽力让我在本地运行良好的代码具有互通性，这种互通性让我很有成就感。

目前，Autotools是一个为给定系统自动产生完美makefile的系统，它现在也开始聚焦如何发布代码了。在1.6“以源文件利用库”中你遇到过它了，当时我们用它来快速安装GNU科学计算库。即便你从来没有直接使用过它，包管理系统的维护人员可能就是使用这个工具正确地建造（build）你现在的计算机的。

如果不充分理解makefile是怎么工作的，你就会觉得难以理解Autotools在做什么，所以我们需要首先讲述一点makefile细节。不过一

言以蔽之，makefile其实就是由shell命令组成的。所以你需要首先认识shell提供的用于自动化的各种工具。学习的过程是非常漫长的，不过最终你将可以：

- 用shell来自动化工作。
- 用makefile来组织所有的那些shell中自动化的任务。
- 用Autotools使用户在任何系统上自动产生makefile。

3.1 shell

一个POSIX标准的shell需要满足以下条件。

- 丰富的宏能力，利用它你可以把你的文本用新的文本替代——也就是说，必须提供扩展语法。
- 一个具有图灵完备性的编程语言。
- 一个交互前端（命令提示环境），这个环境还需要包含很多用户友好的技巧。
- 一个用来记录和重用你所输入的所有内容的系统：history。
- 其他一些工具（本节暂时不会介绍），比如任务控制和很多内置的工具。

shell脚本语言的语法有很多内容，所以本节仅讲述这些门类中常用的一些语法点。有很多shell可以学习（后面有个边栏说明将建议你尝试一个与你现在用的不同的shell），但是除非有特别说明，本节所讲述的都是指POSIX标准的shell。

我不想花太多时间在交互特性上，但是我必须讲述一个非POSIX标

准的用法：tab自动完成。在bash中，假设你输入一个文件名并按Tab键，如果只有这一个文件备选项的话，这个名字将会被自动补齐完成；如果有多个选项，再次敲击Tab键可以看到所有选项的列表。如果你想知道你可以在命令行中键入多少种命令，在一个空行上敲击Tab键2次，bash就会给你完整的列表。其他的shell比bash提供的功能更多：在Z shell中键入make<tab>，它会为你在makefile中寻找可能的目标。而Friendly Interactive shell（fish）会检查手册页的简介行，所以当你输入man l<tab>的时候，它给你所有以L开头的命令的一行总结，从而可以节省你翻找手册页的时间和精力。

世界上存在两种shell用户：一种根本不知道tab自动完成存在，另一种则随时准备在每一行中都使用它。如果你还是属于前一种用户，那么你很快就会变成第二种用户了。

3.1.1 用shell命令的输出来替换命令

shell的行为特别像宏语言，其中一块的文本被其他几块的文本所替换。这在shell的世界里被称为扩展，并有很多种类型：本节我们会接触到变量替换、命令替换以及一些历史替换，并会提供一些例子，以讲述tilde扩展和用在快速的桌面计算器的算数替换。对于别名扩展、括号扩展、参数扩展、词语拆分、路径名扩展和块扩展等详细内容，你们自己去阅读shell手册。

变量就是简单的扩展。如果你在命令行上设定一个如下的变量：

```
onething="another thing"
```

然后输入：

```
echo $onething
```

那么，“another thing”将被输出在屏幕上。

你的shell可能会要求在“=”的两边不能有空格，这一点可能会多少让你讨厌。

当一个程序启动另外一个新的程序的时候（在POSIX C中，也就是当使用fork()系统调用的时候），一个全套环境变量的备份将会被送入子程序。当然，你的shell就是这么工作的：当你输入一个命令的时候，shell会发起一个新进程并把所有的环境变量发送给这个子进程。

所谓环境变量，实际上就是shell变量的一个子集。当你像刚才那样做了一个赋值操作，你也就为shell设定了一个变量来使用；当你采用：

```
export onething="another thing"
```

的时候，这个变量也就可以在shell中使用了，并且它的export属性也被设定了。一旦这个export属性被设定，你还是可以改变这个变量的值。

下一个扩展是关于反引号（`）的，它与单引号（'）的区别是看起来不那么垂直（当然，实际差别并不只这些）。

提示

垂直的引号（'，不是反引号）表示你不想扩展。所以下面的命令序列：

```
onething="another thing"

echo "$onething"

echo '$onething'
```

将打印出如下的结果：

```
another thing

$onething
```

反引号会将其中的命令用它自己的输出来替代，这非常类似于宏处理，即命令文本被就地以输出文本所替代。

例3-1是一个来计算C代码行数的脚本，如果结尾是“；”“）”或者“}”，我们认为这是一行。虽然代码行数在大多数的情况下都是一个很糟糕的指标，它仍然不失为一种不错的手段，并且在shell脚本中可以用一行实现。

例3-1　利用shell变量和POSIX工具计算行数（linecount.sh）

```
 # Count lines with a ;, ), or }, and let that count be named Lines.
Lines='grep '[;]]]' *.c | wc -l'

 # Now count how many lines there are in a directory listing; name it File
s.
Files='ls *.c |wc -l'

echo files=$Files and lines=$Lines
```

```
 # Arithmetic expansion is a double-paren.
 # In bash, the remainder is truncated; more on this later.
echo lines/file = $(($Lines/$Files))

 # Or, use those variables in a here script.
 # By setting scale=3, answers are printed to 3 decimal places.
 # (Or use bc -l (ell), which sets scale=20)
bc << ---
scale=3
$Lines/$Files
---
```

你可以使用. linecount.sh来运行这个脚本。点代表的是POSIX标准命令来运行这个脚本，你的Shell也允许你使用非POSIX标准而且有点复杂的方法：source linecount.sh。

提示

在命令行中，反引号大体上相当于$()。例如，echo `date`和echo $（date）基本相同。然而，make对$()有自己特别的用途，所以当你写makefiles时就需要反引号。

3.1.2 用shell的循环来处理一组文件

本节我们来做一点具体的编程，使用if语句和for循环。

首先这里有几条关于shell脚本的使用警告和须知。

- 作用域会带来一些困难——几乎所有的东西都是全局可见的。

- 它实际上是一种宏语言，因此所有那些当你写C预处理器代码的时候的注意事项，大部分也适用于你写shell脚本（参见8.1“营造健壮和繁盛的宏”）。
- 虽然现代的shell也可以提供一些工具，可以用来追踪错误或者以详报的方式运行脚本。但是你要知道对脚本语言而言，没有什么调试器。即使你只想使用 2.1“使用调试器”中介绍的那些基本功能也不行。
- 你将不得不熟悉一些容易迷惑人的小技巧，比如你不能在onething=another中的“=”两边使用空格，但是你必须在if [-e ff]中的“[”和“]”前后使用空格（因为它们是关键词，只是碰巧它们没有使用任何字母）。

有些人根本不把那些麻烦当回事儿，并且深爱 shell。就我而言，我会把那些需要在命令行重复执行的命令写成shell脚本来自动执行。一旦事情复杂到出现调用其他函数的函数，我就花点时间转移到Perl、Python、awk或者其他更合适的工具。

有一种编程语言，可以用来直接敲在命令行上以对多个文件执行相同的命令。让我们用传统的方式备份每一个.c文件，把每一个.c文件复制进一个以.bkup为后缀的新文件中：

```
for file in *.c;
do
 cp $file ${file}.bkup;
done
```

注意其中的第一个分号的位置：在文件列表的尾部将要开始循环的地方，在for语句的同一行。之所以特别指出，是因为每当我临时起意要

用这种单行的表述时，总是会忘记正确的顺序是;do而不是do。

```
for file in *.c; do cp $file ${file}.bkup; done
```

for循环对于处理一个程序的一系列*N*个操作是非常有用的。作为一个简单的例子，benford.sh搜索我们的C代码，寻找以某个十进制数字开头的数（例如行的开头或者以非数字开头紧随数字的情况），并把每个具有指定数字的行写入一个文件，参见例3-2。

例3-2　对于每个十进制数*i*，在文本文件中寻找（非数字的）序列；并计数（benford.sh）

```
for i in 0 1 2 3 4 5 6 7 8 9; do grep -E '(^|[^0-9.])'$i *.c > lines_with_
${i}; done
wc -l lines_with*   #A rough histogram of your digit usage.
```

作为练习，读者可以用它来测试一下是否符合Benford定律。

${i}中的花括号用来区分变量名及随后的文本；你现在还不需要，但是如果你想要一个类似${i}lines的文件名你就需要了。

有些读者的机器上可能安装有seq命令——它是BSD/GNU命令，不属于POSIX标准。那么我们可以使用反引号来产生序列：

```
for i in ‘seq 0 9‘; do grep -E '(^|[^0-9.])'$i *.c > lines_with_${i}; done
```

连续一千次执行上述程序现在也是轻而易举的事情了，例如：

```
for i in 'seq 1 1000'; do ./run_program > ${i}.out; done

#or append all output to a single file:
for i in 'seq 1 1000'; do
  echo output for run $i: >> run_outputs
```

```
  ./run_program >> run_outputs
done
```

3.1.3 针对文件的测试

现在假设你的程序依赖于一组数据，该数据需要从某个文本文件中读取并输出到某个数据库中。你只想做一次读入操作，如果以伪代码表达就是：if（数据库已经存在）then（什么也不做），else（从文本文件中产生数据库）。

在命令行中，你应该用test，一个经常在shell中内置的多才多艺的命令。为了试用，运行一下ls，得到一个你确认存在的文件名，然后用test来检查该文件是否存在：

```
test -e a_file_i_know
echo $?
```

test自己什么都不输出，但是作为一个C程序员，你知道每个程序都有一个main函数，且该函数返回一个整数，而我们这里就使用这个返回值。习惯上是读取返回值作为问题编号，所以0=文件存在，而在这种情况下1=文件不存在（这就是为什么main的默认返回值为0，参见7.1“不需要明确地从main函数返回”）。shell并不把返回值打印在屏幕上，而是把它存在一个叫作$?的变量中，并且你可以用echo来输出到屏幕。

警告

echo命令本身也有一个返回值，因此在你运行echo $?之后，

> $?的值又被设定为echo命令的返回值了。如果你想为了某种目的而多次使用$?，需要把它赋值给另一个变量，比如returnval=$?。

现在让我们在例3-3中的一个if语句中使用test，以便在一个文件不存在的时候特别处理。在C中，!意味着“否”。

例3-3　一个为test建立的if/then语句——反复运行它几次（.iftest.sh;.iftest.sh;. iftest.sh）观察test文件的产生和消亡（iftest.sh）。

```
if test ! -e a_test_file; then
    echo test file had not existed
    touch a_test_file
else
    echo test file existed
    rm a_test_file
fi
```

注意，跟之前的for循环一样，分号处于一个很奇怪的位置，并且这里有一个非常不错的规则，就是用fi来结束if语句块。相应地，else if是一个无效的语法；应该用elif关键词。

为了使你更加容易地反复运行它，让我们把它写在一个长的行中。关键词“[”和“]”等同于test，所以当你在别人的脚本中看到这个并想知道会发生什么的时候，答案在man test中。

```
if [ ! -e a_test_file ]; then echo test file had not existed; ↩
   touch a_test_file; else echo test file existed; rm a_test_file; fi
```

因为太多的程序遵从“0=OK”而“非0=有问题”的约定，我们能用没有test的if语句来表达下面的分句：if程序运行正常，then…。举个例子，用tar将一个目录打包成单个.tgz文件，然后删除该目录是个很常见

的操作。但如果tar文件因为某种原因没有生成而源目录已经删除，那真是一场灾难。所以，我们在删除所有文件之前要测试一下tar命令是否完全成功。

```
#generate a test file:
mkdir a_test_dir
echo testing ... testing > a_test_dir/tt

#Compress it, and remove only if the compression succeeded.
if tar cz a_test_dir > archived.tgz; then
    echo Compression went OK. Removing directory.
    rm -r a_test_dir
else
    echo Compression failed. Doing nothing.
fi
```

如果你想看到程序执行失败这种情况，先输入chmod 000 archived.tgz命令使要打包的文件是不可写的，然后再运行一次本脚本程序即可。

需要知道上面的方式是关于程序的返回值，程序可以是test或者是其他一些程序。现在你想用真实的输出，让我们回头使用反引号。例如，cat youfile | wc-l会产生一个数字，代表youfile文件中有多少行（假设youfile这个文件已经建立并存在），那么把这些嵌入test是合适的。

```
if [ ‘cat yourfile | wc -l‘ -eq 0 ] ; then echo empty file.; fi
```

尝试一个复用器

在编码的时候我习惯开两个终端：一个是编辑器中的代码，另一个则用来编译和运行程序（有可能在一个调试器中）。在终端之间灵活的切换突然变得难以置信的重要。

有两个终端复用器可以选择，各来自竞争的双方：GNU和BSD，即GNU Screen和tmux。可能你的包管理器已经安装了它们中的一个或全部。

两个都是用一个单一命令键工作的。GNU Screen缺省使用Ctrl-A。Tmux默认使用Ctrl-B，但是共识是，看起来每个人都重新映射到Ctrl-A上了，即通过添加：

```
unbind C-b

set -g prefix C-a

bind a send-prefix
```

到主目录的.tmux_conf。手册里也列出了一些你可以在配置文件里添加的其他的事项。当你寻找提示和文档的时候，顺便提一句，请注意你应该在Internet搜索引擎里键入GNU Screen，因为仅仅查询Screen不会得到关于GNU Screen的任何有用的信息。

当把Ctrl-A设定为命令键，Ctrl-A Ctrl-A将在窗口间切换，你也可以从手册里读到Ctrl-A（其他键）的组合让你可以在窗口列表里向前或向后移动一个，或者显示所有的窗口列表以便你可以从列表中挑选切换的窗口。

这两种复用器都可以解决多窗口问题。但是它们能做的还远不止这些：

- Ctrl-A-D将分离对话，就是说你的终端将不再显示在这个复用器的控制下的不同的虚拟终端。但是它们仍然在后台运行。

——在用了GNU Screen/Tmux一整天之后，分离。过后，从家里或者第二天在办公室重新绑定，用screen-r或者tmux attach，并从你离开的地方捡起来。这种分离后继续运行的能力，使得你哪怕是在位于伯利兹或者乌克兰的某台配置凌乱的服务器上工作，也可以得到不错的体验。

——更神奇的是，在你分离之后，多路复用器把程序留给它自己的虚拟终端运行，这对那些要运行整夜的长过程是非常有用的。

- 具有剪切/粘贴功能。

——现在我们还没有用鼠标：一旦在复制模式，你可以在你的终端里翻页，高亮一个段落，然后把一段复制到多路复用器的内部剪贴板，再粘贴到命令行上。

——当你浏览需要剪切的内容时，你可以上滚历史内容并寻找特定的字符串。

的确，那些终端复用器最终把终端从用来工作的地方变成了用来快乐工作的地方。

3.1.4 fc

fc是一个（POSIX标准）命令，用来把你在shell中输入的那些命令变成一个可重复的脚本。输入：

```
fc -l # The l is for list and is important.
```

你就将在屏幕上得到一个最近所输入的命令的带编号列表。输入history也可能得到同样的效果。

-n选项可用于限制行数，这样你就可以把第100到200行历史写入一个文件：

```
fc -l -n 100 200 > a_script
```

然后删除用不到的命令行，从而把命令行上的尝试变成一个干净的shell脚本。

如果忽略了-l选项，那个fc就是一个非常快速和灵活的工具。它可以立刻启动一个编辑器（就是说如果你用>重定向，你基本上就停在那里），不显示行号，而当你退出编辑器，文件中的内容就立刻继续执行。这对快速重复最后几行是非常棒的，但是如果你不够小心，后果就是灾难性的。当你意识到自己忘记了-l，或者惊讶地发现自己在编辑器里，请删除屏幕的一切以防止执行一些不期望的命令。

fc代表fix command，这也是它最简单的用法。如果没有任何设定，它仅编辑前一行，因此当你需要对某个命令进行较为复杂的更正而不是仅代替一个输入错误时非常有用。

尝试一个新的**shell**

除了操作系统默认提供的之外，shell还存在很多种类。这里我列举几个Z shell的有趣功能，以便展示给你换用bash之外的shell可能带来的体验。

Z shell的功能和变量有几十页，简约是潮流——但是既然有交互的便利，为啥要费心去做（朴素的）斯巴达人呢？（如果你有斯巴达美学，又想使用bash，那么建议尝试ash。）Z shell在～/.zshrc文件设定变量（或者你可以直接在命令行中输入并导出它们）；你将需要下面的例子：

```
setopt INTERACTIVE_COMMENTS

#now comments like this won't give an error
```

字块的展开，比如用file.c file.o file.h代替file.是*shell*的职责。*Zsh*在扩展这个时的最有用方式是，*/告诉shell在扩展的时候去遍历目录树。POSIX标准的shell把～当作主目录，所以如果你想得到你权限范围内的所有.c文件，你可以尝试：**ls ~/**/*.c.**

让我们把所有.c文件的目前状态都备份下来

```
# This line may create a lot of files all over your home directory.

for ff in ~/**/*.c; do cp $ff ${ff}.bkup; done
```

记得bash只对整数提供算数扩展，所以$（（3/2））总是1吗？Zsh和Ksh（以及其他的）是C风格的，并给你一个实数（不只是整数）的答案，如果你把分子和分母设为浮点数。

```
echo $((3/2.))        #works for zsh, syntax error for bash


#repeating the line-count example from earlier:

Files='ls *.c |wc -l'

Lines='grep '[]];]' *.c | wc -l'

echo lines/file = $(($Lines/($Files+0.0)))  #Add 0.0 to cast to float
```

文件名中的空格可能在bash中捣乱，因为空格键隔离了元素，Zsh有一个显式的数组语法，这样就不需要使用空格作为一个元素的分隔符了。

```
# Generate two files, one of which has spaces in the name.
echo t1 > "test_file_1"
echo t2 > "test file 2"

# This fails in bash, is OK in Zsh.
for f in test* ; do cat $f; done
```

如果你决定切换shell，你有2种方式来做：你可以用chsh命令在登录（login）系统中正式存储更换的shell（会修改/etc/passwd）；如果不能使用chsh，你可以在你的.bashrc的最后一行加上exec–l /usr/bin/zsh（或者别的你喜欢的shell），那么bash每次将用你喜欢的shell来代替掉自己。

如果你想让makefile使用一个非标准的shell，加入：

```
SHELL=command -v zsh
```

（或者任何你想要的shell）到你的makefile。POSIX标准的command -v打印你的命令的全路径名，所以你不必自己去查看。SHELL是一个奇怪的变量，它必须在makefile中设定或者用make命令行参数去设定， make会忽略掉名字为SHELL的环境变量。

3.2 makefile还是shell脚本

你可能有一群围绕一个项目飞舞着的小程序（词计数、拼写检查、运行测试、写入版本控制、把版本控制推送到远程服务器、备份），而且全部都可以用shell脚本来自动化。与其为每个小任务写一个一到两行的脚本，不如把它们都放在一个makefile中。

在1.4“使用makefile”中我们首次讲述了makefile，现在我们又介绍了很多关于shell的细节，因此有更多内容可以放在makefile中。这里有一个来自我的实际生活的例子，其中用到shell的if/then语句和test。在工作

中我用Git来管理我的代码，但是有三个子版本资料库需要处理，而我从来记不住这个过程。如例3-4所示，现在我有一个makefile来为我记住它。

例3-4　把一个if/then和一个测试放在一个makefile中（make_bit）

```
push:
    @if [ "x$(MSG)" = 'x' ] ; then \    ❶
            echo "Usage: MSG='your message here.' make push"; fi
    @test "x$(MSG)" != 'x'              ❷
    git commit -a -m "$(MSG)"
    git svn fetch
    git svn rebase
    git svn dcommit

pull:
    git svn fetch
    git svn rebase
```

❶ 在每次进行提交时都需要指定说明信息，所以我设置了一个环境变量来帮助即时提醒：MSG="This is a commit. " make。这一行是一条if-then语句，如果我忘记了，它会打印一个提示。

❷ 测试"x$（MSG）"是否能展开成"x"以外的值，也就是$（MSG）不是空串。这是一个一般常见的shell用来检查是否是空串的技术。如果这个测试失败了，make不会继续运行。

makefile中执行的这些命令就是你应该在命令行中输入的，但另一方面它们又有很大的不同：

- 在一个独立的shell中，每一行命令都独立运行。如果你在makefile中写下：

```
clean:
    cd junkdir
    rm -f *      # Do not put this in a makefile.
```

那么你可能就倒大霉了。在脚本中的那两行实际上和C代码中的这两行是对等的：

```
system("cd junkdir");
system("rm -f *");
```

由于system（"cmd"）是和sh -c "cmd"对等的，我们的make脚本也等同于：

```
sh -c "cd junkdir"
sh -c "rm -f *"
```

并且对于shell极客们，（cmd）在一个子shell中运行cmd，所以make代码段相当于下面的shell命令：

```
(cd junkdir)
(rm -f *)
```

在任何情况下，第二个子shell对第一个子shell发生了什么是一无所知的。make将首先产生一个shell并进入你打算清空的目录下，然后make在那个子shell中完成工作。然后它在你开始的目录下开始一个新的子shell并调用rm -f *。

好的方面是，make将为你把这个出错的makefile删掉。如果你想正确地表达你的想法，可以用：

```
cd junkdir && rm -f *
```

这里&&以短路序列的方式来运行命令（即如果第一个命令失败了，不用再麻烦去运行第二个）。或者用一个反斜杠将两行命令并作一行：

```
cd junkdir&& \
rm -f *
```

对于这种要命的情景，我不愿意简单相信一个反斜杠。所以在实际中，你最好使用rm -f junkdir/*。

- make会用适当的值来替换$x（用于单字母或者单符号的变量名）或$（xx）（用于多字母的变量名）。
- 如果你希望是shell而不是make来执行替换，那么就去掉括号，并在前面加两个$$。例如，为了在makefile中用shell的变量来做备份：for i in *.c; do cp $$i $${i%%.c}.bkup; done。
- 回忆一下1.4“使用makefile”中的技巧，在那里你可以在紧挨着一个命令的前面设定一个环境变量，例如，CFLAGS=-O3 gcc test.c。在每个命令拥有自己独立shell环境的时候，这个技巧就十分方便了。仅仅是不要忘记，赋值必须在一个命令之前发生，而不是一个类似if或while的shell关键词之前。
- 每行开头的@意思是，运行这个命令但是不把任何输出显示在屏幕上。
- 每行开头的-意思是，如果命令返回一个非零值，继续执行。否则，在第一个非零值的地方停止脚本。

对于比较简单的项目或者你日常的那些重复工作，一个使用那些来自shell的功能的makefile应该可以充分满足你的需求了。你了解你使用

计算机来完成的每个工作，而一个makefile可以叫你把它们在一个地方写下来，然后你就不需要再操心它们了。

你的makefile会对你的同事也有效吗？如果你的程序就是一系列常见的.c文件和一些已经安装的必要的库，并且你的makefile中的CFLAGS和LDLIBS对目标系统也适合，那么它可能会正常工作，或者最差也就是需要一两封邮件澄清。如果你正在产生一个共享库，那么忘记它吧——为Mac、Linux、Windows、Solaris或者它们的不同版本产生共享库的过程是非常不同的。当你广泛发布库的时候，所有的动作都应该尽量地自动化，因为不可能为几十上百个人同时用邮件来讨论配置问题，而且大多数人也不会愿意花费那么大的精力去处理陌生人的代码。基于所有这些原因，我们需要为公开发行包添加另一层机制。

3.3 用Autotools打包代码

你可以安装一个下载的库或者程序，正是Autotools使得这一切成为可能，只要运行下面的命令：

```
./configure
make
sudo make install
```

来安装它（不需要其他的动作）。请注意，这就是现代科学的一项奇迹：开发者对你用什么计算机、你在哪里保存你的库和程序（/usr/bin? /sw?还是/cygdrive /c/bin?）以及还有哪些不为人知的奇怪的设定都一无所知。但是congfiure可以搞清楚每样东西并使得make能够无缝运行。所以Autotools在现今已经成为所有程序发布的核心。如果你希望所有人平等地使用你的代码（或者如果你想把你的程序加入Linux发

行版的包管理器），那么你需要让Autotools来为你制作安装程序。

经常发现一个包要依赖于某些存在的框架才能安装，例如Scheme，要求版本号大于2.4并且小于3.0的Python，同时还需要Red Hat 包管理器等。框架使得用户很容易地安装包——只要它们正确地安装好框架。对于那些没有root权限的用户，这个要求有点太难了。Autotools很与众不同，因为它只要求你有一台POSIX兼容的计算机就可以了。

很快你就将意识到Autotools可以变得有多么复杂，但是其基础是十分简单的。本章后面，我们将写出六行的包管理文本和运行四个命令，就能够有一个可供发行的完整包（尽管很简单）。

现在说说Autoconf、Automake和Libtool的实际历史：这些是不同的包，并且都独立地开发过。这里只是我尽量想象的一种情况。

梅　　诺：我喜欢make。我可以把编译我项目的每一个小步骤写在一起，真不错！

苏格拉底：是的，自动化是伟大的。无论何时，万事万物都应该实现自动化。

梅　　诺：是啊。我开始在我的makefile中添加很多目标，这样用户输入make就可以编译生成程序，使用make install来安装，make check来运行测试等。写这些makefile目标的工作量很大，但是完成后它就会非常流畅。

苏格拉底：嗯。我会写一个系统——叫它Automake吧——让它从一个非常短的前置makefile中自动产生带有常用目标的makefile。

梅　　诺：真棒！产生共享库是特别烦人的事情，因为每个系统都有一个不同的流程。

苏格拉底：的确很烦人。有了系统信息，我将写一个程序，用于产生可以从源代码来制作共享库的脚本，然后把它们放在一个自动产生的makefile中。

梅　　诺：哇！所以我只需要告诉你我的操作系统，还有我的编译器叫作cc、clang还是gcc或者别的什么，然后你就会为那个我工作的系统产生正确的代码？

苏格拉底：那是很容易出错的。我将写一个叫作Autoconf的系统，它将检测所运行的系统并产生一个报告，包含Automake和你的程序所需知道的关于系统的一切。随后Autoconf将运行Automake，利用报告中的变量来产生一个makefile。

梅　　诺：我很惊讶——你已经自动化了产生makefile的过程。但是这听起来像我们利用探查到的不同平台来为Autoconf写配置文件，以及为Automake准备makefile模板，从而改变了我们必须做的工作。

苏格拉底：你是对的。我将写一个工具，Autoscan，它可以用来扫描你为Automake写的Makefile.am，然后为你自动生成Autoconf的configure.ac。

梅　　诺：那么现在只需要自动产生Makefile.am了。

苏格拉底：没错，自己看文档学习，然后自己做一个吧。

这个故事里的每一步都为前一步添加了一点自动化：Automake用一个简单的脚本来产生makefile（这种自动编译已经比手工键入编译命令方便很多了）；Autoconf测试环境并用这些信息来运行Automake；Autoscan检查程序代码找到运行Autoconf所需要的信息。Libtool在背后运行来帮助Automake。

3.3.1 一个Autotools的示例

例3-5展示了一个使用脚本来自动编译“Hello, World”的示例。它是以shell命令的形式实现的，你也可以将这些指令复制/粘贴到你的命令行（只要你确保反斜线的后面不要有空格）。当然，除非你使用包管理器安装了Autotools的系列工具（Autoconf、Automake，以及Libtool工具），否则脚本不会运行。

例3-5　打包Hello, World（auto.conf）

```
if [ -e autodemo ]; then rm -r autodemo; fi
mkdir -p autodemo                          ❶
cd autodemo
cat > hello.c <<\
"--------------"
#include <stdio.h>

int main(){ printf("Hi.\n"); }
--------------

cat > Makefile.am <<\                      ❷
"--------------"
bin_PROGRAMS=hello
hello_SOURCES=hello.c
--------------

autoscan                                   ❸
sed -e 's/FULL-PACKAGE-NAME/hello/' \      ❹
    -e 's/VERSION/1/' \
    -e 's|BUG-REPORT-ADDRESS|/dev/null|' \
```

```
    -e '10i\
AM_INIT_AUTOMAKE' \
       < configure.scan > configure.ac

touch NEWS README AUTHORS ChangeLog     ❺
autoreconf –iv                          ❻
./configure
make distcheck
```

❶ 创建一个目录autodemo，并且用Here文档（here document）的方式在该目录创建hello.c。

❷ 我们需要亲手准备一个有两行长的Makefile.am。其中的hello_SOURCES是可选的，因为Automake会猜出要从hello.c这个文件建立hello。

❸ 运行autoscan，产生configure.scan。

❹ 编辑configure.scan以给出你的项目的参数（名称、版本、联系邮件地址），且把AM_INIT_AUTOMAKE这一行加入以便初始化Automake。（是的，这很麻烦，特别是当Autoscan用Automake的Makefile.am来收集信息，这样我们就充分意识到要用Automake。）你可以亲手做这个；我习惯用sed直接把定制化的版本导入configure.ac。

❺ GNU编程标准需要这4个文件，否则Autotools不会处理。我有时会用POSIX标准的touch命令创建一个空的版本来作弊；你的则需要有实际的内容。

❻ 对于configure.ac，运行autoreconf以产生所有需要发行的文件（特别是，configure）。-i配置选项将产生系统需要的额外的样板文件。

这些宏将做多少事情？hello.c程序本身只有区区3行，而Makefile.am也只有2行，也就是说我们自己写的程序也就5行文本。你的结果可能有一点不同，但是当我在脚本运行后产生的目录中运行wc -l *，我发现了11000行文本，包括一个4700行的configure脚本。

它的可移植性造成了它的臃肿：你的目标机器可能还没有安装Autotools，而且谁也不知道别人的电脑上还会缺少什么，因此这个脚本只能简陋地依赖于POSIX兼容。

我在这个600行的makefile中找到了73个目标。

- 默认目标，也就是当你在命令行键入make的时候，会产生执行程序。
- sudo make install会按照使用者要求来安装这个程序；运行sudo make uninstall来删除它。
- 甚至有一个令人兴奋的选择用来make Makefile（如果你对Makefile.am做了这一点调整，或者希望快速重新产生makefile，这一点实际上很方便）。
- 作为这个包的作者，你将对make distcheck感兴趣。这将产生一个tar文件，内置一个用户需要解包并运行通常的./configure; make; sudo make install所需要的所有内容（不需要像开发用机器上安装的Autotools），并通过运行任何一个你指定的测试来验证这个发行版是正确的。

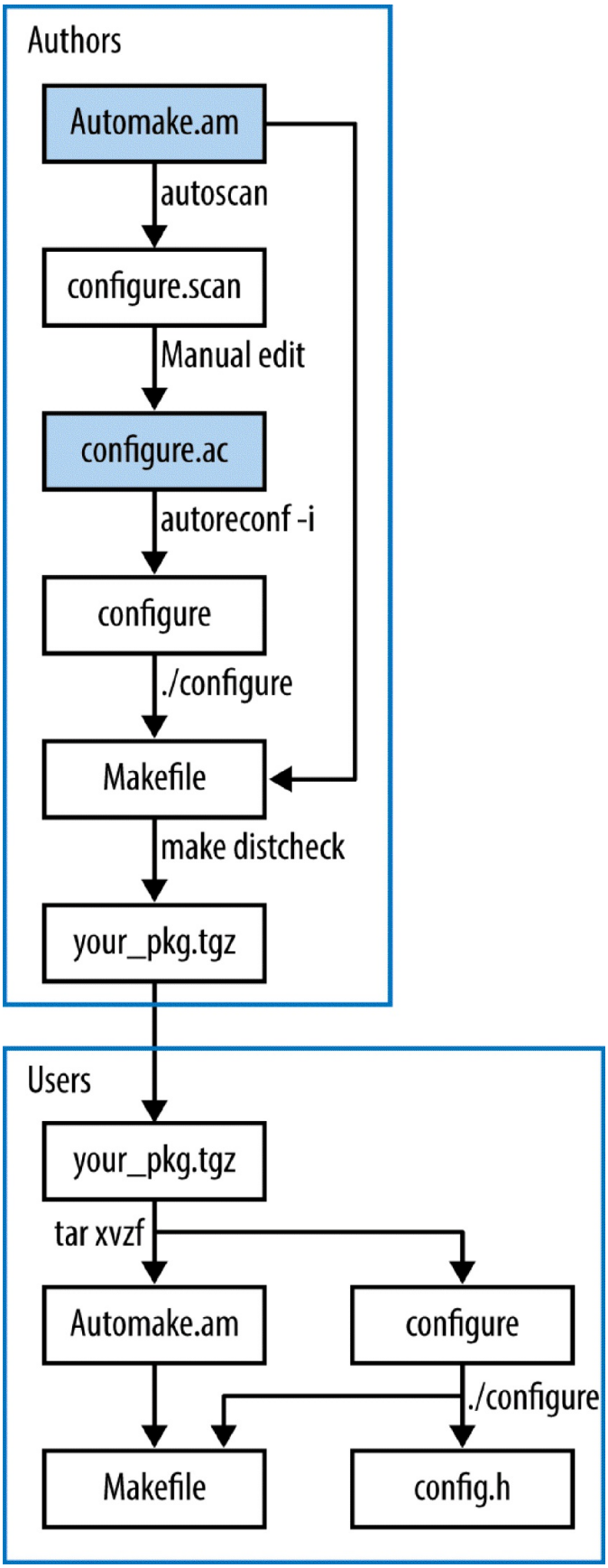

Authors
Automake.am
autoscan
configure.scan
Manual edit
configure.ac
autoreconf -i
configure
./configure
Makefile
make distcheck
your_pkg.tgz
Users
your_pkg.tgz
tar xvzf
Automake.am
configure
./configure
Makefile
config.h

图3-1　自动工具流程图。你只需要编写其中的2个文件（阴影部分）；而其他的都由命令自动产生

你有可能只写这些文件中的两个（带阴影的部分）；其他所有的一切都是被给定命令自动生成的。让我们从最底部开始：用户一般是以tar包的方式得到你的包，用tar xvzf your_pkg.tgz解包，这将用你的代码产生一个目录，内置你的代码，Makefile.am，configure，以及一些其他不值得在这里讨论的辅助工具的文件。用户键入./configure，这将产生configure.h和Makefile文件。现在所有事情都就位了，只需要键入make; sudo make install。

作为一个程序员，你的目的是用一套高质量的configure和Makefile.am来产生那个tar包，这样用户可以自己运行而不会遇到问题。尝试开始自己写Makefile.am吧。运行autoscan得到一个预发布的configure.scan，然后你将手动编辑configure.ac。（补充信息：GNU代码规范需要这4个文件：NEWS、README；AUTHORS，以及ChangeLog。）然后运行autoreconf–iv以产生configure脚本（外加很多辅助文件）。有了configure脚本，你现在可以运行它一次以生产makefile，有了makefile，你可以运行make distcheck以产生可以公开发行的tar包。

注意，这里有点重叠：你和用户使用同样的configure和Makefile，虽然你的目的是生成包而用户想的是安装包。那就意味着，如果你安装了这些工具，你就可以在没有完整打包的前提下测试代码，而如果有必要，用户也可以方便地重新打包代码。

3.3.2 用makefile.am来描述makefile

一个典型的makefile中，一半是关于你的项目中的什么部分依赖于其他的什么部分的结构，另一半是关于用于执行的特定变量和程序。你的Makefile.am将聚焦于什么需要被编译以及它们的相依性，而变量和程序的定义将被Autoconf和Automake内置的关于在不同平台编译的知识填充。

Makefile.am包含两种类型的项目，我将它们称为形式变量和内容变量。

1. 形式变量

一个需要被makefile处理的文件可能有很多种目标，每一种都被Automake用一个短字符串标注。

bin

可执行程序安装路径，例如，/usr/bin或者/usr/local/bin。

include

头文件安装路径，例如，/usr/local/include。

lib

库文件安装路径，例如，/usr/local/lib。

pkgbin

如果你的项目被命名为project，安装在主程序目录的一个子目录内，例如，/usr/local/bin/project（pkginclude或pkglib与之类似）。

check

当用户键入make check的时候用来测试程序。

noinst

不要安装，仅用于保存某文件以用于其他目标。

Automake产生make脚本的模板，并且准备了不同的模板：

```
PROGRAMS
HEADERS
LIBRARIES    static libraries
LTLIBRARIES  shared libraries generated via Libtool
DIST          items to be distributed with the package, such as data files
 that didn’t go elsewhere
```

一个目标加上一个模板就等于一个形式变量。例如：

```
bin_PROGRAMS     programs to build and install
check_PROGRAMS   programs to build for testing
include_HEADERS  headers to install in the system-wide include directory
lib_LTLIBRARIES  dynamic and shared libraries, via Libtool
noinst_LIBRARIES static library (no Libtool), to keep on hand for later
noinst_DIST      distribute with the package, but that's all
python_PYTHON    Python code, to byte-compile and install where Python pac
kages go
```

现在你都了解了这些形式，你能用这些来详细区分每个文件是怎么处理的。在那个之前的“Hello, World”的例子中，只有一个必须被处理的文件：

```
bin_PROGRAMS = hello
```

为了给你另外一个例子，noinst_DIST是我用来放那些需要在编译后测试中使用的，却不值得安装的数据的。你可以在每行放置任意多的条目。例如：

```
pkginclude_HEADERS = firstpart.h secondpart.h
noinst_DIST = sample1.csv sample2.csv \
              sample3.csv sample4.csv
```

2. 内容变量

放在noinst_DIST中的内容会被简单地复制到发行包中，而HEADERS被复制到目标目录中并被设置合适的权限。这样就基本准备好了。

对于编译步骤，例如…_PROGRAMS和…_LDLIBRARIES，Automake需要知道关于编译的更多的细节。至少，它需要知道什么源文件需要被编译。那么对于有关编译的形式变量等号右面的每个条目，我们都需要一个变量用来明确指定源文件。例如，在这两个程序中我们需要两个SOURCES行：

```
bin_PROGRAMS= weather wxpredict
weather_SOURCES= temp.c barometer.c
wxpredict_SOURCES=rng.c tarotdeck.c
```

一个基本的包也就需要这些了。

警告

这里我们又一次违反了原则，那就是做不同任务的事物应该看起来不同：内容变量有同样的lower_UPPER形式，看起来和之前的形式变量一样，但是它们是从完全不同的部分构建的，所服务的目的也完全不同。

回忆一下，之前介绍基本的makefile时，提到了关于几条make内置的一些预设规则，像CFLAGS之类的变量以处理一些内部细节。Automake的形式变量有效地定义了更多的默认规则，并且每条规则都有它们自己的相关的变量。

例如，用来连接目标文件以形成一个可执行文件的规则可能看起来是这样的：

```
$(CC) $(LDFLAGS) temp.o barometer.o $(LDADD) -o weather
```

警告

GNU Make在连接命令的后半段用LDLIBS作为库变量，而GNU Automake在连接命令的后半段中用LDADD。

用你喜欢的Internet搜索引擎，并不难查找一个形式变量如何被转化为最终的makefile里的若干目标，但是我发现想知道Automake做了什么的最快的方式就是运行它并查看输出的makefile长什么模样。

你可以在一个为每个程序或每个库准备的文件里设定所有这些变量，比如weather_CFLAGS=-O1。或者，用AM_VARIABLE来为所有的编译或者连接设定变量。这里是我喜欢的编译器选项，也是你在之前的1.4“使用Makefile”中遇到过的：

```
AM_CFLAGS=-g -Wall -O3
```

我没有包含-std=gnu99以使gcc回避掉一点标准障碍，因为这是一个编译器特有的选项。如果我在configure.ac中放入AC_PROG_CC_C99，那么Autoconf将为我把CC变量设定给gcc –std=gnu99。Autoscan还没有聪明到把这个放在它为你产生的configure.scan中的地步，所以你可能不得不自己把它放在configure.ac里。（在本文中，AC_PROG_CC_C11宏还不存在。）

特殊的规则可以重载基于AM_的规则，所以如果你想保持那些通用规则并为一个选项添加重载，你可以这么做：

```
AM_CFLAGS=-g -Wall -O3
hello_CFLAGS = $(AM_CFLAGS) -O0
```

3. 添加测试

我还没有为你展示字典库（将在11.1“扩展结构和字典”中充分讲述），但是我已经在“单元测试”中向你展示了如何利用测试框架（harness）去测试它。当Autotools产生库后，是有必要再运行一次测试的。步骤如下所示。

- 一个基于dict.c和keyval.c的库。它有头文件，dict.h和keyval.h，这些也都需要和库一起发行。

- 一个测试程序，也就是Automake需要知道这个程序是用来测试的，而不是用来安装。
- dict_use程序是这个库的使用者。

例3-6描述了这些步骤。这个库先被编译出来，这样它就可以用来产生使用它的程序和测试框架。TEST变量指定当用户键入make check的时候，具体被运行的程序或脚本。

例3-6　一个处理测试的Automake文件（dict.automake）

```
AM_CFLAGS=‘pkg-config --cflags glib-2.0‘ -g -O3 –Wall    ❶

lib_LTLIBRARIES=libdict.la                               ❷
libdict_la_SOURCES=dict.c keyval.c                       ❸

include_HEADERS=keyval.h dict.h

bin_PROGRAMS=dict_use
dict_use_SOURCES=dict_use.c
dict_use_LDADD=libdict.la                                ❹

TESTS=$(check_PROGRAMS)                                  ❺
check_PROGRAMS=dict_test
dict_test_LDADD=libdict.la
```

❶ 这里我做了一个弊，因为其他的用户可能没有安装pkg-config。如果我们不能假设有pkg-config，我能做的最好的方式就是用Autoconf的AC_CHECK_HEADER和AC_CHECK_LIB来检查库。如果有什么没有找到，请用户修改CFLAGS或LDFLAGS环境变量来指定正确的-I或-L选项。因为我们还没有讨论configure.ac，我现在仅使用pkg-config。

❷ 第一个任务是产生共享库（通过Libtool，所以LTLIBRARIES以LT开头。）。

❸ 当把文件名改写成内容变量时，把任何不是字母、数字或者@的内容都要改变为下划线，就像libdict.la改为libdict_la。

❹ 现在我们指定如何产生一个共享库，我们可以用这个共享库来组装程序和测试。

❺ TESTS变量指定了在用户键入make check的时候运行的测试程序。因为这些经常都是shell脚本而不需要编译。它是与check_PROGRAM不同的，因为后者指定的程序倾向于那些需要编译的测试程序。在我们这个例子里，两个是相同的，所以我们设定为相同名称。

4. 添加makefile细节

如果你已经做了研究并发现Automake不能处理某些奇怪的目标，那么你可以把它写入Makefile.am，就像你写入通常的makefile一样。只要写一个目标并把它与操作关联起来：

```
target: deps
    script
```

并放在Makefile.am中的任何地方，而Automake将把它逐字地复制到最终的makefile中。例如，在5.3“与Python一起工作”中Makefile.am明确地指示了如何编译一个Python包，因为Automake自己不知道如何去做（它只知道如何编译单独的.py文件）。

Automake格式之外的变量也会被“原封不动”地添加进来。这在与Autoconf联合工作的时候特别有用，因为如果Makefile.am有类似下面的变量赋值：

```
TEMP=@autotemp@
HUMIDITY=@autohum@
```

而你的configure.ac是：

```
#configure is a plain shell script; these are plain shell variables
autotemp=40
autohum=.8

AC_SUBST(autotemp)
AC_SUBST(autohum)
```

那么最终的makefile将是：

```
TEMP=40
HUMIDITY=.8
```

这样你就有了一个便利的途径，也就是将Autoconf产生出的shell脚本内容写到最终的makefile。

3.3.3 配置脚本

shell脚本configure.ac产生两个输出：一个（在Automake的帮助下的）makefile，另一个叫作config.h的头文件。

如果你打开到现在为止产生的任何一个configure.ac的例子，你可能已经注意到，它看起来就是一个shell脚本。这是因为它利用了很多一系列的被Autoconf预定义的宏（用m4宏语言）。其他的部分保证展开后就是看起来很熟悉的shell脚本。就是说，configure.ac不是一个产生configure shell脚本的配方或者规格，它就是configure，只是被很多令人印象深刻的宏压缩了。

m4语言并没有太多语法。每个宏都是类似函数的，在宏的名字后面带有括号，列出分号分割的参数（如果有参数的情况下；否则括号一般就可以不带了）。虽然多数的语言使用'literal text'，但是Autoconf中的m4使用 [literal text]，而且为了避免类似m4过度解析你的输入的情况，建议你将所有的宏输入放到中括号中。

Autoscan产生的第一行是一个很好的例子：

```
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
```

我们知道这个将产生几百行的shell代码，并且在其中的某处，给定的元素将被设定。根据任何现实中的情况来改变方括号中的值。你常可以忽略一些元素，如果你不想从你的用户听到任何意见，那么类似：

```
AC_INIT([hello], [1.0])
```

的语句是有效的。在极端的情况下，某人可能不会给类似AC_OUTPUT的宏任何参数。在这种情况下你其实不需要再担心括号了。

警告

当前m4文档的用户将使用方括号标注所有的可选参数，这一点可不是瞎说。所以请记住，在为Autoconf准备的m4宏当中，方括号字面上意味着不需要扩展的文本，但是在m4宏文件中它代表的却是一个可选的参数。

> 回忆起shell允许你把if test ..; 写成 if [….];，因为configure.ac只是一个压缩的shell脚本，它可以包含这样的脚本代码。但是你需要使用if test…的样式，因为中括号会被m4“吃掉”。

我们需要为一个有效的Autoconf文件准备什么样的宏呢？按照出场顺序来讲如下所示。

- AC_INIT（…），已经讲过了。
- AM_INIT_AUTOMAKE，指示Automake产生一个makefile。
- LT_INIT安装Libtool，当且仅当你安装一个共享库的时候会需要它。
- AC_CONFIG_FILES（[Makefile subdir/Makefile]），告诉Autoconf遍历所有的文件并将类似@cc@的变量替换为恰当的值。如果你有多个makefile（一般是在不同的子目录中），那么在这里罗列出来。
- AC_OUTPUT，产出结果。

这样我们用简简单单的4、5行就为任何地方的任何POSIX系统产生有效的发行包提供了规范，而且Autoscan可能还会为我们写其中3行。

但是真正把configure.ac从“有用”提升到“智能”的艺术在于，预测某些用户可能遇到的问题，并找到Autoconf宏来探测这些问题（甚至于，如果可能的话，修正这些问题）。拿你之前见过的例子来说：我推荐添加AC_PROG_CC_C99到configure.ac中以检查一个C99编译器。POSIX标准要求必须有一个叫作c99的命令代表编译器，但是并不是因为POSIX这么说了就意味着每个系统就会这么做，所以一个好的配置脚本必须检

查一下。

检查库是否已经存在是必须检查的先验条件。回过头来看一下Autoconf的输出，config.h是一个标准的C头文件，包含了一系列的#define语句。例如，如果Autoconf验证了GSL的存在，你将在config.h中看到：

```
#define HAVE_LIBGSL 1
```

然后你就可以把#ifdef放在你的C代码中，以实现在合适的环境里的恰当的行为。

Autoconf的检查并不是仅仅依靠名字来寻找库，Autoconf还希望库实际上也能用。它会写一个什么也不做的程序来调用库里面的任意函数，然后试图将程序和该库连接。如果这个连接成功了，那么连接器就能按期望找到并使用这个库。所以Autoscan不能自动产生一个针对这个库的检查，因为它不知道从中可以找到哪个函数。检查这个库的宏是一行的代码，你向其提供库的名字和一个它可以用来检验的函数，例如：

```
AC_CHECK_LIB([glib-2.0],[g_free])
AC_CHECK_LIB([gsl],[gsl_blas_dgemm])
```

对于你要使用的函数库，如果你不确定它是不是100%的C标准库，那么为每个你使用的库添加一行到configure.ac里面。而你添加的一行代码将展开成恰当的shell脚本。

你也许回忆到，包管理器总是把库分成共享的目标文件包和带有头文件的开发用包。你的库的用户可能并不记得（或者根本不知道）去安装头文件包，那么我们来检查一下，例如：

```
AC_CHECK_HEADER([gsl/gsl_matrix.h], [AC_MSG_ERROR(
    [Couldn't find the GSL header files (I searched for \
    <gsl/gsl_matrix.h> on the include path). If you are \
    using a package manager, don't forget to install the \
    libgsl-devel package, as well as libgsl itself.])])
```

请注意两个逗号（宏的参数是这样的形式）：要检查的头文件，如果发现采取的动作，如果没发现采取的动作，而我们把第二个留作了空白。

在一个编译中还会有什么能出错呢？要熟知世界上所有计算机的所有花招，并成为其权威是很困难的，因为我们每个人只能接触到几台机器而已。Autoscan会给你一些好的建议，如果你在configure.ac中添加了一些元素，那么你会发现运行autoreconf会产生一些警告。它会给你好的建议——那就请你遵从这些建议。但是我所见过的最好的参考是Autoconf的手册本身,包括一系列的POSIX标准、不成功的实现和实战建议。手册中分类讲述了一些Autoconf处理的问题，并且（非常清醒地）略去了一些其他的吹毛求疵的部分。手册的某些部分是针对你如何写代码的好建议，同时有一些对系统特性的描写以及对应的Autoconf宏的名字，以帮助你在与自己情况符合的时候在自己的configure.ac中使用这些宏。

VPATH建造

假设在～/pkgsrc下有一个源文件包；最典型的办法是进入这个目录去编译它。但是你也可以从别的地方编译它。

```
mkdir tempbuild

cd tempbuild

~/pkgsrc/configure
```

> 这被叫作vpath建造，如果pkgsrc是一个共享的或者只读的目录，这种方法就很有效了，或者你可以编译同一个包的不同变体。
>
> 为了使用vpath功能，Autoconf定义了一个srcdir环境变量，可以用下面的方式来使用。
>
> ● 在configure.ac的脚本片段中用$srcdir。
>
> ● 在自动生成的makefile 中用$（srcdir）。
>
> ● 在configure.ac中AC_CONFIG_FILES列举出的文件中通过@srcdir@，Autoconf会把它改写为正确的目录。
>
> 现在值得测试一下你的项目可以在不同的目录建造出来了。如果一个文件没有被发现（对于测试用的数据文件最容易发生这个问题），这个时候你就需要用一种上面列出的方法来指定文件的位置了。

Shell的细节

因为configure.ac是一个用户将要运行的configure脚本的压缩版，你可以把任何你喜欢的shell代码放在里面。在你做这个之前，再次确认现有宏不支持你想做的——一般来说，你的情况不会特殊到之前的Autotools用户都没有遇到过。

如果你没能在Autoconf包中找到合适的宏，你可以检查一下针对附加宏的GNU Autoconf macro archive。你可以把这些保存在你的项目目录中的m4子目录，这样Autoconf就可以找到并利用它们。参见[Calcote，2010]，其中对Autotools的各种复杂的细节有很好的描述。

最好能有一个通知，告诉用户接下来要进行的设定。这不需要宏来实现这个，因为你所需要的就是echo。这里是一个通知的例子：

```
echo \
"-------------------------------------------------------------
```

```
Thank you for installing ${PACKAGE_NAME} version ${PACKAGE_VERSION}.

Installation directory prefix: '${prefix}'.
Compilation command: '${CC} ${CFLAGS} ${CPPFLAGS}'
Now type 'make&& sudo make install' to generate the program
and install it to your system.

------------------------------------------------------------"
```

这个通知使用了Autoconf定义的几个变量。关于那些系统为你预定义好以便使用的shell变量，你可以翻阅相关的文档，也可以通过浏览configure自身来找到它们。

在5.3“与Python一起工作”中，还有一个更加深入的关于Autotools如何工作的例子：连接一个Python库。

第4章　版本控制

本章讨论版本控制系统（RCS），其目的是保存一个项目在开发过程中的众多版本的代码快照，例如一本书、一封缠绵悱恻的情书、或者一个程序的开发过程的各个阶段。版本控制系统主要给我们提供三种重要功能。

- 为我们的文件系统提供了一个时间维度，这样我们就可以知道一个文件在上一周是什么样子，以及它是如何变成现在的样子的。单单这个能力就足以使你成为一个更自信的作者。
- 可以记录一个项目的不同版本，比如我的版本和我的合作者的版本。甚至在我自己的工作中，我也可能需要一个项目的带有一些实验性特点的版本（一个分支），这个分支版本是需要与稳定的、不能在运行中发生任何意外的稳定版本隔离的。
- 在http://github.com网站上有大约218000个自称用C语言编写的项目。另外也有一些C语言项目存在于一些比较小的RCS的版本库服务器，比如GNU的Savannah。即便你不是想修改代码，从那些代码库克隆文件也可以作为一种快速安装程序或者库的手段。当你自己的项目可以向公众发布的时候（或者在这之前），你可以把公开那些代码库作为另外一种发行的方法。
- 现在你和我都有了同一个项目的不同版本，也都同等地拥有了研究代码不同版本的能力，版本控制给了我们尽可能方便地融合各自不同分支的能力。

本章将讲述Git，这是一个分布式版本控制系统。也就是说，项目的任何给定版本都是这个项目独立的库，库中包含项目的所有历史记录。另外有Mercurial、Bazaae等分布式版本控制系统。这些系统的功能基本上都是一一对应的，而且那些主要的差别随着时间的流逝已经逐渐消失了，所以读过本章以后，你也应该能直接使用其他的工具。

4.1 通过diff查看差异

最简陋的版本控制手段就是使用diff和patch，这些都是属于POSIX标准的，所以基本上肯定已经安装在你的系统里了。你很可能已经在你的硬盘的某个地方，而且很可能是同一个地方，有两个基本相似的文件。如果没有，就造两个文本文件，比如改几行什么的，并把更新后的版本保存成一个新的名字。输入：

```
diff f1.c f2.c
```

你会得到一个列表，有点像给机器读的而不是给人读的，显示出在两个文件之间你改变的那几行。通过diff f1.c f2.c>diffs建立管道导入到一个文本文件，然后打开diffs文件，你的文本编辑器将给你一个色彩高亮的容易读懂的版本。你将看到一些行给出了文件名和在那个文件中的位置，可能也伴随一些没有在两个文件中改动的行，和一些用+和-开头的行表述出该行是增加的还是被移除的。用-u配置运行diff，可以得到在增加/删减周围的几行内容。

假设你项目的两个版本，*v*1和*v*2，存放在两个目录中，可以通过迭代选项（-r）为整个目录产生统一格式的diff文件：

```
diff -ur v1 v2 > diff-v1v2
```

patch命令可以读取diff文件并执行改变列表。如果你和你的朋友都拥有项目的*v*1版本，你可以把diff-v1v2文件发给你的朋友，那么他就可以运行：

```
patch < diff-v1v2
```

从而把你所有的改变应用在她的*v*1版本上。

或者，如果你没有朋友，你也可以时不时在你的代码上运行diff命令，从而记录下随着时间的流逝你所做的改变。如果你发现你在代码中插入了一个错误，diff文件应该是你第一个去查找你动了什么不该动的东西的地方。如果这还不够，你已经删除了*v*1，你可以从*v*2目录反方向运行patch，patch–R < diff-v1v2，把*v*2回退到*v*1。如果你在*v*4，你甚至可以想象运行一系列的diff来回退：

```
cd v4
patch -R < diff-v3v4
patch -R < diff-v2v3
patch -R < diff-v1v2
```

相信你也看到了，像这样维护一系列的diff文件是琐碎而容易出错的。

因此我们需要版本控制系统，它将为你制作和追踪diff文件。

4.2 Git的对象

像其他工具一样，Git是一个C程序，并且基于一小组对象。关键对

象是提交对象，它们就是一组diff，和之前我们看到的diff有基本一致的格式。给定上一个提交对象和一些以此为基线的变化，一个新的提交对象就可以打包所有的信息。它利用了一些来自index的支持，记录自从上一次提交开始所注册下来的变化列表，主要的用途是为了产生下一个提交对象。

提交对象连接起来以形成一个树。每个提交对象有（至少）一个父提交对象。用patch和patch–R在相邻版本之间单步切换，就好像沿着树上下遍历。

在Git源代码中，版本库本身不是一个正式的单一对象，但是我把它等同为一个对象，因为在通常的操作中，一个人将定义类似新建、复制和释放整个的版本库的动作。这样你可以得到一个用来工作的新版本库：

```
git init
```

OK，你现在有一个版本控制系统了。你可能看不到它，因为Git把所有它自己的文件放在一个叫作.git的目录中，而名字中的点意味着所有常用的类似ls的工具都把该目录视为隐藏的。你可以通过类似ls -a或者在你常用的文件管理器中的显示/隐藏文件的选项来查看它。

或者，可以通过git clone来复制一个版本库。这正是你从Savannah或者Github上得到一个项目的方法。为了用git来得到Git的代码：

```
git clone https://github.com/gitster/git.git
```

读者也许有兴趣将本书中的实例代码库克隆到本地：

```
git clone https://github.com/b-k/21st-Century-Examples.git
```

如果你想对在～/myrepo中的一个版本库里的某个项目做测试，并担心会弄坏什么东西，可以先转到一个临时目录（假设为mkdir ～/tmp; cd ～/tmp），用git clone～/myrepo来克隆你的版本库，然后尽情地做试验。试验结束后删除这个克隆（rm –rf～/tmp/myrepo），这对原始的版本库没有任何影响。

因为所有关于你的版本库的数据都在你的项目目录下的.git子目录中，所以释放一个版本库的方法就非常简单了：

```
rm -rf .git
```

把整个版本库做成一个如此自包含的程度，意味着你可以做一个额外的备份，以便在家里和办公室之间分别保存，把所有的东西复制到一个临时文件来做一个快速的试验等类似的操作，而不会遇到任何麻烦。

我们基本上已经准备好产生一些提交对象了，但是因为提交对象记录着从最开始点或者前一个提交到目前的差异，所以我们必须先产生一些差异，然后才能建立commit。Index（Git source：struct index_state）是一个变化列表，记录着想在下一个提交中提交的变化。它的存在是因为我们实际上并不希望项目目录中的每个差异都被记录下来。例如，gnomes.c和gnomes.h将产生gnomes.o和可执行文件gnomes。你的版本控制系统应该记录下gnomes.c和gnomes.h的变化并让它的东西都随需要重新生成。所以index的关键操作就是向它的变化列表里添加元素。用：

```
git add gnomes.c gnomes.h
```

来把这些文件加入index中。其他需要被加入index的文件追踪列表的典型变化还有：

```
git add newfile
git rm oldfile
git mv flie file
```

那些你对文件做出的改变虽然已经被Git追踪到，但并不能自动添加到index中，这一点可能叫别的版本控制系统的用户感到意外（请看下面的解释）。通过git add changedfile加入每个文件，或者用：

```
git add -u
```

某一点上，你可能在变化列表中有足够多的变化，可以在版本库中生成一个提交了。可以通过下面的命令产生一个新的提交：

```
git commit -a -m "here is an initial commit."
```

这个-m选项附加了一条消息到指定版本上，以后当你运行git log之后将看到这条消息。如果你忽略了这条消息，Git会按照环境变量EDITOR所指示的为你打开一个文本编辑器，这样你就可以输入这条消息了（默认的编辑器一般是vi；如果你想改变，可在你的shell的启动脚本中修改这个变量，例如.hashrc或者.zshrc）。

-a选项告诉git：很奇怪我忘记了运行git add–u，所以请在提交前运行它。在实践中，这意味着你从来也不用去特意地运行git add–u，只要你还记得在git commit -a中的-a。

警告

很容易发现有经验的Git专家致力于产生一系列连贯的、叙述干净的提交。专家级的Git作者应该创建2个提交，一个带有信息“添加了一个index对象”，一个带有信息“问题修复”，而不是用-a选项只产生一个提交信息，如“添加一个index对象，顺便加入一些问题修复”的提交。我们的作者需要有这样的能力去控制把哪些变化加入到index，这是因为默认来说变化不会自动添加到index，所以只加入那些有修改目的的程序变化，产生一个提交，接着再将其他变化加入到index，产生另外一个提交。

我发现一个博客作家花了好几页来描述他的提交操作：“在最复杂的情况下，我会打印出diff，彻底把它们读一遍，并把它们用六种颜色高亮显示……”然而，除非你可以称为一个Git专家，这些对index的过度控制，远超过你的需求。那就是，在git commit中不用-a是很少被掌握的高级技巧。在一个完美的世界里，-a应该是默认配置的，但现实中它不是，所以请不要忘记它。

调用git commit -a写一个新的基于index的所有的变化的提交到版本库，这个变化可以被追踪，写完以后它会清理index。保存好你的工作，现在就能添加更多的工作了。进一步——这是迄今一个版本控制系统的真实的、主要的益处——你可以删除任何你不想要的，因为可以确信你可以在需要的时候将它恢复回来。不要用大块的注释来注释到掉过时的代码了，这样会把你的代码搞得乱七八糟，直接将它们删掉就好了！

> 提示
>
> 在你提交之后，你非常有可能拍着额头想起你忘记了什么。你可以运行git commit -amend -a来重新做最后的提交，而不用执行另一次提交。

> **Diff/双重快照**
>
> 物理学家有时宁愿把光视为一种波，有时也视为一个粒子；同样，一个提交对象有时最好被认为项目在当时的一个完整的快照，或者有时作为一个它的父版本的diff。无论是哪个视角，它都包括一个关于作者、对象的名字（我们未来会讨论），你通过-m选项附加的消息记录，并且（除非它是最初的提交）一个指向其父提交的对象（们）的指针。
>
> 那么从内部来看，一个提交是一个diff还是一个快照呢？两种都有可能。以前Git总是保存一个快照，除非你运行git gc（垃圾清扫）来把整套的快照压缩为一组差异（或者叫diff）。用户抱怨必须记得去运行git gc，所以它现在是在一定的命令后自动执行这个动作，意味着Git可能是（但也并不总是）以diff的形式存储的。

产生一个提交对象后，你与它的互动将主要是查看它的内容了。你可以用git diff来查看有哪些改变，也就是提交对象的核心，以及git log来看元数据。

关键的元数据是对象的名字，它对用户不友好但是却有其存在意义的：SHA1 hash，一个40位的十六进制数值被赋值给一个对象，我们可以假设没有2个对象有同样的hash值，且同样的对象将在版本库中的每个版本使用同样的hash值。当你产生了提交对象，你将在屏幕上看到这个对象的hash数值的最前的几个数字，并且你可以运行git log来看当前

提交对象的历史提交对象列表。列表中有每个提交的hash数值和当你提交的时候写的人类语言信息（并参见git help log以了解其他的可用元数据）。幸运的是，你只需要hash值的前几位（只要不和其他提交的hash值重复就行）就能提供唯一的的标识以区分你的提交。所以如果你看着log并决定你想要检查版本号fe9c49cdddac5150dc974de1f7248a1c5e3b33e89，你可以这么做：

```
git checkout fe9c4
```

这简直就是patch提供的通过diff来实现的时间旅行，重新回到fe9c4的提交时候的项目状态。

因为一个给定的提交仅有指向其父提交的指针，而没有指向其子提交的指针，当你在导出一个旧的提交后检查git log，你可以看到导致这个提交的对象的历史的痕迹，而看不到这个提交之后的。有个很少被使用的git reflog将把版本库所知道的完整的提交对象列表展示给你，但是跳到项目最当前版的一个相对容易的办法是通过一个标签（tag），一个人类可理解的名字。这样你就不用在log中上下翻找了。标签在版本库中是按照单独的对象来维护的，并有一个指向提交对象的指针。最常用的标签是master，是主分支的最新提交（主分支可能是你拥有的唯一的分支；我们将在下面讲解分支）。那么，为了从过去返回到现在状态，使用：

```
git checkout master
```

回过来继续讨论git diff，其表示从你最近的提交版本开始你都做了什么改变。它的输出是通过git commit -a的时候需要写入下一个提交对

象的内容。如同传统的diff程序的输出，你可以用git diff > diffs将写入一个文件，使你在彩色高亮的文本编辑器中更适于阅读。

没有参数，git diff就直接显示最近的提交和目前项目目录内容的diff。如果有一个提交对象的名字，git diff显示从那个提交到项目目录内容之间的一系列变化。如果用两个名字，它就显示从一个提交到另一个提交之间的变化：

```
git diff              Show the diffs between the working directory and the
 index.
git diff 234e2a       Show the diffs between the working directory and the
 given commit object.
git diff 234e2a 8b90ac Show the changes from one commit object to another.
```

提示

有一些名字可以避免你去使用那些十六进制的hash值。HEAD代表目前最后一个提交，也就是一个分支的头部；如果它不是，git会给出你一个错误报告，指出这是一个“分离的头”（detached head）。

加上～1代表这个提交的父提交，～2代表祖父提交，以此类推。这样，所有下面的内容都是合法的：

```
git diff HEAD~4        #Compare the working directory to four commits ag
o.
git checkout master~1  #Check out the predecessor to the head of the mas
ter branch.
```

```
git checkout master~  #Shorthand for the same.

git diff b0897~ b8097 #See what changed in commit b8097.
```

到这里，你就知道了如何做如下事情：

- 保存你的项目的持续的改动。
- 得到一个提交版本的日志清单（log）。
- 找出你最近的改变和添加。
- 导出早前的版本，这样有必要的时候你就可以恢复到早前的状态。

拥有一个组织良好的备份系统，可以使你随心所欲地删除和恢复代码，这也会使你成为一个更好的程序员。

stash

提交对象是参考点，而大部分的Git活动是基于这些参考点发生的。例如，Git倾向于相对于提交使用补丁，你也可以跳到任何提交，但是如果你当前的工作目录的内容没有对应的提交来恢复的话，跳过去也可以，但是你却跳不回来了。在当前的工作目录中存在未提交的改变时，Git不会做什么事情。它一般是问你是否要提交新的工作，然后如果你有这个意愿再为你执行操作。如果想回到某个提交，你应该把自从最后一个提交以后所做的工作保存在某处，然后回到你最后的提交，执行某个操作，当你结束跳转或者打补丁后，再从保存的地方恢复你以前的工作。

因此有了stash，一种特殊的提交对象，大体上和你运行git commit -

a的时候效果一样，但是具有一些特殊功能，比如保留所有你的工作目录中没有追踪的内容。下面是典型用法：

```
git stash
# Code is now as it was at last checkin.
git checkout fe9c4

# Look around here.

git checkout master # Or whatever commit you had started with
# Code is now as it was at last checkin, so replay stashed diffs with:
git stash pop
```

假设你的工作目录存在变化，另一个有时更合适的导出代码的替代做法是git rest --hard。这个操作把工作目录返回到你上次导出代码的状态。这个命令看起来很严厉，它也的确是这样：要知道这样做是要把从上次导出代码所做的工作全部扔掉。

4.3 树和它们的分支

版本库中有一个树状结构，是在一个新库的第一个作者运行git init的时候产生的。你应该熟悉数据结构中的树，包含一组节点，同时每个节点都和自己的父节点和子节点的连接（而类似Git这样的比较古怪的树，父节点可能不止一个）。

确实，除了最初的那个，所有的提交对象都有一个父对象，并且对象本身记录了自己和其父对象之间的diff。整个序列中最终的节点，也就是树枝的末梢，用一个分支名字作为标签。我们有意的安排是，分支的末梢和产生分支的diff序列之间是一一对应的。这个一一对应关系意味着我们可以互换指代分支和在分支末梢的提交对象。如果master分支

的末梢是编号234a3d的提交，则git checkout master和git checkout 234a3d是完全对等的（直到一个新的提交被写入，替换了master的标签）。这还意味着一个分支的提交对象列表可以在任何时候被重新推导出，只要从命名这个末梢的提交对象开始，回溯到树的起源。

习惯上一般是保持主分支随时具有完整的功能。当你想添加一个新的特性或尝试新的研究的时候，就为其创建一个新的分支。当这个分支充分实现了功能的时候，你可以用下面的方法把新的特性融合到主分支中。

有两种方法可以从你的项目的当前状态创建新的分支：

```
git branch newleaf    # Create a new branch...
git checkout newleaf  # then check out the branch you just created.
    # Or execute both steps at once with the equivalent:
git checkout -b newleaf
```

在创建新的分支以后，可以通过git checkout master和git checkout newleaf在两个分支间切换。

你现在工作于哪个分支？可以用下面的方法查询：

```
git branch
```

该命令会列出所有的分支，并在当前活动的分支前面加一个*。

如果你要建立一个时光机器，回到你出生前并杀掉你的父母，那会怎么样?如果说我们从科幻小说中学到了什么，那就是如果我们改变了历史，现在却不会改变，但是一个新的历史路径产生了。所以如果你导出一个老的版本，做一些修改，然后把你最新的修改用一个新的提交对

象来交付，那么现在你有一个与主分支（master）差异甚大的新分支。当完成这样的分叉之后运行git branch，你会发现自己现在处于（no branch）。没有标签的分支容易产生问题，所以只要你处于（no branch），那么运行git branch -m new_branch_name来命名你刚分出的分支。

> 可视化帮助
>
> 有几个图形化界面可以利用，尤其对于跟踪分支的派生和融合特别有用。建议以在Tk为基础的GUI下尝试gitk或git gui，或者运行git instaweb以启动一个web服务器，这样你就可以在一个浏览器进行互动了，或者求助你的包管理器或者Internet搜索引擎来找到更多的选择。

4.3.1 融合

到此为止，我们通过以一个提交对象作为开端，并应用一系列的在index中的改动来产生了新的提交对象。一个分支也是一系列的改动，所以任意给定一个提交对象和来自一个分支的一系列的改动，我们应该能创立一个新的提交对象，就是来自于将分支的改动都应用于已经存在的提交对象。这就是一个融合。为了融合所有的发生在newleaf与master之间的变化，切换到master分支然后用git merge：

```
git checkout master
git merge newleaf
```

例如，你已经用master分出的一个分支开发了一个新的特性，并且它最后通过了所有的测试；那么把在开发分支中的所有diff应用于master将产生一个具有响当当的新功能的新的提交对象。

让我们假设，当工作于新的特性的时候，你从没有导出master代

码，因此没有对它做任何修改。那么当你把来自于别的分支的diff序列应用于master就相当于把该分支中的每个提交对象变化的录像做了快速向前的播放，这在Git术语中叫作“快进”。

但是如果对master做了任何改变，那么这就不再是一个简单的把所有的diff快速应用的问题了。例如，假设在分出分支的时间点，gnomes.c有如下代码：

```
short int height_inches;
```

在Master中，你移除了修饰类型：

```
int height_inches;
```

newleaf的目的是转换为公制单位：

```
short int height_cm;
```

此时，Git就无法自动处理了。要明白程序员的意图才能合并这一行代码。Git的解决方案是把你的文本文件修改成同时包含两个版本的样子，就像：

```
<<<<<<< HEAD
int height_inches;
=======
short int height_cm;
>>>>>>> 3c3c3c
```

这时融合就停下来了，等待你对文件的编辑，以体现你希望看到的变化。在这个例子里，你可能把Git留在文本文件中的这个5行的段落减少为：

```
int height_cm;
```

这就是一个无法快进的情况下的融合过程，也就是在两个分支派生后各自都存在一定变化的情况。

1．运行git merge *other_branch*。

2．很有可能，被告知存在你必须解决的冲突。

3．用git status来检查未融合的文件列表。

4．选择一个文件来手工检查。用一个文本编辑器打开它，如果有内容冲突，找到需要手工融合的标记。如果有一个文件名或者文件位置出现冲突，把这个文件转移到合适的位置。

5．运行git add your_now_fixed_file。

6．重复3～5步，直到所有的文件和未融合的文件都被提交。

7．运行git commit来确认融合并结束

请放心，这个手册基本都能适用。Git在融合方面是保守的，并不会在某种情形下自动去做什么可能使你丢失一些工作内容的事情。

当你做好了融合，所有相关的且出现在这一旁支的diff将被体现在最终的提交对象中，所以习惯上是删除旁支：

```
git delete other_branch
```

other_branch标签已经被删除了。不过提醒您一下，那些引导你到

这里的提交对象还在你的版本库中。

4.3.2 迁移

假设你有一个主分支，并且在星期一那天分出了一个测试分支。然后在星期二到星期四，你对主分支和测试分支都做了很多改变。在星期五，当你试图把测试分支融合到主分支，你得到一堆小冲突需要解决。

让我们重新开始这一周。星期一那天你从主分支中分出了测试分支，意味着两个分支共享一个共同的祖先，即主分支的在星期一的提交。在星期二，你有了一个主分支的新的提交；假设是编号为abcd123的提交。在那天结束的时候，你把所有发生在主分支的改动用在测试分支上：

```
git branch testing  # get on the testing branch
git rebase abcd123  # or equivalently: git rebase main
```

利用rebase命令，所有自从共同的祖先开始发生在主分支上的变化被重新应用于测试分支。你可能需要手工融合，但是因为只有一天的工作量需要融合，我们希望融合的任务是可管理的。

现在所有到abcd123的变化在两个分支都出现了，这就好像两个分支实际上是从这个提交迁移分化出来的，而不是星期一的那个提交。这就是为什么这个过程会有这个名字：测试分支已经被设定为从主分支的新的点迁移并分化出来了。

你也可以在星期三、星期四和星期五执行迁移，相对不那么痛苦，因为整个这一周中测试分支一直保持着与主分支的一致。

迁移经常被认为是Git的高级技巧，因为那些没有diff应用的别的系统并不具有这样的技术。但是在时间中迁移和融合是基于同样的立意的：两种做法都把从一个分支得到的改动应用于另一个分支以产生一个提交。而唯一的问题是，是否你想要把两个分支最终放在一起（此时叫作融合），还是希望每个分支继续它们的独立生存状态一段时间（此时叫作迁移）。常见的用法是从主支到旁支迁移diff，而从旁支到主支融合diff，所以这里出现一个两种实践之间的对称。请注意，让diff在多个分支堆起来会使得最终的融合成为一个痛苦，所以合理频率的迁移是有益的。

4.4 远程版本库

到目前为止，发生的一切都还是在一棵树的范围内。如果你从某地方克隆一个版本库，那么在克隆的那一刻，你和这个原点都有一模一样的树和一模一样的提交对象。然而，你和你的同事将继续工作，所以你将肯定会添加新的和不同的提交对象。

你的版本库有一个远程的列表，指向与这个库有关的存放在别处的版本库。如果你通过git clone得到你的版本库，那么你用来克隆的那个版本库就被命名为origin。在常见的情形下，这就是你将用到的唯一的远程版本库。

当你第一次克隆并运行git branch的时候，你将看到一个单独的分支，不管这个最初的版本库有多少的分支。而运行git branch -a可以看到Git知道的所有不论是本地的还是远程的分支。如果你从Github克隆一个版本库，打个比方，你可以用这个来检查是否别的作者已经推送

（push）了其他的分支到这个中心的版本库中。

在你的本地版本库的那些分支的复制就和你第一次拉取（pull）的时候是一样的。下一周，为了用原始的版本库的信息以及更新的远程分支，运行git feth。

现在你的版本库有了与时俱进的远程分支了，你可以用分支的全名来融合，例如，git merge remotes/origin/master。

作为git fetch; git merge remotes/origin/master的简化，用：

```
git pull origin master
```

来拉取远程变化，并把它们一次性地融合进入你当前的版本库。

相反的动作是push，是你用来根据你的最新的提交来更新远端版本库（而不是你的index和工作目录的状态）。如果你是在一个叫作bbranch的分支并想用同样的名字推送给远程库：

```
git push origin bbranch
```

非常有可能，当你推送你的变化时，应用来自你的分支的改变到远程分支将不是一个快进（如果是，那就是你的同事还没做什么工作）。解决一个非快进式的融合一般来说需要人的干预，而远程那里非常有可能没有人。因此，Git将允许只有快进式的提交推送。那你怎么确保你的推送是一个快进呢？

1．运行git pull origin以得到自从你上次导出所做的变化

2．以前面讲述的方式融合，在这里你是一个解决问题的人类，而一台计算机不能做到这一点

3．运行git commit –a –m"*dealt with merges*"

4．运行git push origin bbranch，因为Git现在只用一个单一的diff，所以可以自动实现。

目前，我假设你在本地的分支里，而且远端的分支里有同样的名字（也许两边都是master）。如果你想跨越名字，使用冒号分割的*source*:*destination*分支名字对。

```
git pull origin new_changes:master #Merge remote new_changes into local ma
ster
git push origin my_fixes:version2 #Merge the local branch into a different
ly named remote.
git push origin :prune_me        #Delete a remote branch.
git pull origin new_changes:      #Pull to no branch; create a commit name
d FETCH_HEAD.
```

上面的操作都不改变你当前的分支，但是有些命令产生了一个新的分支，你可以通过checkout切换过去。

中心版本库

不管所有的关于去中心化的讨论怎么说，最方便的分享方式还是建立一个大家都可以克隆的中心版本库，也就是说每个人都有一个同样的原始版本库。这也是从Github和Savannah下载的一般的工作方式。当为这类任务建立一个版本库的时候，用git init --bare，意味着没有人可以在那个目录做任何工作了，用户必须先克隆再做自己的工作。这里也有几个好用的许可配置项，比如--shared=group使得一个POSIX组的所有成员可以有这个版本库的读写权限。

你不能向一个“已占用”的，也就是版本库的所有者已经导出代码的版本库推送一个分支；如果这么做将引发混乱。如果发生了这个事情，叫你的同事用git branch产生一个不同的

> 分支，然后在目标分支处于这个背景的时候推送。
>
> 或者，你的同事可以设定一个公共的“未占用”的版本库和一个私有的工作版本库。当你向公共的版本库推送，而你的同事在恰当的时候拉取（pull）他工作的版本库的代码变动。

Git的版本库的结构不是特别复杂的：提交对象代表与它们父节点之间的变化，并被组织成一棵树，而一个index收集包含在下一个提交中的变化。但是利用这些元素，你就可以把你的工作组织成多个版本、自信地删除内容、创建试验性的分支以及当通过所有的测试后把它们融合回主分支，也可以把你的同事的工作融合到你自己的工作中。最后，git help和你常用的Internet搜索引擎将教会你大量的实用技巧，这样你就可以把所有这些事情做得更加顺畅。

第5章　协助开发

如今的编程语言已经有些数不胜数了，而且它们中的大多数都具有C语言接口。本章相对简短，介绍了与其他语言混合编程的接口问题，并用Python语言为例进行了详细的示范。

每种语言都有自己的打包和发行的习惯，就是说当你完成了C语言和宿主语言的桥接代码，就将不得不面对这样一个任务，即如何用打包系统来编译和连接程序的每个组成部分。这给了我一个机会以向你展示一些更加高级的Autotools技巧，比如有条件地处理一个子目录和添加安装挂钩。

5.1　动态装载

再讲解别的语言之前，我们应该花一点时间来感谢C语言的两个函数，正是这两个函数，使得动态装载成为可能，它们是dlopen和dlsym。这两个函数打开动态链接库并抽取出符号，例如库中的静态对象或者是函数。

这两个函数是POSIX标准的一部分。Windows系统有类似的函数，但是函数名是LoadLibrary和GetProcAddress；为了简化说明，我将坚持用POSIX名字。

“共享目标文件” 这个名字有很好的描述性：这个文件包含一系列的

目标（object），包含函数和静态定义的结构，而且它们可以被其他的语言使用。

使用这个文件就像是从包含一系列元素的文本文件中获取一个元素一样。对于文本文件，你可能需要首先打开fopen文件，从而获得一个文件的句柄，然后调用正确的函数来搜索文件并且返回要取得的项目的指针。对于一个共享目标文件，打开文件的函数是dlopen，搜索文件的符号使用dlsym。神奇的地方在于你如何使用传递回来的指针。对于一系列文本元素，你有一个指向一般文本的指针，并且你可以做一些文本处理的工作。如果你使用dlsym获得了一个指向函数的指针，你就可以调用这个函数；如果你获得一个指向结构的指针，你就可以立即使用这个结构就像它已经被初始化一样。

当你的C程序调用一个动态库中的函数时，以上的过程描述了这个函数是如何被得到并被调用的。当主程序被发布后，一个有插件系统的程序装入不同作者写的函数库。如果脚本语言想调用C代码，它们会使用自己语言的与dlopen和dlsym具有相同功能的函数。

为了演示dlopen和dlsym能做什么，例5-1是一个C解释器的原型。

1．要求用户键入C函数的代码

2．编译这个函数成为一个共享目标文件

3．通过dlopen装入共享目标库

4．通过dlsym得到函数

5．执行用户刚刚建立的函数

以下是示例运行：

```
I am about to run a function. But first, you have to write it for me.
Enter the function body. Conclude with a '}' alone on a line.

>>double fn(double in){
>> return sqrt(in)*pow(in, 2);
>> }
f(1) = 1
f(2) = 5.65685
f(10) = 316.228
```

例5-1　程序要求用户输入一个函数，现场编译，然后运行这个函数（dynamic.c）

```
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>

void get_a_function(){
    FILE *f = fopen("fn.c", "w");
    fprintf(f, "#include <math.h>\n"                    ❶
               "double fn(double in){\n"};
    char *a_line = NULL;
    char *prompt = ">>double fn(double in){\n>> ";
    do {
        free(a_line);
        a_line = readline(prompt);                      ❷
        fprintf(f, "%s\n", a_line);
        prompt = ">> ";
    } while (strcmp(a_line, ")"));
    fclose(f);
}

void compile_and_run(){
    if (system("c99 -fPIC -shared fn.c -o fn.so")!=0){  ❸
        printf("Compilation error.");
        return;
    }
```

```
    void *handle = dlopen("fn.so", RTLD_LAZY);          ❹
    if (!handle) printf("Failed to load fn.so: %s\n", dlerror());

    typedef double (*fn_type)(double);                  ❺
    fn_type f = dlsym(handle, "fn");
    printf("f(1) = %g\n", f(1));
    printf("f(2) = %g\n", f(2));
    printf("f(10) = %g\n", f(10));
}

int main(){
    printf("I am about to run a function. But first, you have to write it
for me.\n"
       "Enter the function body. Conclude with a '}' alone on a line.\n\n"
);
    get_a_function();
    compile_and_run();
}
```

❶ 这个函数把用户的输入写入到函数，包含一个数学库的头文件（这样，pow、sin等函数就可以使用了）以及正确的函数声明。

❷ 这里大部分是读库的界面。你给用户显示出一个提示，它会要求用户基于你的提示来提供输入，并且返回用户输入的字符串。

❸ 现在用户的函数完整地定义在*.c*文件中，使用标准的呼叫方式调用C编译器，你也可以修改C编译器的选项。

❹ 打开共享目标文件以读入对象，后绑定（Lazy binding）代表函数名字只在需要的时候才被解析。

❺ 函数dlsym会返回void *，所以你需要指定这个函数的类型信息。

这是本书中最和系统相关的例子。我使用GUN的Readline库，很多系统默认安装它，因为它会只用一行代码就解决用户的输入问题。我使

用system命令来调用编译器，但是编译器选项的不一致是臭名昭著的，所以这些选项也许需要在你的系统上做调整。

动态装载的缺点

现在让我们把这个程序整理一下，加入正确的#ifdefs来使用loadlibrary，以便它能在Windows系统上使用（虽然Glib已经为我们做这些了，参见Glib文档中的gmodules），并且把它建成一个完整的C 的读入—计算—打印的开发流程，这样做可行吗？

但是很不幸，使用dlopen和dlsym不能完成这个任务。例如，如果我想从目标文件抽取出单独的一行执行代码，我该如何告诉dlsym去完成这个任务呢？本地的变量丢失了，因为dlsym函数只能从共享目标文件中抽取那些声明为文件范围内的静态变量或者是函数。我们这个不太成熟的例子，已经显示了dlopen和dlsym的局限性。

即使你只把C语言当成全局变量和函数的组合，但还是有很多其他的可能。函数可以根据需要生成新的对象，全局变量可能是一个结构，结构中包含一些函数，或者就是一些函数名字的字符串，然后调用者通过dlsym来得到这些字符串。

当然，调用系统需要知道得到哪些符号以及如何使用这些符号。在上面的例子中，我指定函数有一个double fn（double） 原型。对于一个插件系统，调用系统的作者可以写下一个准确的指令集合，这个指令集合是关于需要什么符号以及如何使用这些符号的。对于一个装入任意代码的脚本语言来说，共享目标文件的作者需要写一些脚本代码以便正确的调用对象。

5.2 流程

你写了一些代码，为了能让宿主使用dlopen/dlsym轻松地调用你写的代码，我们还需要关注以下这些地方：

- 在C侧，写一个很容易从其他语言调用的函数。
- 在宿主语言侧编写包装函数，用来调用C函数。
- 处理C侧数据结构。它们可以被来回传递吗？
- 与C库链接。就是说，一旦所有的东西都编译好了，我们需要确保在运行时系统知道在哪里找到库。

5.2.1 为外来语言写程序

Dlopen/dlsym的局限性已经提示了我们应该如何写被调用的C代码。

- 宏会被预编译器读取，所以最终的共享库对此毫无知觉。在第10章，我将讨论在C语言中如何利用各种宏来更加愉快地使用函数，所以你甚至都不需要依赖一个脚本语言来准备一个更优化的接口。但是当你确实需要从外部链接C的库，你手里并没有那些宏，而你的包装函数不得不模仿出一些函数，那些函数以前是通过调用宏来完成的。
- 你需要告诉宿主语言，如何使用从dlsym获得每个对象，例如，以宿主语言能理解的方式提供函数的头文件。这就意味着每一个单独的可见对象在宿主一侧都需要另外的、重复的工作，这也意味着限制接口函数的数量是很必须的。一些C库有全功能的函数集，以及

一个简单的包装函数，你可以通过一个调用来完成一些典型的工作流程。如果你的库有几十个函数，可以考虑写几个容易的接口函数。最好是有一个只提供C侧库的核心函数的宿主包，而不是一个疏于维护的甚至崩溃而无法工作的宿主包。

- 这种情况下对象很重要。如果把第11章中的详细讨论做一个简述，那就是：一个文件定义一个结构和使用这个结构作为接口的一些函数，包括struct_new、syruct_copy、struct_free、struct_print函数等。一个设计好的对象具有一小部分接口函数，或者至少有一个最小子集给宿主语言用。就像在下面讨论的，用一个中心结构掌控数据也会使事情变得容易些。

5.2.2 包装函数

对于每个你期望用户调用的C函数，你都需要在宿主侧提供一个包装函数。这个函数有以下作用：

- 客户服务。宿主语言的用户对C缺乏了解，也不想去思考调用C的问题。他们期望帮助系统会提到你的函数，而且帮助系统也许就是直接与宿主语言绑定的。如果用户已经习惯于函数就是对象的一个元素，但你并没有在C侧如此准备，那么可以依据宿主侧的习惯来准备对象。
- 往返翻译。宿主语言对整数、字符串和浮点数的表示可能是int、char*和double，但是在大多数时候，你在宿主与C语言之间需要某种翻译，事实上，你将需要两次这样的翻译：一次是从宿主语言到C：然后在你调用了你的C函数，还有一次是从C到宿主语言的翻译，参见随后关于Python的例子。

用户将期望与宿主侧函数互动，所以很难避免为每个C函数都准备宿主函数，于是突然间我们需要维护双倍的函数。这里可能有所冗余：作为默认，你对C侧输入的描述通常也需要在宿主侧再次描述；一般来说，那些来自宿主的参数列表在你每次在C侧修改它们的时候也需要再次被检查。没有理由去抵触这些：冗余就是存在的，并且必须记住每次你改变C侧接口的时候检查宿主侧代码，必须这么做。

5.2.3 跨越边境的代理数据结构

暂时先忘记非C语言吧；让我们考虑两个C文件，struct.c和user.c，其中第一个文件中一个数据结构被按照带有静态链接的本地变量创建，而第二个将使用这个数据结构。

跨文件来引用数据的最方便的方式就是简单的指针：struct.c分配了这个指针，user.c收到了它，然后就可以了。这个结构的定义可以是公开的，也就是说user文件可以看到被指针所指向的数据，并任意改动。因为user中的这个过程是修改所指向的数据，所以在struct.c和user.c之间就不会有不一致。

相反地，如果struct.c发送出数据的一个复制，那么一旦user做了任何修改，我们在两个文件内部所持有的数据之间就存在了不一致。如果我们期望接受的数据被使用且立刻被扔掉，或者被当作只读来对待，或者struct.c将从不在乎再看到那个数据，那么把所有权传递给user就没有问题。

如果struct.c期望再次操作那些数据结构，我们应该传递一个指针；对于可以抛弃的结果，我们可以传递数据本身。

如果这个数据结构不是公开的呢？看起来user.c中的函数将收到一个指针，然后将无法用来做任何事情。但是它可以做一件事情：它可以把这个指针传递回struct.c。当考虑这一点时，你会发现这很常见。例如，你可能通过链表分配函数来分配（虽然Glib并没有这样的函数）一个链表对象，然后用g_list_append来添加元素，再用g_list_foreach来应用某个操作到所有的链表元素等，只要简单地把链表的指针从一个函数传递到下一个就可以了。

当建立C和另一种不能理解C的数据结构的语言的桥接时，这个方法经常被称为不透明的指针或者外部指针。因为typedef不是共享目标文件中的一个对象，所以它不能通过dlsym来获取，所有的在你C语言中的结构对于调用语言来说，都是不透明的。就像在两个.c文件之间互动的例子中，数据的所有者不存在模糊性，而且有足够的接口函数，我们也还可以做很多事情。这为世界上相当多种类的宿主语言解决了数据分享的问题，因为这里为传递不透明指针提供了一种明确的方法。

如果宿主语言不支持不透明指针，那么不管怎样就返回这个指针。一个地址就是一个整数，记下它并不会产生任何模糊的问题（见例5-2）。

例5-2　将一个指针按照普通整数来处理是完全可行的——在纯粹的C中没有什么理由要这么做，但是在与一个宿主语言工作时却可能是必要的（intptr.c）

```
#include <stdio.h>
#include <stdint.h> //intptr_t

int main(){
    char *astring = "I am somwhere in memory.";
```

```
    intptr_t location = (intptr_t)astring;    ❶
    printf("%s\n", (char*)location);          ❷
}
```

❶ intptr_t 类型保证有足够大的范围来保存一个指针地址 [C99 @7.18.1.4（1）& C11 @7.20.1.4（1）]。

❷ 当然，把指针转为一个整数会丢失所有的类型信息，所以我们重新显式地指定指针类型。这么做很容易出错，这就是为什么这个技术仅在处理不能理解指针的系统的情境中才有用。

什么地方会出错呢？如果你的宿主语言中整数的范围太小，那么取决于你的数据在内存中的位置，地址的值很大时就会失败。在这种情况下，你把指针写在一个字符串中会更好，然后当你重新得到这个字符串的时候，用strtoll（字符串转换为长整型）解析回来。方法总是有的。

另外，我们假设指针不会在其第一次被传递给宿主和宿主再次请求它的时候被移动或释放。例如，如果在C侧有一个realloc调用，一个新的不透明指针（以任何形式的）将必须递交给宿主语言。

5.2.4 链接

你已经看到了，动态链接到你的共享目标文件是用dlopen/dlsym和它们在Windows里面对应的功能来解决的。

但是这里有另一个层次的连接：如果你的C代码需要一个系统中的库并且需要运行时链接呢（参见1.3.3“运行时连接”）？在C的世界里，简单的答案是用Autotools，搜索所需要的库的路径并设定正确的编译选项。如果你的宿主语言的开发系统支持Autotools，那么你在连接系统中

的其他库的时候将不会遇到其他问题。如果你可以用pkg-config，那么它也可以做你所需要的。如果Autotools和pkg-config两样都没有，那么我只有祝福你能够找到如何使宿主的安装系统可靠而正确地链接你的库。不过看起来很多脚本语言的作者仍然认为将C库和另一种语言链接只是一个特殊情况，所以最好每次都手工处理。

5.3 与Python一起工作

本章的余下部分通过Python展示了一个例子，利用例10-12中的理想气体方程式，展示了我们上面介绍的那些过程；现在，既然这个函数已经存在，我们将聚焦如何打包它。Python有很多在线文档来为你讲解其工作的细节，但是例5-3已经为你展示了在工作中的一些抽象步骤：注册函数、将宿主格式的输入转换为常见的C格式，以及将常见C格式输出转换为宿主格式。然后把它们链接起来。

理想气体库只提供一个函数，以计算理想气体在一个给定温度下的压力，所以最终的包将会比仅能在屏幕上打印“Hello, World”的库多少有点意思。现在让我们启动Python并运行：

```
from pvnrt import *
pressure_from_temp(100)
```

第一行将pvnrt包中的所有元素装入Python命名空间。下一行调用Pythom命令pressure_from_temp，它将装入C函数（ideal_pressure）并完成相应的工作。

故事在例5-3中开始，提供了一段C代码，这段代码用Python API包装C函数，且把它注册为随后安装的Python包的一部分。

例5-3　理想气体函数的包装函数（py/ideal.py.c）

```
#include <Python.h>
#include "../ideal.h"

static PyObject *ideal_py(PyObject *self, PyObject *args){
    double intemp;
    if (!PyArg_ParseTuple(args, "d", &intemp)) return NULL;    ❶
    double out = ideal_pressure(.temp=intemp);
    return Py_BuildValue("d", out);                            ❷
}

static PyMethodDef method_list[] = {                           ❸
    {"pressure_from_temp", ideal_py, METH_VARARGS,
     "Get the pressure from the temperature of one mole of gunk"},
    { }
};

PyMODINIT_FUNC initpvnrt(void) {
    Py_InitModule("pvnrt", method_list);
}
```

❶ Python发送一个单一对象，包含所有的函数参数，类似argv。这一行把它们按照格式描述（类似scanf）读入C语言的变量。如果要解析的是双精度、字符串、整数，代码应该是这样的：PyArg_ParseTuple（args, "dsi", &indbl, &instr, &inint）。

❷ 输出也接收一个类型和C变量的列表，将它们合并在一起返回给Python使用。

❸ 其余部分是注册。我们必须添加一个{ }来结尾的列表，内容是函数库中的方法（包括Python名字、C函数、调用方式、单行文档），然后写入一个名叫initpkgname的函数读入这个列表。

这个例子展示了Python在处理输入和输出转换时并不是太麻烦（上面的转换发生在C语言侧，虽然有些别的系统在宿主侧做这个工作）。

这个文件包含一个注册段落，其实也还不赖。

接下来是编译问题，还需要真正解决一些问题。

5.3.1 编译与连接

就像你在3.3“用Autotools打包你的代码”看到的那样，设定一个Autotools以产生库，需要一个两行的Makefile.am和一个只需要轻微改动由Autoscan产生的configure.ac文件的模板。在这之上，Python有它自己的建造系统，即Distutils。因此我们要安装它，然后修改Autotools文件以使得Distutils自动运行。

5.3.2 Automake的条件子目录

我决定把所有的Python相关的文件都放在主项目目录下的一个子目录。如果Autoconf探查到正确的Python开发工具，那么我将要求进入那个子目录并开始工作；如果开发工具没有被找到，那么它可以忽略子目录。

例5-4展示了一个configure.ac文件，用以查找Python及其开发工具，当且仅当正确的组件都被找到后，就编译py子目录。头几行和之前的一样，从autoscan产生的内容加上来自以前的附加部分。下面的几行检查Python，这部分是我从Automake文档中剪切并复制来的。它们将用指向Python的路径产生一个PYTHON变量；对于configure.ac，有两个分别叫作HAVE_PYTHON_TRUE和HAVE_PYTHON_FALSE的变量；对于makefile，只有一个叫作HAVE_PYTHON的变量。

如果Python或者它的头文件找不到，那么PYTHON变量被设定

为“:”供以后比较。如果需要的工具都存在，那么用一个简单的shell的if-then-fi块要求Autoconf来配置py子目录与当前的子目录。

例5-4　一个用于Python编译任务的configure.ac文件（py/configure.ac）

```
AC_PREREQ([2.68])
AC_INIT([pvnrt], [1], [/dev/null])
AC_CONFIG_SRCDIR([ideal.c])
AC_CONFIG_HEADERS([config.h])

AM_INIT_AUTOMAKE
AC_PROG_CC_C99
LT_INIT

AM_PATH_PYTHON(,, [:])                                    ❶
AM_CONDITIONAL([HAVE_PYTHON], [test "$PYTHON" != :])

if test "$PYTHON" != : ; then                             ❷
AC_CONFIG_SUBDIRS([py])
Fi
AC_CONFIG_FILES([Makefile py/Makefile py/setup.py])       ❸
AC_OUTPUT
```

❶ 这些行检查Python，不存在就将PYTHON变量设定为“:”，再根据检查结果将HAVE_PYTHON变量设定为适当的数值。

❷ 如果PYTHON变量没有设定为“:”，那么Autoconf将进入py子目录；否则将忽略该子目录。

❸ 在py子目录中有一个Makefile.am文件，需要被转化为一个makefile；需要通知Autoconf还有这个任务。Autoconf会使用setup.py.in来生成setup.py文件。

提示

本章你将看到Autotools很多的新语法，比如本章前半段提到的AM_PATH_PYTHON，以及之后的Automake的all-local和install-exec-hook目标。Autotools的天性就是它是一个基本系统（我希望我已经在第3章说清了此事），带有一个为各种可以想象的突发事件或异常准备的挂钩。没有必要一定要记住它们，而且对于大多数情况，它们也不可能从基本法则中派生出来。用Autotols工作的本质就是当奇怪的意外出现的时候，我们可以期望在手册或者Internet上找到大量适当的处理方法。

我们也需要把子目录的事情告诉Automake，也就是另一个if/-then块，如例5-5所示。

例5-5　一个为带有Python子目录的项目主目录准备的Makefile.am（py/Makefile.am）

```
pyexec_LIBRARIES=libpvnrt.a
libpvnrt_a_SOURCES=ideal.c

SUBDIRS=.

if HAVE_PYTHON      ❶
SUBDIRS += py
endif
```

❶ Autoconf产生了这个HAVE_PYTHON变量，而这里就是我们用它的地方。如果它存在，Automake就将py添加到它的要处理的目录列表

中；否则它将只处理当前目录。

头两行指定了Libtool应该设定与Python可执行文件一同安装的共享函数库，名称为libpvnrt，这个库基于ideal.c中的源代码。之后，我指定第一个需要处理的子目录，也就是“.”（当前目录）。静态库应该在Python包装函数库编译之前编译，并且我们通过把“.”放在SUBDIRS列表的开头，确保它被第一个处理。如果HAVE_PYTHON检查的结果是OK，我们可以用Automake的+=操作符来把py目录添加到列表上。

这里有了一个设置，当且仅当Python开发工具被正确安装的时候处理py目录。现在进入py目录本身并审视Distutils和Autotools如何合作的。

5.3.3 Autotools支持下的Distutils

现在，你可能已经熟悉了编译程序和库的过程。

- 描述相关的文件（例如，通过Makefile.am中的your_program_SOURCES，或者直接在本书中一直在用的示例的makefile中的objest列表）。
- 描述编译器选项（一般是通过一个名为CFLAGS的变量）。
- 描述连接器需要的配置选项和附加的库（例如，GNU Make的LDLIBS或者GNU Autotools的LDADD）。

就是这三个步骤，虽然实际上还有很多问题，但是总的方法很清楚了。本书到这里，我已经展示给你如何通过一个简单的makefile、Autotools，甚至shell替代物，把这三个部分联系在一起。现在我们必须

把它们与Distutils联系在一起。例5-6提供一个setup.py文件，控制一个Python包的产生。

例5-6　setup.py用模板来控制一个Python包产生的过程（py/setup.py.in）

```
from distutils.core import setup, Extension

py_modules= ['pvnrt']

Emodule = Extension('pvnrt',
       libraries=['pvnrt'],          ❶
       library_dirs=['@srcdir@/..'], ❷
       sources = ['ideal.py.c'])     ❸

setup (name = 'pvnrt',               ❹
       version = '1.0',
       description = 'pressure * volume = n * R * Temperature',
       ext_modules = [Emodule])
```

❶ 源文件和链接器选项。libraries行指示会有一个-lpvnrt发送给链接器。

❷ 这一行指示出一个-L选项将被加到链接器选项，以指示它应该在哪里搜寻你的库。我们可以通过Autoconf在源路径中填充进绝对路径，就像3.3.3中“VPATH 建造”中介绍的那样。

❸ 把源文件列在这里，就像你在Automake中做的那样。

❹ 这里我们给出与被Python和Distutils使用的包相关的元数据。

Python的Distutils通过setup.py的设定来建造程序，参见例5-6，其中具有一些关于与包有关的典型模板：包的名字、版本、一个单行描述等。这里我们先讨论这三个列出的元素。

- 供宿主语言使用的包装函数的C源文件（与Autotools自身处理的库相对应）会列在一个名为sources的列表中。
- Python识别CFLAGS环境变量。Makefile变量是不会被输出给make所调用的程序的，所以在例5-6中，为py目录准备的Makefile.am，将shell变量CFLAGS赋值给Autoconf的@CFLAG@，然后才去呼叫python setup.py build。
- Python的Distutils要求你把库和库的路径隔离开。因为这些信息不经常变化，你可以手工修改库的列表，就像在例子中所看到的（不要忘记包含在主Autotools版本中产生的静态库）。由于目录结构是随不同的机器而不同的，这也是使用 Autotools协助产生AM_LDADD的原因。

我选择写一个配置包以供用户调用Autotools，然后Autotools调用Distutils。那么下一步就是让Autotools知道它要调用Distutils。

事实上，这也就是Automake在py目录中的唯一责任了，所以子目录中的Makefile.am也只处理这个问题。在例5-7中，需要一个步骤来编译包以及另一个步骤来安装，每个步骤都与一个makefile目标对应。对于配置，目标是all-local，当用户运行make的时候将会被调用；对于安装，目标是install-exec-hook，当用户运行make install的时候会被调用。

例5-7　配置Automake以驱动Python的Distutils（py/Makefile.py.am）

```
all-local: pvnrt

pvnrt:
        CFLAGS='@CFLAGS@' python setup.py build
install-exec-hook:
```

```
        python setup.py install
```

故事讲到这里，Automake已经在主目录中拥有了一切它可以生成库的信息了，Distutils也在py目录有了它需要的所有信息，而且Automake知道在正确的时刻去运行Distutils。从这里开始，用户就可以通过常用的“./configure;make; sudo make install”系列命令，来建造C库和相应的Python包装函数。

第2部分　语言

在本书的第2部分，我们将重新审视C语言的所有细节。

这将分两个部分来进行：一是整理出那些不再使用的语言细节，二是了解与新内容相关的细节。有些新内容属于语法特性，例如，能够通过名字对结构的成员列表进行初始化。有些内容则与我们经常使用的函数相关，例如，那些允许我们极为方便地写入到字符串的函数。

下面简单地列出了本书第2部分的各章所包含的材料。

第6章讨论指针有关的复杂细节（也许不是很轻松）。

从第7章开始，我们采用先破后立的方式，首先讨论在典型的教科书中所描述的哪些概念将不再被提倡，甚至认为应该被摒弃。

第8章从另外一个方向入手，对于一些教科书只是一笔带过或者根本没有提及的那些概念，我们进行了深入的讨论。

在第9章中，我们特别关注字符串，讨论怎样能够在处理字符串时不需要考虑内存分配以及烦人的字符计数细节。malloc函数将被打入冷宫，因为我们将不再调用它。

第10章讨论一些新的内容，它们允许我们在ISO标准的C中调用函数时使用任意长度的列表（例如sum(1, 2.2, [...] 39, 40)）作为输入，或

者使用命名的可选成员（例如new_person(.name="Joe", .age=32, .sex='M')）作为输入。就像摇滚乐一样，这些语法特性对我而言就像救命仙丹。如果没有它们，我很可能早就抛弃了C语言。

第11章是面向对象的解析。这是一个“九头怪”，试图把所有的东西都对应到C语言上明显没什么益处。但是当有一些方面需要的时候，还是很容易实现出来的。

你也许不相信，用一行代码，你可以加快2倍或者4倍的速度来运行你的程序（甚至更好）。秘密就在于并行线程，第12章讨论了如何把你的单线程程序变成一个多线程程序。

讨论了创建库的基本方法之后，第13章讨论怎样完成一些高级的数学计算、通过某种协议与Internet服务器进行会话、运行数据库等话题。

第6章　玩转指针

就像一首歌曲、一部好莱坞电影，指针就是一种描述其他数据的数据。就像引用的引用、别名、内存管理和malloc这一类的东西，很容易把我们搞得天旋地转。还好我们可以把指针的知识分解为独立的部分。例如，可以使用指针作为别名，却不需要使用malloc，这个函数并不像20世纪90年代的教科书常教导的那么常用。一方面，C语法中星号的用法常常令人困惑。另一方面，C语法也向我们提供了工具，用于处理那些格外复杂的指针声明，例如函数指针。

本章讲述了一些常见C语言指针的错误以及一些容易混淆的地方。如果你使用C语言很长时间了，这些问题你都很熟悉了，那么你可以快速地浏览一下本章。这一章是为了那些使用指针还有一些困难的人（这样的人非常多）准备的。

6.1　自动、静态和手工内存

C语言提供了三种基本的内存管理模式，比大多数语言要多上两种，事实上我们真正需要关注的也只是其中一种。但是在本书的后面，我还要向读者介绍两种额外的内存模型（12.3“线程本地”中介绍的thread-local和在13.2.2“针对大数据集合使用mmap”中介绍的mmap）。

自动

在第一次使用一个变量时对它进行声明，当离开自己的作用域之后变量就会被销毁。如果不使用static关键字，在函数内部所声明的所有变量都是自动变量。一般的编程语言只具有自动类型的数据。

静态

静态变量在程序的整个生命期内一直存在。数组的长度在一开始就是固定的，但它所包含的值却可以改变（因此它并不是完全静态的）。数据是在main函数启动之前被初始化的，因此所有的初始化值都必须是常量，并且不需要计算。在函数的外部（属于文件作用域）所声明的和函数内部用static关键字声明的变量都是静态变量。最后，如果忘了对一个静态变量进行初始化，它会默认初始化为全零（或NULL）。

手工

手工类型的变量涉及malloc和free函数，这也是许多段错误（segfault）的根源。这种内存模型是让许多C编程员欲哭无泪的罪魁祸首。另外，这也是唯一允许在声明之后改变数组长度的内存类型。

表6-1显示了三种内存模型的区别所在。在接下来的几章中，我们将详细讨论这些区别。

表**6-1**　三种不同类型的内存模型以及对应的特性

	静　　态	自　　动	手　　工
在启动时设置为0	◊		

受作用域限制	◊	◊	
可以在初始化时设置值	◊	◊	
可以在初始化时设置非常量值		◊	
用sizeof计算数组长度	◊	◊	
在不同的函数调用之间保持值	◊		◊
可以作为全局变量	◊		◊
在运行时设置数组长度		◊	◊
可以改变长度			◊
对程序员的折磨			◊

表中有些特性适用于变量，例如改变长度和方便的初始化。有些特性是内存系统的设计所造成的结果，例如是否可以在初始化时设置值。因此如果我们想要一个不同的特性，例如能够在运行时改变长度，就不得不关注malloc函数和指针所指向的堆。如果可以抹掉这一切重新开始，我们就不会把这三组特性与相关联的技术性细节捆绑在一起。但是已经这样了，必须面对这一切。

堆栈和堆

任何函数都在内存中占据一个空间，称为函数帧，用以保存与这个函数有关的信息，例如当函数执行完成之后返回到哪里，以及一个空间用于保存所有自动分配的变量。

当一个函数（例如main）调用另一个函数时，第一个函数的函数帧中的活动就会暂停，并且一个新的函数被添加到这个堆栈帧中。当函数执行完成时，它的帧就从这个堆栈帧中弹出。在这个过程中，保存在这个函数帧中的所有变量都会消失。

遗憾的是，堆栈的长度限制要比一般内存小得多，大约是2MB～3MB（本书写作时在Linux系统下大致如此）。这点空间对于保存莎士比亚的所有悲剧作品已经足够，因此不必担心分配一个包含10000个整数的数组会出现问题。但是，我们很可能会用到更大的数据集，因此需要使用malloc为它们在其他地方分配空间。

通过malloc所分配的内存并不位于栈中，而是在系统中称为堆的空间中。堆的大小可能有限制，也可能没有限制。在普通的PC上，可以粗略地认为堆的大小就是所有可用内存的大小。

下面是一些没有出现在C11标准中的的词语：

半导体	C++
CPU	帧
快乐	堆
爱	栈

环境和实现的细节一般都是标准以外的东西，如何实现堆栈中的帧就是这样一种实现上的细节。但是对于实现的方法，总是有一定广度的共识存在的。C标准里面定义的自动分配的变量和具体实现上在堆栈中的帧上分配并注销的那些变量是类似的。那些调用函数分配内存的描述和那些从堆中获得内存的行为也是一致的。

这些就是当我们把数据存放到内存时所浮现的相关问题。它与变量本身不同，有另外有趣的特性：

（1）如果我们在函数外部，或者用static关键字在函数内部声明一

个struct、char、int、double或其他类型变量，它就是静态的。否则它们就是动态的。

（2）如果声明了一个指针，它本身具有一种内存类型，很可能如规则1所述的属于自动或静态类型。但是，这个指针可以指向三种数据类型的任何一种，包括指向由malloc函数所分配数据的静态指针和指向静态数据的自动指针。所有的组合都是可以成立的。

规则2意味着你不能只是根据表示方式就识别出内存的模型。一方面，它带给我们的优点是，我们不需要用一个符号来表示自动分配的数组，同时又要用另一个符号表示手工分配的数组。另一方面，你要时刻记住你要处理的内存模型的类别。不要试图去改变自动数组的长度，或者忘掉释放手工申请的数组。

指针指向手工分配的内存和指针指向自动分配的内存是对初学者来说最著名的容易混淆的地方：int an_array[]和int *a_pointer到底有什么区别？

当你的程序在你的代码中遇到这句话：

```
int an_array[32];
```

你的程序会：

（1）在栈上分配出一个空间，足够容纳32个整型数；

（2）把an_array当成一个指针；

（3）然后把这个指针和新分配的地址绑定。

这个空间是自动分配出来的，这意味着你不能重新改变它的大小，或者当它脱离了范围后被自动释放以后，你还能重新得到这个空间。作为一个附加的约束，你不能让an_array指向别的地方。因为变量an_array和你分配的32个整型数空间之间是不可分割的。K&R和C标准说，an_array就是一个数组。

不管这些限制，an_array就是一个内存中的指针，通常的对指针的解引用都适用于它。（下面详细讨论。）

当程序在代码中遇到这行：

```
int *a_pointer;
```

程序会只做一件事：

把a_pointer声明为指针。

这个指针没有和任何的地址绑定，可以被重新指向任何的地方。下面的用法都合法：

```
//manually allocating a new block; pointing a_pointer to it:
a_pointer = malloc(32*sizeof(int));

//pointing the pointer to an_array, as declared above.
a_pointer = an_array;
```

所以int an_array[]和int *a_pinter是有不同的效果的。在另外的例子中，例如在typedef的声明（如一个新的结构）或者是一个函数调用中，并没有那么多的区别，如给定函数声明：

```
int f(int *a_pointer, int an_array[]);
```

a_pointer和an_array几乎有同样的行为。没有内存被分配，所以指向自动分配的内存和指向手工分配的内存的区别是没啥实际意义的。C函数收到的是输入参数的一个复制，而并不是实际的参数。所以指向自动分配内存的指针的复制没有原始指针的那些限制。C99§6.7.5.3（7）和C11§6.7.6.3（7）指出：一个参数声明为数组类型，应该被调整成经过修饰的指向这种类型的指针。（修饰符号 const, restrict, volatile, 或者 _Atomic, 在指针和数组之间进行的转化都是可以保持的。）上面的例子没有给出数组的尺寸。但是转化为指针这种行为在int g（int an_array[32]）的时候是会发生的。

我更习惯在函数声明的头文件中使用*a_piointer和typedef的形式，因为当你读复杂声明的时候，你不需要遵守从右向左这种规则了（参看8.3.1“名词-形容词方式”）。

> 自己动手：检查一些现有的代码，研究变量的分类：哪些数据位于静态内存、自动内存或手动内存，哪些变量是指向手工内存的指针、指向静态值的自动指针等。如果手头没有现成的代码，可以用例6-6作为素材完成这个练习。

6.2 持久性的状态变量

本章主要讨论自动内存、手工内存和指针的交互，对静态变量的讨论较少，但是静态变量还是很有用处的，因此值得在此花点篇幅探讨一下它们的作用。

静态变量可以具有局部作用域。也就是说，我们可以让一个静态变

量只存在于某个函数内部，但是当这个函数执行时，这个变量会保持它的值。因此它可以作为内部计数器或可复用的临时空间。由于静态变量永远不会移动，因此指向一个静态变量的指针在函数完成执行之后仍然是有效的。

例6-1展示了一个传统的教科书例子：菲波那契数列。我们把前两个成员声明为0和1，以后的每个成员都是它之前两个成员的和。

例6-1　由一个状态机所生成的菲波那契数列（fibo.c）

```
#include <stdio.h>

long long int fibonacci(){
    static long long int first = 0;
    static long long int second = 1;
    long long int out = first+second;
    first=second;
    second=out;
    return out;
}

int main(){
    for (int i=0; i< 50; i++)
        printf("%lli\n", fibonacci());
}
```

可以看到main函数何其的简单。这里的fibonacci函数是一台小型的自己运行的机器。main函数只是驱动这个函数运行，由后者不断吐出另一个值。也就是说，fibonacci函数是个简单的状态机，而静态变量正是在C中实现状态机的关键技巧。

在一个必须做到线程安全的世界里，我们应该怎样使用静态的状态机呢？ISO C委员会看到了这个问题，因此C11包含了一种_Thread_local内存类型。只要把它放在声明中：

```
static _Thread_local int counter;
```

就可以为每个线程获取一个不同的计数器。第12章中的12.3 “线程本地”这一节我将详细讨论这个问题。

声明静态变量

静态变量（即便是位于函数内部）是在程序启动之前被初始化的，此时main函数还没有启动，因此我们不能用非常量值对它们进行初始化。

```
//this fails: can't call gsl_vector_alloc() before main() starts
static gsl_vector *scratch = gsl_vector_alloc(20);
```

这确实是件麻烦的事情，但是我们可以使用下面这个宏，宏在最开始的时候初始化为0，然后在第一次使用时再为它分配值：

```
#define Staticdef(type, var, initialization) \
    static type var = 0; \
    if (!(var)) var = (initialization);

//usage:
Staticdef(gsl_vector*, scratch, gsl_vector_alloc(20));
```

只要初始化的值不为零（用指针的说法为NULL），这种方法就是有效的。如果初始化为零，它也会在下一次执行时重新进行初始化。因此不管是什么情况，这个方法都是可行的。

6.3 不使用malloc的指针

当我们告诉计算机把*A*设置给*B*时，意思可能是下面两者之一：

- 把*B*的值复制给*A*。此时用*A*++来增加*A*的值，*B*的值不会改变。

- 使*A*成为*B*的别名。于是*A*++也会增加*B*的值。

在代码中，每次表示把*A*设置为*B*时，需要明确是为了创建一份复制还是创建一个别名。这绝不是C特有的问题。

对于C而言，我们总是创建一份复制。但是，如果我们复制了数据的地址，这个指针的一份复制就成为了这个数据的别名。这是一种精巧的实现别名的方式。

其他语言具有不同的习惯：LISP系的语言非常依赖于别名，并有专门的set命令用于复制。Python对于标量一般进行复制，对于列表则执行别名操作（除非使用了copy或deepcopy）。事先知道语言代表的意义，可以避免产生大量的错误。

GNU科学库包含了vector和matrix对象，它们都具有data成员，后者本身是一个double值的数组。我们假设有一些用typedef定义的vector/matrix对，并有一个包含了这些数据对的数组：

```
typedef struct {
    gsl_vector* vector;
    gsl_matrix* matrix;
} datapair;

datapair your_data[100];
```

那么第1个矩阵的第1个成员表示如下：

```
your_data[0].matrix->data[0]
```

如果你熟悉这样的语法，就很容易接受它。但是，输入起来还是比较麻烦的。我们可以为它设置一个别名：

```
double *elmt1 = your_data[0].matrix->data;
```

在上面所显示的两种类型的赋值中，这里的等号所表示的是别名类型的赋值：只有一个指针被复制，如果我们修改了*elmt1，your_data内部被指向的数据也会被修改。

别名操作是一种与malloc无关的体验，它可以使我们获得指针操作的灵活性，同时又不必关心内存管理。

下面是没必要使用malloc函数的另一个例子。假设有一个接受一个指针变量作为输入的函数：

```
void increment(int *i){
    (*i)++;
}
```

如果函数的用户过于紧密地把指针与malloc关联在一起，可能觉得自己必须分配内存才能把指针传递给这个函数：

```
int *i = malloc(sizeof(int)); //so much effort, wasted
*i = 12;
increment(i);
...
free(i);
```

事实上，最方便的用法是通过自动分配的内存来完成任务：

```
int i=12;
increment(&i);
```

> 自己动手：正如前面我提供的建议，每次编写一行表示把*A*设置到*B*这样的代码时，都需要明确自己想要的是创建一个别名还是

> 一份复制。研究你手头的一些代码（不管是用什么语言编写的），然后逐行检查，问问自己在哪些情况下把复制替换为别名是合理的。

6.3.1 结构被复制，数组创建别名

如例6-2所示，复制一个结构的内容只需要一行操作代码就可以完成。

例6-2 不需要逐个复制结构中的每个成员（copystructs.c）

```
#include <assert.h>

typedef struct{
    int a, b;
    double c, d;
    int *efg;
} demo_s;

int main(){
    demo_s d1 = {.b=1, .c=2, .d=3, .efg=(int[]){4,5,6}};
    demo_s d2 = d1;

    d1.b=14;            ❶
    d1.c=41;
    d1.efg[0]=7;

    assert(d2.a==0);    ❷
    assert(d2.b==1);
    assert(d2.c==2);
    assert(d2.d==3);
    assert(d2.efg[0]==7);
}
```

❶ 修改*d*1，观察*d*2有没有发生变化。

❷ 这些断言都可以通过。

和以前一样，我们需要知道自己的赋值操作是创建了数据的一份复制还是在创建一个新的别名。这里是什么情况呢？我们修改了d1.b和d1.c，但d2中的b、c值并没有发生变化，因此它是创建了一份复制。但是一个指针的复制仍然指向原先的数据，因此当我们修改d1.efg[0]时，这个修改还会影响指针d2.efg的复制。我的建议是如果需要对指针内容也进行复制，就需要一个结构复制函数。如果我们并不需要追踪任何指针，那么使用复制函数就有点小题大作了，只需要使用等号就可以。

对于数组，等号将会复制一个别名，而不是数据本身。在例6-3中，我们可以进行相同的创建复制测试，即修改原先的数据，并检查复制值。

例6-3　结构被复制，但是把一个数组设置为另一个数组只是创建了一个别名（copystructs2.c）

```
#include <assert.h>

int main(){
    int abc[] = {0, 1, 2};
    int *copy = abc;

    copy[0] = 3;
    assert(abc[0]==3);  ❶
}
```

❶ 通过：当复制被修改时，原数据也被修改。

例6-4逐步显示了灾难发生的过程。两个函数自动分配两块内存：第一个函数分配了一个结构，第二个函数分配了一个较短的数组。对于自动分配的内存，我们知道在每个函数结束时，它们各自的内存块将被释放。

指定return x为返回语句的函数会把x的值返回给调用函数[C99和C11，§6.8.6.4（3）]。这看上去相当简单，但是这个值必须被复制到调用函数中，而后者的函数帧即将要被销毁。如前所述，对于结构、数值甚至是指针类型，调用函数将得到返回值的一份复制。对于数组，调用函数将得到指向这个数组的指针，而不是数组中数据的复制。

最后一种情况是个很讨厌的陷阱，因为被返回的指针可能指向一块自动分配的数组数据，而后者在函数退出时将要被销毁。返回一个指向一块可能已经被自动销毁的内存的指针是再糟糕不过的事情了。

例6-4　可以从一个函数中返回结构，但不能直接返回数组（automem.c）

```
#include <stdio.h>

typedef struct powers {
    double base, square, cube;
} powers;

powers get_power(double in){
    powers out = {.base = in,                    ❶
                  .square = in*in,
                  .cube = in*in*in};
    return out;                                  ❷
}

int *get_even(int count){
    int out[count];
    for (int i=0; i< count; i++)
        out[i] = 2*i;
    return out; //bad.                           ❸
}

int main(){
    powers threes = get_power(3);
    int *evens = get_even(3);
    printf("threes: %g\t%g\t%g\n", threes.base, threes.square, threes.cube
);
```

```
    printf("evens: %i\t%i\t%i\n", evens[0], evens[1], evens[2]); ❹
}
```

❶ 这个初始化是通过指定的初始化列表进行的。如果读者没有遇到过这种方法，等几章后再介绍。

❷ 这里是合法的。在函数退出时，会创建自动分配的out的一份复制，然后这个局部复制被销毁。

❸ 这条语句则是非法的。在这里，数组事实上是被当作指针看待的，因此在退出时，会创建指向out的指针的一份复制。但是一旦这块自动分配的内存被销毁，这个指针就指向一块坏数据。如果编译器够聪明的话，它会对这种情况提出警告。

❹ 回到调用get_even的那个函数，evens是个合法的指向int类型的指针，但它所指向的数据已经被释放。这可能会产生段错误、打印出垃圾值，在很幸运的情况下或许会打印出正确的值（像这次一样）。

如果需要对数组进行复制，我们仍然可以通过一行代码来完成，但这样就回到了内存操纵的语法，如例6-5所示。

例6-5　复制一个数组需要使用memmove，它已经过时，但是能够完成任务（memmove.c）

```
#include <assert.h>
#include <string.h> //memmove

int main(){
    int abc[] = {0, 1, 2};
    int *copy1, copy2[3];

    copy1 = abc;
    memmove(copy2, abc, sizeof(int)*3);
```

```
    abc[0] = 3;
    assert(copy1[0]==3);
    assert(copy2[0]==0);
}
```

6.3.2 malloc和内存操纵

现在回到内存部分，也就是在内存中直接处理地址。这些操作常常需要通过malloc手工分配的内存。

避免与malloc有关的缺陷的最简单方法，就是避免使用malloc。过去（20世纪80年代和90年代），我们需要使用malloc处理各种类型的字符串操作，但在第9章将详细介绍怎样在完全避免使用malloc的情况下处理字符串。我们也需要malloc处理那些必须在运行时设置长度的数组，这是一种相当常见的情况，正如本书第142页“在运行时设定数组的长度”一节所描述的那样，这也是过时的方法。

下面是我粗略整理的使用malloc的原因列表。

（1）为了改变一个现存的数组的长度，需要使用realloc，而重新分配仅对于那些一开始就是通过malloc分配的内存块才起作用。

（2）如前面所解释的那样，我们无法从函数返回一个数组。

（3）有些对象在执行初始化函数很久后仍然应该存在。不过，在第11章，我们将把这些内存管理操作包装到new/copy/free函数中，使它们不至于产生不良影响。

（4）自动内存是在栈中函数帧中分配的，它的长度限制可能只有

几兆字节（甚至更少）。因此，大块的数据（即任何以MB为单位的数据）应该在堆中分配，而不应该在堆栈中分配。另外，很可能已经有某个函数可以将数据存储到对象中，因此在实践中应该调用一个object_new函数而不是使用malloc本身对它进行操作。

（5）有时候，我们发现函数会被要求返回一个指针。例如，在多线程一节中，模板要求我们编写一个返回void *的函数。为了避开这个麻烦，我们简单地返回了NULL，但有时候会遇到无法像这样简单处理的情况。另外，注意10.10 “从函数返回多个数据项”一节讨论了从一个函数返回结构，因此我们可以发送回相对复杂的返回值，而不需要进行内存分配，这就避免了在函数内部进行内存分配的常见情况。

可见，情况实际上并没有那么多，第5种情况是极为罕见的，第4种常常是第3种的一种特殊情况，因为大的数据集一般会放在类似对象的数据结构中。在产品代码中，要尽可能少地使用malloc，就算用也使用new/copy/free函数来作为包装函数，使主代码不需要进一步处理内存管理。

6.3.3 错误来源于星号

好了，我们现在已经清楚，指针和内存分配是独立的概念，但是处理指针本身仍然可能存在问题，因为那些星号还是令人困惑的。

关于指针声明语法的设计，书面原因是为了让指针的声明形式和它的使用形式看上去相似。它们的具体含义取决于声明方式：

```
int *i;
```

由于*i是个整数，因此我们只有通过int *i把*i声明为整数才是自然的。

就是这样，如果它可以帮到你，那就太好了。我并不确信能够发明一种歧义更少的方法来完成这个任务。

在唐•诺曼的*The Design of Every day Things*一书中，始终倡导一条常见的设计规则，即“功能截然不同的事物看上去应该明显不一样”[Norman，2002]。书中提供了飞机控制的例子，两个看上去相同的控制杆常常完成截然不同的任务。在危急情况下，这可能会诱使飞行员犯错。

在这里，C的语法也存在类似的尴尬，因为在声明中的*i和声明之外的*i所表示的含义截然不同。例如：

```
int *i = malloc(sizeof(int)); //right
*i = 23;                      //right
int *i = 23;                  //wrong
```

在我的脑海中，已经抛弃了让声明和使用看上去相似的规则。下面是我所采用的规则，它很好地满足了我的需要：当用于声明时，星号表示指针。不用于声明时，星号表示指针的值。

这里有一个合法的代码片断：

```
int i = 13;
int *j = &i;
int *k = j;
*j = 12;
```

根据上面这个规则，在第2行代码中可以看到，这种初始化方式是

正确的，因为*j是个声明，因此表示指针。在第3行代码中，*k也是个指针声明，因此把它赋值给j是合理的。在最后一行，*j不是出现在声明中，因此它表示一个普通的整数，并且我们可以把12赋值给它（i也会随之被修改）。

因此下面是第一个提示：记住在声明行中看到*i时，它是个指向某对象的指针。在非声明行中看到*i时，它是指针所指向的值。

在稍后讨论一些指针运算之后，我将提供另一个技巧，它在处理奇怪的指针声明语法时会有用。

6.3.4 你需要知道的各种指针运算

数组的某个成员可以用数组的基地址加上一个偏移量来表示。我们可以声明一个指针double *p;，把它作为基地址，然后就可以像数组一样在这个基地址上使用偏移量。在基地址上，我们可以找到第1个成员p[0]的内容，在基地址上前进一步可以找到第2个成员p[1]的内容，接下来以此类推。因此，只要提供一个指针以及两个相邻成员之间的距离，就可以把它作为数组使用了。

我们可以直接采用基地址加偏移量的书面形式，类似（p+1）。正如教科书所描述的那样，p[1]等同于*（p+1），这就解释了为什么数组的第1个成员是p[0] == *（p+0）。K & R[第2版，第5.4和5.5节] 中花了6页的篇幅解释这个问题。

这个理论提示了一些规则，用于在实际应用中表述数组和它们的成员。

- 可以通过显式的指针形式double *p，或静态/自动形式double p[100]来声明数组。
- 不管是哪种情况，第*n* + 1个数组成员都是p[n]。不要忘了第一项是0而不是1，这样就可以采用特殊形式p[0] == *p。
- 如果需要第*n*个成员的地址（而不是实际值），使用&符号：&p[n]。当然，第1个成员的地址就是&p[0] == p。

例6-6展示了这些规则的一些实际应用。

例6-6　一些简单的指针运算（arithmetic.c）

```
#include <stdio.h>

int main(){
    int evens[5] = {0, 2, 4, 6, 8};
    printf("The first even number is, of course, %i\n", *evens);   ❶
    int *positive_evens = &evens[1];                                ❷
    printf("The first positive even number is %i\n", positive_evens[0]); ❸
}
```

❶ 使用特殊形式 将evens[0]写成*evens。

❷ 成员1的地址，赋值给一个新指针。

❸ 引用数组第1个成员的通常方式。

下面我再送你一个很好的技巧，这个技巧建立在指针运算规则“p+1表示数组中下一个成员的地址（&p[1]）”的基础上。根据这个规则，我们不需要在遍历数组的循环中使用下标。在例6-7中我们就使用了一个备用指针来指向list的头部，然后用p++在数组中向前遍历，直到数组尾部的NULL标记，从而获得了整个数组值。如果你查看了接下来的指针

声明的提示，会更容易理解这种用法。

例6-7　我们可以利用p++表示“前进到下一个位置”实现循环的迭代（pointer_arithmetic1.c）

```
#include <stdio.h>

int main(){
    char *list[] = {"first", "second", "third", NULL};
    for (char **p=list; *p != NULL; p++){
        printf("%s\n", p[0]);
    }
}
```

> 自己动手：如果不了解p++，你打算怎样实现这个目标？

如果目标是为了实现简洁的语法表示形式，基地址加偏移量这个技巧并不能提供太多的帮助，但它确实解释了C的许多工作原理。事实上，我们可以考虑一下使用结构，例如：

```
typedef struct{
    int a, b;
    double c, d;
} abcd_s;

abcd_s list[3];
```

若将其作为一种智力模型来分析，我们可以把list看成基地址，list[0].b与基地址的距离正好用来表示b。也就是说，假设list的位置是整数（size_t）&list，b位于（size_t）&list + sizeof（int）;，这样list[2].d的位置将是（size_t）&list + 6*sizeof（int） + 5*sizeof（double）。根据这种思路，结构就与数组非常相似了，区别是结构的成员是用名称而不是

序号表示的，并且它们具有不同的类型和长度。

这个思路并不是非常正确，因为存在对齐这个因素，系统可能会决定数据需要位于某个特定长度的内存块中，因此字段尾部可能会填充一些额外的空间，使下一个字符从正确的位置开始，并且结构的尾部可能也会进行填充，使结构列表中的每个结构能够大致对齐[C99和C11，§6.7.2.1（15）和（17）]。stddef.h头文件定义了offsetof宏，它精确地描述了基地址加领偏移量的思路：list[2].d的实际地址是（size_t）&list + 2*sizeof（abcd_s） + offsetof（abcd_s, d）。

顺便说一下，在结构的起始处不可能出现填充，因此list[2].a肯定等于（size_t）&list+ 2*sizeof（abcd_s）。

下面是个笨拙的函数，它以递归的方式对列表中的成员进行计数，直到遇到值为0的成员。假设我们想把这个函数用于零值为合理数据的任何类型的列表，因此我们让它接受一个void指针（当然这不是一种好的思路）。

```
int f(void *in){
    if (*(char*)in==0) return 1;
    else return 1 + f(&(in[1])); //This won't work.
}
```

“基地址加偏移量”的规则解释了为什么这种做法是不行的。为了表示a_list[1]，编译器需要知道a_list[0]的准确长度，这样才能知道应该从基地址偏移多少。但是，由于没有与之相关联的类型，它无法计算这个长度。

多维数组

定义多维数组的一个方法就是使用数组的数组，如int　an_array[2][3][7]。这个声明和another_array[2][3][6]声明之间有细微的差别，使用过程中带来的问题比解决的问题多，特别是当你写一个函数，用来处理以上两种类型的数组的时候。教科书一般都坚持数组就应该是固定大小的，就像一年就应该有12个月。或者不要把数组的数组传递给函数。

让我说，忘记它吧。用来处理细微的类型的差别太痛苦了，每个人都有不同的代码的世界，但是我很少在教科书以外看见这种形式，但是基地址加偏移量的方式更常见。

让我们实现一个N_1-N_2-N_3维的多维double数组：

- 定义一个结构，里面有一个数据指针（这里是data）和一个步长列表。
- 定义一个alloc函数分配内存，将指针指向这个内存data=malloc（sizeof（double）*N1*N2*N3），并且记录步长S1=N1, S2=N2, S3=N3。你同时需要一个Free函数来释放掉你申请的内存。
- 定义get/set函数：get（x,y,z）用data[x+S1*y+S1*S2*z]来获取数据，set在同样的位置写入数据。利用这些get和set函数，data中S_1数据块有（*x*,0,0）的形式。下一块数据，从S1+0到S1+S1有（*x*,1,0）的形式。执行这种一行接一行的模式来覆盖所有形如（*x*,*y*,0）的值，共需要S_1*S_2个槽，下一个槽的位置为（0,0,1），直到所有的S_1*S_2*S_3个值都被考虑到。

我们可以检查get/set函数中输入的那些值是否在数组的范围之外，因为我们已经记录了步长，不需要记录S3的步长，但是如果你想检查边界条件，你应该进行记录。

GNU科学库有一个二维数组的这种实现。实现有点不一样，包括对第一维度的步长以及一个偏移量的指示。得到基于行列的vector的子集或者通过改变起始点和步长来获得子数组都是正常操作。对于一个三维或更多维的数组，你的搜索引擎也许会给你一些其他的就像我上面描述的基于开始点和步长的实现。

6.3.5　将typedef作为一种教学工具

任何时候当我们遇到一种复杂的类型时，类似于指向某种类型的指针的指针的指针等情况，可以考虑用typedef进行简化。

例如，下面这个常见的定义有效地减少了字符串数组的视觉混乱，

使它们的意图变得清晰。

```
typedef char* string;
```

在前面的指针运算p++例子中，char *list[]这样的声明是否很清楚地告诉你它表示一个字符串列表，而*p是一个字符串？例6-8对例6-7的for循环进行了重写，用string替换了char *。

例6-8　添加一个typedef声明使笨拙的代码变得清晰（pointer_arithmetic2.c）

```
#include <stdio.h>
typedef char* string;

int main(){
    string list[] = {"first", "second", "third", NULL};
    for (string *p=list; *p != NULL; p++){
        printf("%s\n", *p);
    }
}
```

list的声明行现在变得简单，很清晰地表示它是个字符串列表，并且string *p也很清晰地表示p是个指向字符串的指针。因此，*p表示一个字符串。

最后，我们仍然需要记住字符串是个指向字符的指针。例如，NULL是个合法的字符串值。

我们甚至可以更进一步，例如，使用上面的typedef加上typedef stringlist string*声明一个字符串的二维数组。这种方法有时候非常实用，但有时候只会增加记忆的负担。

从概念上讲，函数类型的语法实际上是指向一个特定类型的函数的指针。如果我们有一个下面这样的函数头：

```
double a_fn(int, int); //a declaration
```

然后只要添加一个星号（并加上括号以保证优先级），就可以描述一个指向这种类型的函数的指针：

```
double (*a_fn_type)(int, int); //a type: pointer-to-function
```

然后在前面加上typedef来定义一种类型：

```
typedef double (*a_fn_type)(int, int); //a typedef for a pointer to functi
on
```

现在我们可以把它当作一种类型使用，例如，声明一个接受另一个函数作为其输入参数的函数时，可以这样做：

```
double apply_a_fn(a_fn_type f, int first_in, int second_in){
    return f(first_in, second_in);
}
```

通过对函数指针类型的重新定义，那些接受其他函数作为输入的函数定义就变得简单了。如果没有这种方法，那些复杂星号的书写曾是令人生畏的。

最后需要说明的是，指针实际上要比教科书所描述的简单得多，因为它实际上只是一个位置或别名，根本不需要涉及不同类型的内存管理。像指向字符串的指针的指针这样的复杂构造总是会让人感到迷惑，但这只不过是因为我们以狩猎为生的祖先从来没有见到过这玩意而已。至少，C提供了typedef这个工具来处理它们。

第7章　教科书不应该再过多介绍的C语言语法

C语言也许挺简单，但是C标准有700页，所以，如果你不想花费毕生精力去研究它，那么你应该知道哪些部分可以被忽略。

让我们从二合字母或者三合字母开始，如果你的键盘缺少{}键，你可以用<%和%>来替代，就像是int main()<%...%>。这在20世纪90年代还是有用的，因为那个时候世界上的键盘有不同的样式，但是今天你很难找到没有{ }的键盘了。三合字母类似于??<和??>，这个东西更没用，以至于gcc和clang的作者都没有花任何时间去编码解析它们。

像三合字母这种，语言中过时的东西很容易就被忽略掉了，因为根本没有人提到它们。但是语言的其他部分在教科书里面却被反复地提及，只是用来满足旧的C89规范的要求，或者为了适应20世纪90年代计算机硬件的限制。限制越少，我们写代码的效率就越高，如果你能从删除代码并去掉那些没用的冗余中获得快乐，那么本章适合你。

7.1　不需要明确地从main函数返回

我们首先去除一行几乎在每个程序中都会遇到的代码，并以此热身。

所有程序必须有一个main函数，它的返回类型是int，因此在程序中

必须要有下面这条代码：

```
int main(){ ... }
```

读者会觉得其中必然有一条return语句，表示main函数所返回的整数值。但是，C标准知道这个返回值极少使用，所以不想麻烦我们。C标准要求：“……到达结束main函数的}之前返回一个0值。”[C99和C11，§5.1.2.2（3）]。也就是说，如果我们在程序最后一行没有编写return 0;，C会默认加上它。

想起来了吗？当你运行完你的程序后，你可以使用echo$?来了解这个程序的返回值；你可以用它来查看你的main函数是否运行完毕，并返回了零值。

本书的最开始已经向读者展示了这个版本的hello.c程序，可以看到其中只包括了一条#include语句和一行代码[1]：

```
#include <stdio.h>
int main(){ printf("Hello, world.\n"); }
```

> 自己动手：检查自己的程序，从main函数中删除return 0这一行，观察这样做会不会有区别。

7.2 让声明的位置更灵活

让我们回想一下以前阅读剧本时的情况。在剧本的一开始是人物表，里面列出了剧中出场的所有人物。但在开始阅读剧本之前，一串人

名列表对于我们而言并没有太大的意义。因此，我们大多会跳过这一部分，直接阅读正文的内容。当我们沉浸在情节中并忘了Benvolio是谁时，可以很方便地返回到剧本的最前面，看看人物表中对他的简单介绍（哦，他是Romeo的朋友，Montague的侄子）。但是，前提是我们所阅读的是纸版的剧本。如果我们是在屏幕上阅读剧本，就必须搜索Benvolio最早出现的地方。

简而言之，人物表对于读者而言并不是非常有用的。在人物第一次出场时再介绍他们显然更为合适。

我经常看到类似下面这样的代码：

```
#include <stdio.h>

int main(){
    char *head;
    int i;
    double ratio, denom;

    denom=7;
    head = "There is a cycle to things divided by seven.";
    printf("%s\n", head);
    for (i=1; i<= 6; i++){
        ratio = i/denom;
        printf("%g\n", ratio);
    }
}
```

首先是3～4行的介绍性材料（要看是不是算上空行），然后才是实质性的程序。

这是ANSI C89的复古风格，它要求所有的声明都出现在代码块的头部，这样做的原因是早期编译器的技术限制。现在，我们仍然必须声明所有的变量，但可以尽量减轻作者和读者的负担，在第一次使用变量

时才声明它们：

```
#include <stdio.h>

int main(){
    double denom = 7;
    char *head = "There is a cycle to things divided by seven.";
    printf("%s\n", head);
    for (int i=1; i<= 6; i++){
        double ratio = i/denom;
        printf("%g\n", ratio);
    }
}
```

在上述代码中，声明恰好出现在需要的位置，因此声明的责任降低到最低限度，相当于在第一次使用变量前加上它的类型名。如果使用了彩色语法高亮显示，这种声明仍然很容易被发现（如果你的编辑器不支持彩色，最好还是换一种支持它的，这类编辑器数不胜数）。

在阅读不熟悉的代码时，我看到一个变量的第一直觉就是回过头去看看它是在什么地方声明的。如果它是在第一次使用时或者在第一次使用之前的那一行被声明的，我就可以省掉这几秒浏览的时间。另外，根据应该使变量的作用域尽可能小的规则，我们要想方设法降低活动变量对前面代码的依赖，这对于较长的函数而言是非常重要的。并且，还有一点好处，在使用我们在第12章介绍的OpenMP进行并行处理的时候，在循环内部声明会产生便利。

在上面这个例子中，声明出现在它们各自代码块的起始处，然后是非声明性的代码行。这正是这个例子所显示的结果，我们可以自由地混合声明行和非声明行。

我把denom的声明放在函数的头部，但我们也可以把它移动到循环

的内部（因为它只是在循环内部使用）。我们可以信任编译器能够充分地理解，而不会浪费时间和精力在每次循环迭代时对这个变量进行销毁和重新分配［尽管在理论上会这样，参见C99和C11，6.8（3）］。站在索引的角度，它应该和循环同时结束。因此，把它的作用域限制在循环之内是非常自然的做法。

> **这种新语法会拖慢程序吗？**
>
> 不会。
>
> 编译器所执行的第1个步骤是把代码解析为独立于语言的内部表示形式。这样gcc（GNU编译器集合）在解析步骤的最后才能够产生C、C++、ADA和FORTRAN的可兼容目标文件，它们看上去都是相同的。因此，C99为了阅读方便而在语法上所提供的便利一般在可执行文件被生成之前就已经被抽掉了。
>
> 同时，运行程序的目标设备所看到的只是经过编译的机器指令，因此不管源代码所遵循的是C89、C99还是C11标准，对它来说是没有区别的。

在运行时设置数组的长度

除了把声明放在任意位置外，你也可以在运行时分配数组，并根据声明之前的计算结果来确定它的长度。

同样，这个方法在以前是不允许的。25年前，我们要么必须在编译时知道数组的长度，要么使用malloc。

例如，假设想创建一组线程，但线程的数量是用户通过命令行设置的。旧式的方法就是通过atoi（argv[1]）（也就是把第1个命令行参数转换为一个整数）从用户那里获取数组的长度。接着，在运行时确定了长度之后，就分配一个具有正确长度的数组。

```
pthread_t *threads;
int thread_count;
thread_count = atoi(argv[1]);
threads = malloc(thread_count * sizeof(pthread_t));
...
free(threads);
```

我们可以写得更紧凑些：

```
int thread_count = atoi(argv[1]);
pthread_t threads[thread_count];
...
```

后面的写法由于行数少，所以更不容易出错，它看上去像是声明一个数组，而不是对内存寄存器进行初始化。我们必须释放手工分配的数组，但可以简单地把自动分配的数组放下不管，它在程序离开特定的作用域后会被自动清理[2]。

7.3 减少类型转换

在20世纪七八十年代，malloc返回的是一个char*指针，我们必须对它的结果进行类型转换（除非是为字符串分配内存），其形式类似：

```
//don't bother with this sort of redundancy:
double* list = (double*) malloc(list_length * sizeof(double));
```

现在不需要这样做了，因为malloc向我们返回的是一个void指针，编译器可以很方便地将它自动转换为任何类型。最简单的转换方式是声明一个具有正确类型的新变量。例如，必须接受一个void指针为参数的函数，一般可以用下面这样的形式开始：

```
int use_parameters(void *params_in){
    param_struct *params = params_in; //Effectively casting pointer-to-NULL
```

```
    ...                                 //to a pointer-to-param_struct.
}
```

在更普遍的情况下，如果把一种类型的数据项赋值给另一种类型的变量是合法的，即使我们没有指定显式的类型转换，C也会为我们完成这个任务。如果它对于给定的类型是不合法的，就必须编写一个函数设法完成转换。这对于C++而言并不正确，后者对类型更为依赖，因此需要显式地指定每个类型转换。

另外还有两个原因支持使用C的类型转换语法把一个变量从一种类型转换为另一种类型。

首先，当两个数相除时，一个整数除以另一个整数总是返回一个整数，因此下面这两条语句都是正确的：

```
4/2 == 2
3/2 == 1
```

第2个除法是许多错误的来源。这个错误很容易修正：如果i是个整数，则i + 0.0就是与这个整数匹配的浮点数。不要忘了括号，这样就简单地解决了问题。如果常量2是个整数，那么2.0或简单的2.就是浮点数。因此，下面这些变型都是可行的：

```
int two=2;
3/(two+0.0) == 1.5
3/(2+0.0) == 1.5
3/2.0 == 1.5
3/2. == 1.5
```

我们也可以使用类型转换的形式：

```
3/(double)two == 1.5
3/(double)2 == 1.5
```

站在美学的角度，我倾向于加零的形式。当然，读者也可以选用转换为double的形式。但是每次在使用“/”操作符时都要养成其中一种习惯，因为这是许多错误的来源（并不仅仅限于C，许多其他语言也坚持int/int的结果应该是int类型，而不会自动予以修正）。

其次，数组的索引必须是整数。这是规定[C99和C11，§6.5.2.1（1）]，如果发送了一个浮点数的索引，gcc就会报错。因此，我们可能必须去转换，即使我们知道在当前情况下实际提供的总是一个整数值的表达式。

```
4/(double)2 == 2.0     //This is floating-point, not an int.
mylist[4/(double)2]    //So, an error: floating-point index.

mylist[(int)(4/(double)2)] //Works. Take care with the parens.

int index=4/(double)2      //This form also works, and is more legible.
mylist[index]
```

从上面可以看到，尽管存在少数合理的理由需要类型转换，但我们也可以选择避免使用类型转换的语法：加上0.0或为数组索引声明一个整数变量。

这不仅仅是为了减少书写混乱的问题。你的编译器为你进行类型检查并相应地发出一些警告，但是一个显式的转换就是对编译器说：别管我，我知道我正在做什么！例如，考虑下面的程序，试图设置list[7]=12，但是犯了两次典型的错误，应该用一个指针指向的值，但是这里却直接使用了指针的值。

```
int main(){
    double x = 7;
    double *xp = &x;
    int list[100];

    int val2 = xp;       //Clang warns about using a pointer as an int.
    list[val2] = 12;

    list[(int)xp] = 12; //Clang gives no warning.
}
```

7.4 枚举和字符串

枚举的出发点是好的，但是它走上了歧路。

它的优点是足够清楚：整数值并不容易记忆，因此当我们在代码中使用一个简短的整数列表时，最好对它们进行命名。下面是一种甚至更糟的方法，在不使用enum关键字的情况下用#define进行定义：

```
#define NORTH 0
#define SOUTH 1
#define EAST 2
#define WEST 3
```

使用enum，我们可以把上面这4行代码缩减为1行，并且调试器更容易知道EAST的含义。下面是对#define序列的改进：

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

但是，现在全局空间中有了5个新符号：directions、NORTH、SOUTH、EAST和WEST。

为了让枚举发挥作用，它一般必须是全局的（在一个将被整个项目中多次包含的一个头文件中声明）。例如，我们常常在函数库的公共头

文件中看到枚举类型的typedef声明。创建全局变量会产生相应的责任。为了尽可能地减少名字空间的冲突，函数库的作者会使用像G_CONVERT_ERROR_NOT_ABSOLUTE_PATH这样的名字，或者是相对简洁的CblasConjTrans。

此刻，单纯而感性的想法就此破灭。我不想手工输入这些紊乱的名字，对它们的使用频率之少使我在每次使用之前都必须进行查阅（尤其是其中有许多是不常用的错误值或输入标志，因此要相隔相当长的一段时间才会再次使用）。另外，全大写的写法看上去像有人在向你吼叫。

我个人的习惯是使用单个字符，用't'来标记转置，用'p'来表示路径错误。我觉得这已经具备足够的提醒作用。事实上，我觉得'p'的方案在拼写上也比那些采用全大写的方案容易记忆得多，并且它不需要在名字空间中添加新项。

在提到老生常弹的效率问题之前，记住枚举一般是个整数，而char事实上只是C对于单字节的说法。因此，对枚举进行比较时，很可能需要比较16个状态位甚至更多。但是如果使用了char，只需要比较8位。因此，即使在速度问题相当重要的情况下，使用单个char的方案也胜于枚举。

我们有时候需要对标志进行组合。当我们使用系统调用open打开一个文件时，我们可能需要发送O_RDWR|O_CREAT，这是两个枚举的逐位组合。我们很可能不会非常频繁地直接使用open，而是使用POSIX的fopen，后者更为友好。它并没有使用枚举，而是使用了一个单字母或双字母的字符串（例如"r"或"r+"）来表示某个文件是可读的、可写的、同时可读写的等。

在这个上下文环境中，我们知道"r"表示读取。即使没有记住这些约定，在使用过几次fopen之后，我们也可以对这些标记了如指掌。反之，对于CblasTrans或CblasTranspose，我每次在使用之前都必须进行查阅。

枚举当然也有优点，它表示一个小型的固定符号集，因此如果我们误输入了其中一个，编译器就会停止，并迫使我们修正打字错误。对于字符串，在运行时之前是无法知道输入错误的。但反过来，字符串并不是小型的固定符号集，因此我们可以更轻松地对枚举集进行扩展。例如，我曾经看到过一个错误处理程序，将它自身提供给其他系统所使用，前提是新系统所产生的错误要与原系统的枚举所包含的错误匹配。如果这些错误是短字符串，其他人对此进行扩展也是很简单的。

有一些理由支持使用枚举：有时候我们有一个数组，没理由把它作为结构，但是它又需要使用命名元素；或者在完成内核层次的工作时，为位模式提供名字具有重要意义。但是，当枚举用于表示一个简短的选项列表或者一个简短的错误代码列表时，单字符或者短字符串同样可以完成任务，同时又不至于扰乱名字空间和用户的记忆。

7.5 标签、goto、switch和break

在过去的岁月里，汇编代码并不拥有现代的while和for循环这样的奢侈工具。反之，它只能使用条件、标签和跳转。我们现在所编写的while （a[i] < 100） i++;，以前的汇编代码可能要写成下面这样：

```
label 1
if a[i] >= 100
    go to label 2
```

```
increment i
go to label 1
label 2
```

如果你要花一分钟的时间才能明白这段代码将要做些什么，可以想像一下在现实世界的情况中，循环需要被穿插、嵌套或者涉及其他跳转的半嵌套。我可以根据自己悲哀而痛苦的经历证明试图追踪和理解这样的代码基本上是行不通的，而且goto语句在如今被认为是有害的［Dijkstra，1968］。

可以想象，对于那些整日编写汇编代码的人们而言，C所提供的while关键字会受到怎么样的欢迎。但是C有一个子集仍然建立在标签和跳转的基础之上，包括标签、goto、switch、case、break和continue的语法。我个人觉得这是C中的一个过渡部分，供那些习惯了汇编代码的开发人员使用，从而帮助他们转移到更为现代的风格。本节将在这个基础上讨论这些特性，并指出它们在什么时候仍然是有必要的。但是，C的这个子集并非绝对必要，因为我们可以用该语言的其他特性编写具有相同功能的代码。

7.5.1 探讨goto

通过提供一个名字和冒号，可以为一行C代码加上标签。接着，我们可以通过goto跳转到这行代码。例7-1是个简单的函数，它用一行加上outro标签的代码描述了这个思路。函数会对两个数组中所有元素的求和，前提是它们都不是NaN（不是一个数，参见第7章中的“用NaN标记异常的数值”一节）。如果其中一个元素为NaN，就会认为发生了错误，需要退出这个函数。但是不管我们怎样选择退出，都需要释放这两个向量以完成清理工作。我们可以让清理代码在代码清单中出现3次

（一次是vector中出现NaN元素时，一次是vector2中出现NaN元素时，另一次是正常退出时），但更清晰的做法是建立一个退出点，并在需要时跳转到那里。

例7-1　使用goto优雅地处理了错误情况下的资源释放

```
/* Sum to the first NaN in the vector.
  Sets error to zero on a clean summation, 1 if a NaN is hit.*/
double sum_to_first_nan(double* vector, int vector_size,
double* vector2, int vector2_size, int *error){
double sum=0;
    *error=1;
for (int i=0; i< vector_size; i++){
if (isnan(vector[i])) goto outro;
        sum += vector[i];
    }
for (int i=0; i< vector2_size; i++){
if (isnan(vector2[i])) goto outro;
        sum += vector2[i];
    }
    *error=0;

    outro:
    printf("The sum until the first NaN (if any) was %g\n", sum);
    free(vector);
    free(vector2);
return sum;
}
```

goto语句只能用于一个函数的内部。如果需要从一个函数跳转到另一个完全不同的函数，可以在C标准库文档中查阅longjmp。

单跳转本身还是相对容易使用的，如果适量使用并不会影响代码的可读性。即使是Linus Torvalds，Linux内核的领导作者，也推荐有限地使用goto，例如，在出现错误或处理完成时提前退出一个函数，正如这个例子中所做的一样。同样，如第12章所述，当你利用OpenMp的时候，你会发现在并行代码的中间不允许你使用return语句。为了停止运

行，你不得不需要很多if语句，或者一个goto语句直接跳到代码的结束部分。

因此，我们可以对goto的传统认知进行一些修正，它一般来说是有害的，但在有些情况下是可行的，比如在出现多种类型的错误而需要执行清理工作时，而且它往往比其他替代方案更为清晰。

针对病态问题的一个关键词

当有问题发生时，一个函数要结束运行了，这个时候goto语句在执行一些清理工作的时候是非常有用的。从全局来说，你可以使用三个退出函数：exit, quick_exit和_exit。你可以使用at_exit和at_quick_exit函数来注册一些其他的清理过程。

程序开始的时候，你可以调用at_exit（fn）来注册fn，这样exit函数在关闭流和结束引用前，会首先调用你注册的fn。例如，如果你打开了一个数据库的句柄，或者需要关掉一个网络的链接，或者需要XML文档关掉所有打开的元素，你可以把这些操作放到这样一个函数里。它必须有void fn（void）的形式。任何传递给这个函数的信息必须是全局变量。当你注册的函数被调用后（遵循后入先出的顺序），打来的流和文件被关闭，程序结束执行。

你可以把整个的一个函数的集合通过at_quick_exit来注册。这些函数（并不是你通过at_exit来注册的那些）当你的程序调用quick_exit的时候被调用。这种退出并不关闭流，也不清空缓冲区。

最后，_exit函数用最快的方式来离开；没有任何注册的函数被调用，也没有任何缓冲区被清空。

例7-2演示了一个简单的例子，当你注释掉不同的没有返回值的函数，程序会打印出不同的内容。

例7-2　放弃希望，利用_Noreturn函数限定符来标识输入的函数

```
#include <stdio.h>

#include <unistd.h> //sleep

#include <stdlib.h> //exit, _Exit, et al.
```

```
void wail(){
    fprintf(stderr, "OOOOooooooo.\n");
}

void on_death(){
    for (int i=0; i<4; i++)
        fprintf(stderr, "I'm dead.\n");
}

_Noreturn void the_count(){    ❶
    for (int i=5; i --> 0;){
        printf("%i\n", i); sleep(1);
    }
    //quick_exit(1);           ❷
    //_Exit(1);
    exit(1);
}
int main(){
    at_quick_exit(wail);
    atexit(wail);

    atexit(on_death);
    the_count();
}
```

❶ _Noreturn关键字告诉编译器，不要为我准备函数返回的那些信息。

❷ 把那些注释去掉，看看哪些函数被调用了。

7.5.2 switch

下面是一段取自教科书的代码，代码中使用了C标准的get_opt函数来解析命令行参数：

```
char c;
while ((c = getopt(...))){
    switch(c){
       case 'v':
            verbose++;
            break;
       case 'w':
            weighting_function();
            break;
       case 'f':
            fun_function();
            break;
    }
}
```

当c == 'v'时，显示的详细程度就会增加。当c == 'w'时，权重函数会被调用等。

注意break语句（只是跳出switch语句而不是跳出while循环，while循环仍然继续）的大量出现。switch函数只是跳转到适当的标签（记住冒号指定了一个标签），然后程序流继续执行，就像以其他方式跳转到一个标签一样。因此，如果verbose++后面没有break语句，则程序将会欢快地继续执行weighting_function函数，接下来也是如此。这个特性被称为贯穿（fall-through）。虽然有些原因支持实际使用贯穿行为，但对我而言，用swith-case来减少对标签、goto和break的使用就像榨干了水份的柠檬一样乏味。Peter Van Der Linden 调查发现，在大型的代码工程中，贯穿只有3%的可能性是对的。

如果你很不喜欢由于忘记break和default从而带来了的bug， 我有一个更简单的建议：不要使用switch。

取而代之，我们可以使用一组简单的if和else语句：

```
char c;
while ((c = getopt(...))){
    if (c == 'v') verbose++;
    else if (c == 'w') weighting_function();
    else if (c == 'f') fun_function();
}
```

上面的代码还是存在冗余，因为存在对c的重复引用，但它的长度更短，因为我们不需要每三行使用一条break语句。由于它并不是原始的标签和跳转的简单包装器，因此很难出现错误。

7.6 被摒弃的float

浮点数的数学运算所面临的挑战往往令人吃惊。我们很容易编写一个合理的算法，它在每个步骤所出现的误差只有0.01%，但是经过1000次迭代之后，结果就面目全非了。我们可以找到大量关于如何避免这类令人吃惊的误差的建议。有许多建议现在仍然有效，但这个问题现在很容易处理：使用double代替float。对于计算过程中的中间值，只要使用long double就不会出问题。

例如，Writing Scientific Software建议用户避免使用计算方差的single pass算法 [Oliveira，2006；第24页]。它们提供了一个条件不良的例子。读者可能知道，浮点数之所以如此命名是因为这种类型的数据是按照小数点左右浮动来缩放数据的。为了便于阐述，我们假设计算机就

是按照小数点模式操作浮点数的。因此，这种类型的系统在存储23000000时就像存储.23或.00023一样方便，只要浮动小数点就可以了。但是23000000.00023就存在困难了，因为所拥有的位仅能表示小数点之前的数据，如例7-3所示。

例7-3　float不能存储太多的有效位 （floatfail.c）

```
#include <stdio.h>

int main(){
    printf("%f\n", (float)333334126.98);
    printf("%f\n", (float)333334125.31);
}
```

例7-3在我的上网本（使用32位的float）上所产生的输出如下。

```
333334112.000000
333334112.000000
```

这种精度显然无法让人满意。这也解释了为什么以前的计算机书籍特别关注算法的编写，就是为了尽可能地减少这种类型的误差，它只保证7位可靠的数字。

32位的float就是如此，这也是最低的标准了。我甚至不得不显式地转换为float，不然系统会把这些数值存储为64位的值。

64位对于存储15位数字是足够了：100000000000001不再成为问题（可以尝试一下！提示：printf（%.20g, val）把val打印为20位的数字）。

例7-4展示了运行Oliveira和Stewart例子的代码，包括平均值和方差

的一种single pass算法。同样，这些代码只适用于演示，因为GSL已经实现了平均值和方差计算。这个例子运行了两次：一次是病态数据的版本，给出了和作者在2006年得到的一样的可怕的结果；另一次是在每个数减去34120之后，这样即使是普通的float也能得到完整的精度。因为所提供的不是病态的数据，所以其结果就是精确的。

例7-4　病态的数据：不再成为大问题（stddev.c）

```
#include <math.h>
#include <stdio.h> //size_t

typedef struct meanvar {double mean, var;} meanvar;

meanvar mean_and_var(const double *data){
    long double avg = 0,                                    ❶
          avg2 = 0;
    long double ratio;
    size_t cnt= 0;
    for(size_t i=0; !isnan(data[i]); i++){
        ratio = cnt/(cnt+1.0);
        cnt  ++;
        avg  *= ratio;
        avg2 *= ratio;
        avg  += data[i]/(cnt +0.0);
        avg2 += pow(data[i], 2)/(cnt +0.0);
    }
    return (meanvar){.mean = avg,                           ❷
                     .var = avg2 - pow(avg, 2)}; //E[x^2] - E^2[x]
}

int main(){
    double d[] = { 34124.75, 34124.48,
                   34124.90, 34125.31,
                   34125.05, 34124.98, NAN};

    meanvar mv = mean_and_var(d);
    printf("mean: %.10g var: %.10g\n", mv.mean, mv.var*6/5.);

    double d2[] = { 4.75, 4.48,
                    4.90, 5.31,
                    5.05, 4.98, NAN};
```

```
    mv = mean_and_var(d2);
    mv.var *= 6./5;                                          ❸
    printf("mean: %.10g var: %.10g\n", mv.mean, mv.var);  ❹
}
```

❶ 根据经验，使用更高层次的精度表示中间值可以避免累积的四舍五入问题。也就是说，如果我们的输出是double类型，则avg、avg2和ratio应该是long double。如果我们都使用double，这个例子的结果会不会发生变化？（提示：不会。）

❷ 这个函数将返回一个指定初始化值的结构。如果读者还不熟悉这种形式，很快就会在后面看到它。

❸ 上面这个函数计算总体方差（population variance），通过缩放生成样本方差。

❹ 这里使用了%g作为printf系函数的格式指示符，这是一种通用格式，既接受float也接受double。

下面是这个程序运行的结果：

```
mean: 34124.91167 var: 0.07901676614
mean: 4.911666667 var: 0.07901666667
```

因为原始的数值调整的关系，第二个平均数位移了34120，但精度却是差不多的（如果我们允许，.66666可以一直延续下去），病态数据的方差的误差是0.000125%。病态数据并没有产生明显的影响。

亲爱的读者，这就是技术上的进步。当我们为问题留出两倍的空间时，所有原先需要考虑的问题突然之间都不存在了。我们仍然可以构建出数值漂移可能会导致问题的现实场景，但现在难度要大得多。虽然两

个分别采用全double和全float所编写的程序在速度上存在可察觉的差别，但多花几毫秒的时间就可以避免所有头疼的问题还是相当合算的。

那么，我们是不是应该在所有使用整数的场合都使用long int呢？答案并不是那么直截了当。π的double表示形式肯定要比π的float表示形式要精确得多，但对于int和long int而言，在几十亿之内，它们所表示的数据是相同的。唯一可能存在的问题是溢出。对于short而言，这个问题可以说是声名狼籍的，例如，32000就会导致溢出。但在当前这个时代，一般系统上的整数范围大约是±21亿，一般来说不会出现问题。但是，如果你考虑得比较久远，认为有可能几个数相乘达到十亿级以上的规模（例如200×200×100×500），那么就需要使用long int甚至long long int，不然其结果就不是不精确的问题，而是完全错误，因为C会从+21亿突然回滚到−21亿。可以查看limits.h（通常位于/include或/usr/include/目录下）了解相关细节。例如，在我的上网本中，limits.h表示int和long int是相同的。

如果你正在做一些特别严肃的计算，那么#include <stdint.h>并且使用intmax_t类型，它保证至少有$2^{63}-1=9223372036854775807$的范围。

如果要进行切换，还要记住在所有的printf调用中为long int使用%li，如果使用intmax_t，则使用%ji。

7.7 比较无符号整型数

例7-5演示了如何比较int和size_t的一个简单的程序，size_t是一个无符号整型数，有的时候用来代表数组的偏移量（通常，sizeof也返回这种类型）：

例7-5　比较无符号和有符号整数（unit.c）

```
#include <stdio.h>

int main(){
    int neg = -2;
    size_t zero = 0;
    if (neg < zero) printf("Yes, -2 is less than 0.\n");
    else            printf("No, -2 is not less than 0.\n");
}
```

你可以运行这个程序，并验证这个程序得到了一个错误的结果。这个程序片段演示了当有符号和无符号整型数比较的时候，C会强迫有符号书转变成无符号数，这一点并不是我们所期望的。我必须承认我也吃过几次这样的亏，而且这种bug非常难以定位，因为这种比较看起来非常自然。

C给你不同的方法来表示一个数，从无符号短整型到long double。有这么多的类型是因为当初计算机的内存是以KB来衡量的。但是现在，本章和上一章都不建议把数据类型都快使用到它能表达的极限了。细微管理类型的做法，例如，用float来提高效率，并在特定的条件下使用double；或者因为你不会使用负数，所以你用无符号整型数；这些做法都不是推荐的做法，因为这些做法或者带来细微的误差，或者在C中并不那么自然的代数转换中带来bug。

7.8　安全的将字符串解析成数字

有几个函数，可以把字符串解析成对应的数字。最流行的办法就是atoi和atof，它们的用法非常简单，如下：

```
char twelve[] = "12";
```

```
int x = atoi(twelve);

char million[] = "1e6";
double m = atof(million);
```

但是这里缺乏错误检查，如果twelve是“XII”, 那么atoi（twelve）返回0，并且程序继续运行。

一个更安全的方法是strtol和strtod。它们在C89中出现，经常被人们忽略，只是因为它们并没有出现在*K&R*的第一版里。并且也需要多花一点工作来使用它们。很多我调查过的作者（包括我的第一版书）都没有提到过它们，或者把它们放到附录中。

Strtod函数接受第二个参数，一个指向char的指针，这个指针将指向那个第一个不能解析成数字的字母。这就可以用来解析剩下来的的文本，或者你希望文本中只包含数字，那么这个返回值也能告诉你是不是真的这样。如果这个变量被声明为char *end*, 那么在读入的字符串能够完全转换为数字的时候，*end*应该指向的是*'\0'*，所以我们用一个条件来判定它，如*if*（end）printf（"readfailure"）。

例7-6给出一个例子，计算在命令行给出的数值的平方。

例7-6　使用strtod读入一个数字（strtod.c）

```
#include "stopif.h"
#include <stdlib.h>  //strtod
#include <math.h>    //pow

int main(int argc, char **argv){
    Stopif (argc < 2, return 1, "Give me a number on the command line to s
quare.");
    char *end;
    double in = strtod(argv[1], &end);
    Stopif(*end, return 2, "I couldn't parse '%s' to a number. "
```

```
                "I had trouble with '%s'.", argv[1], end);
    printf("The square of %s is %g\n", argv[1], pow(in, 2));
}
```

C99以后，也有了Strtof和strtold函数用来进行float和long double的转换。整型数的版本strtol和strtoll，转换long int和long long int。这些函数接受三个参数：要转换的字符串、指向end的指针和转换基数。传统的转换基数是10，但是你也可以设置为2用来读入二进制，8用来读入八进制，16用来读入十六进制，甚至设置成36。

用NaN标记异常数值

在IEEE的浮点数标准中提供了一些精确的规则来表示浮点数据，包括表示infinity、-infinity和NaN（用于提示像0/0或log（-1）这样的数学错误）这些特殊形式。IEEE 754/IEC 60559（它是这个标准的名称，处理这些事务的人们喜欢在标准后面加上一些数字）并不是C或POSIX标准的组成部分，但它几乎得到了所有环境的支持。如果读者是为Cray大型机或者一些特殊用途的嵌入式设备编写代码，那你可以忽略本节所讨论的细节。（但是即使Arduino单片机的libc库也定义了NAN和INFINITY。）

参见例10-1，NaN可以作为一种非常实用的标记，表示到达了列表的末尾。前提是我们能够保证这个列表的正常元素都不是NaN值。

另外，我们需要知道对NaN执行相等性测试总会失败，甚至NaN==NaN的结果也是false。只能使用isnan（*x*）来测试*x*是否为NaN。

一些需要深入研究数值数据的读者可能会对使用NaN作为标记的其他方式感兴趣。

IEEE标准有很多不同形式的NaN：符号位可以是0或者1，指数可以是全1，剩余部分是非零，因此我们可以得到像下面这样的一串位S11111111 MMMMMMMMMMMMMMMMMMMMMMM，其中S是符号位，M是未指定的尾数。

零作为尾数表示正负无限，具体取决于符号位，但我们也可以把这些M指定为我们想要的任何东西。一旦采用了一种方式控制这些自由位之后，就可以向一个数值网格的单元格添加各种类型的不同标志。

一个产生特定的NAN的优雅的方法是使用函数nan（tagp），返回一个带有“Tagp指定的内容”的NAN[C99和C11, §7.12.11.2]，输入可以是字符串代表的的浮点数，nan函数是一个strtod的包装函数，会写到NaN的尾数中。

例7-7的小程序生成并使用了一个NA（不可用）标记，它在要区分数据是丢失还是出现了数学错误的场景下非常实用。

例7-7　用自己的NaN标记来注释自己的浮点数据（na.c）

```
#include <stdio.h>
#include <math.h> //NAN, isnan, nan


double ref;


double set_na(){
    if (!ref) ref=nan("21");
    return ref;
}


int is_na(double in){                ❶
    if (!ref) return 0; //set_na was never called==>no NAs yet.


    char *cc = (char *)(&in);
    char *cr = (char *)(&ref);
    for (int i=0; i< sizeof(double); i++)
        if (cc[i] != cr[i]) return 0;
    return 1;
}


int main(){
    double x = set_na();
```

```
    double y = x;
    printf("Is x=set_na() NA? %i\n", is_na(x));
    printf("Is x=set_na() NAN? %i\n", isnan(x));
    printf("Is y=x NA? %i\n", is_na(y));
    printf("Is 0/0 NA? %i\n", is_na(0/0.));
    printf("Is 8 NA? %i\n", is_na(8));
}
```

❶ is_na函数检查我们所测试的数的位模式是否与set_na所创建的特定位模式匹配。这通过把两个输入都作为字符串并执行逐字符的比较来实现。

我生成了1个信号并存储在一个数值数据点中，使用21作为被选择的键。我们只要对前面的代码进行少量的修改就可以把十几个不同的标记直接插入数据集。

事实上，有些广泛使用的系统（例如WebKit）走得更远，不仅仅是使用一个信号，而是在它们的NaN的尾数中实际插入一个完整的指针。这个方法称为NaN的装箱，留给读者作为习题。

---

[1] 顺便说一句，这段代码比起旧规范而言，还省略了4个字母。在*K&R*第2版中指出，如果你在括号里面没有包含任何内容，如int main()（这是一种旧式的声明），就代表了参数没有指定任何信息，并不是代表没有参数。所以在旧的规则里，我们需要写成int main(void)来显式地告诉大家，main函数不接受参数。但是自从1999年，函数声明里面的空的括号代表这个函数没有参数。[C99，§6.7.5.3 和C11§，6.7.6.3（14）]

[2] C99标准要求编译器接受可变长数组（VLAs）。C11标准做了一些让步，认为这种接受是可选的。个人认为，这个决定非常不符合C标准委员会的一贯的风格。它们为了能让旧的代码（即使是三合字母）在新

的编译器上编译，所做的任何决定都是非常小心和谨慎的。

因为VLA现在是可选的了，我们必须知道它现在是否被广泛支持。编译器的作者会让自己的编译器最大可能地可以处理现存的代码，所以目前支持C11标准的大的编译器都接受VLA。即使你对一个Arduino单片机编程（它并不是传统的堆栈结构），你可以使用AVR-gcc，它是gcc的一个支持VLA的变体。我认为VLA在很多平台上还是支持的，同时我希望在未来它也被支持。

对于那些对标准支持不友好的编译器，读者最好使用测试宏特性来检测编译器是否支持VLA，细节可以参考本书第166页8.1.2 “检测宏”一节。

第8章　那些C语言教科书经常不讲解的语法

上一章讲述了传统的C语言教科书强调的一些主题，但是这些主题已经不适合现在的编程环境了。本章我讲解一下很多教科书没有或只是涉及了一点的主题。就像上一章一样，本章覆盖了很多小的主题，但是集中在三个主要的部分：

预处理只会被简单提到，我想很多人认为它只是一个辅助的工具，而不是真正的C语言。但是存在就有它的合理性。宏可以做一些事情，但是C语言的其他部分却不能做。并不是所有的兼容标准的编译器提供同样的功能，预处理器也能帮助我们确定开发环境的一些特点，并以此做出必要的反应。

在我对C语言教科书的研究中，我发现了没有提到过static和extern关键词的一两本书。所以本章我们讨论一下链接部分，然后细致分析一下static令人迷惑的用法。

const关键字对于本章而言是非常适合的，因为它的方便和实用令人难以割舍。但是，无论在C语言标准中，还是在常见编译器中的实现中，它都存在一些怪异的地方。

8.1　营造健壮和繁盛的宏

我们必须知道如何躲避一些通常的陷阱，但是如果你能提供一个避免了一些常见错误的宏，你就有了一个安全的用户界面。在第10章中，我们将讨论一些使库的接口更为友好、更不容易出错的方法。这些方法在很大程度上依赖宏来实现。

我看到过很多人的说法，认为宏本身就很容易产生错误，应该避免使用。但是他们并没有建议不要使用NULL、isalpha、isfinite、assert以及像log、sin、cos、pow等类型通用的数学工具，也没有建议不要使用其他的GNU标准库使用宏所提供的许多其他工具。这些精心编写的、健壮的宏每次都能精确地完成自己的任务。

宏用于执行文本替换（由于替换文本一般比宏更长，所以这种文本替换也称为宏的展开），但文本替换的思路与普通的函数并不一样，因为它的输入可能与宏中的文本或源代码中的其他内容发生交互。宏最好在我们需要这种交互，而又不必想方设法加以防范的场合中使用。

有三条规则可以使宏更为健壮，但在讨论它们之前，我们首先区分两种不同类型的宏。一种类型的宏展开之后是个表达式，意味着可以对这类宏进行求值或者输出它们的值。如果表达式的值是数值，还可以把这种类型的宏放在等式中。另一种类型的宏是一段指令，它们可能出现在一条if语句之后或者一个while循环之中。下面是宏的一些使用规则。

- 括号！如果宏只是进行简单的文本粘贴，很容易导致出乎意料的结果。下面是个简单的例子：

```
#define double(x) 2*x      Needs more parens.
```

现在，用户的意思是把（1+1）*8扩大一倍，这个宏的展开结果是

2*1+1*8，其值等于10而不是32。需要加上括号才能让它正确地发挥作用：

```
#define double(x) (2*(x))
```

现在，（2*（1+1））*8的结果正如预期的那样。一条基本原则是把所有的输入放在括号内，除非有特定的理由不使用括号。如果有一个表达式类型的宏，把这个宏表达式本身放在括号中。

- 避免重复作用。下面这个取自教科书的例子存在一些风险：

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

如果用户尝试int x=1, y=2; int m=max（x, y++），期望m变成2（y增值之前的值），并且y的值增加到3。但是，这个宏的展开结果是：

```
m = ((x) > (y++) ? (x) : (y++))
```

这样y++就被求值了2次，导致用户期望只进行一次增值的地方实际进行了两次增值，结果m=3，不是用户所预期的m=2。

如果有个代码块类型的宏，就可以声明一个变量，在这个代码块的头部接受输入值，然后在宏的剩余部分使用输入值的这份复制。

这个规则并不像括号规则那样严格，经常出现的max宏就是一例。因此需要记住，作为宏的用户，应该尽量避免在宏中进行会产生副作用的调用。

- 代码块两端要加上花括号。下面是一个简单的代码块宏：

```
#define doubleincrement(a, b) \    Needs curly braces.
    (a)++;                    \
    (b)++;
```

如果把它放在一条if语句的后面，它的行为可能是不正确的：

```
int x=1, y=0;
if (x>y)
    doubleincrement(x, y);
```

增加一些缩进，可以使这个错误变得更加明显，展开结果是：

```
int x=1, y=0;
if (x>y)
    (x)++;
(y)++;
```

另一个潜在的陷阱是：如果在宏里声明了一个变量total，但用户已经定义了一个total变量会怎么样呢？在一个代码块中声明的变量可能与这个代码块外部所声明的变量冲突。例8-1为上述这两个问题提供了一个简单的解决方案：在宏的两边加上花括号。

在整个宏的两边加上花括号使宏的内部所声明的total变量的作用域仅限于宏两边的花括号之间，因此不会与main函数所声明的total变量发生冲突。

例8-1　我们可以用花括号控制变量的作用域，就像典型的非宏代码一样（curly.c）

```
#include <stdio.h>

#define sum(max, out) {          \
    int total=0;                 \
    for (int i=0; i<= max; i++)  \
        total += i;              \
```

```
    out = total;                  \
}

int main(){
    int out;
    int total = 5;
    sum(5, out);
    printf("out= %i original total=%i\n", out, total);
}
```

但是还有点小问题，回到我们doubleincrement宏的定义，这个代码：

```
#define doubleincrement(a, b) { \
    (a)++;                      \
    (b)++;                      \
}

if (a>b) doubleincrement(a, b);
else     return 0;
```

展开成：

```
if (a>b) {
    (a)++;
    (b)++;
};
else     return 0;
```

else前面多余的分号让编译器很迷惑。用户会得到编译错误，这代表他们不能发布他们的代码。但是删掉分号或者再多加一对大括号的解决方法都不是直接的解决方法，也不符合UI最好不透明的要求。告诉你真相吧，也没啥你能做的。最通常的解决方法就是把宏包含在一个只运行一次的do-while循环里。

```
#define doubleincrement(a, b) do { \
    (a)++;                         \
    (b)++;                         \
```

```
} while(0)

if (a>b) doubleincrement(a, b);
else     return 0;
```

这样，问题解决了，我们有了用户可以不知道的宏。但是如果你有一个宏，内部包含一个break语句或者其他用户提供的break语句呢？下面是一个验证宏，这种用法就不工作了。

```
#define AnAssert(expression, action) do { \
    if (!(expression)) action;            \
} while(0)

double an_array[100];
double total=0;
…
for (int i=0; i< 100; i++){
    AnAssert(!(isnan(an_array[i])), break);
    total += an_array[i];
}
```

用户并没有意识到提供的break被嵌入到宏里面的do-while循环里了，所以很多的编译器可能运行不正确的代码。本例中do-while包装会破坏break的期望的行为。这种情况下你就不能用do-while包装，只能警告用户注意else前面有一个分号这种错误[1]。

使用gcc -E curly.c，可以看到预处理器把sum宏展开为下面所显示的形式，两边的花括号保证了宏作用域中的total绝不会与main作用域中的total发生冲突。因此，下面的代码将把total打印为5：

```
int main(){
    int out;
    int total = 5;
    { int total=0; for (int i=0; i<= 5; i++) total += i; out = total; };
    printf("out= %i total=%i\n", out, total);
}
```

警告

用花括号限制宏的作用域并不能防止所有的名字冲突。在前一个例子中，int out, i=5; sum（i, out）;这条语句会产生什么结果呢？

如果你有一些工作不正常的宏，对于gcc、Clang和icc，使用-E只运行预处理器，向stdout输出所有预处理指令的展开版本。由于其中包含了#include<stdio.h>以及其他大量引用的展开形式，因此我一般采用类似gcc -E *mycode*.c|less这样的格式，把结果重定向到一个文件或页面，然后从中搜索想要调试的宏的展开形式。

上面就是使用宏时需要警惕的地方。保持宏的简单性这个基本原则仍然是合理的，我们将会发现产品代码中的宏倾向于单行的代码，采用某种方式对输入进行预处理，然后调用一个标准函数完成真正的工作。调试器以及不能解析宏定义的非C语言系统本身并不能访问宏，因此我们所编写的东西在无法使用宏时仍然应该有办法可以使用。8.2“static和extern链接”一节提供了一个建议，在编写简单函数时减少这种麻烦。

8.1.1 预处理器技巧

为预处理器所保留的标记是#，预处理器对这个标记有3种截然不同的用法：标记一个指令，输入的字符串化，以及把符号连接起来。

我们知道，像#define这样的预处理器指令在行首是以#开始的。

另外，#之前的空白被忽略[*K&R*第2版，§ A12，第228页]。因此我们的第1个提示就是：可以把随用随弃的宏放在函数的中间，就在它们被使用之前，并采用和函数一样的缩进格式。根据过去的原则，把宏直接放在它们需要被使用的地方违背了程序的“正确”组织（原来的方法是把所有的宏放在文件的头部）。但是，把宏放在这个位置使它便于被引用，清晰地凸显了它的随用随弃的本质。在OpenMp部分，我们将for循环用#pragmas标记，把#放到左边边界的地方会让代码不方便读。

#的下一个用法是在宏中：它把输入参数转换为一个字符串。

例8-2显示了一个sizeof的用法的程序，不过它的主要着眼点还是预处理器宏的使用。

例8-2　既被输出又被求值的文本（sizesof.c）

```
#include <stdio.h>

#define Peval(cmd) printf(#cmd ": %g\n", cmd);

int main(){
    double *plist = (double[]){1, 2, 3};      ❶
    double list[] = {1, 2, 3};
    Peval(sizeof(plist)/(sizeof(double)+0.0));
    Peval(sizeof(list)/(sizeof(double)+0.0));
}
```

❶ 这里是一个复合常量。如果读者对此不熟悉，我将在以后介绍它们。考虑sizeof操作符如何处理plist时，要记住plist是个指向数组的指针，而不是数组本身。

在尝试这种用法时，可以看到宏的输入被打印为普通的文本，然后打印出它的值，因为#cmd是把cmd当作一个字符串。因此

Peval（list[0]）的展开结果是：

```
printf("list[0]" ": %g\n", list[0]);
```

是不是觉得2个字符串"list[0]" "：%g\n"并排放在一起有点不太对劲？下一个预处理器技巧是如果两个字符串文本是相邻的，预处理器会把它们合并为一个："list[0]：%g\n"。而且不仅仅处理宏中的字符串：

```
printf("You can use the preprocessor's string "
       "concatenation to break long strings of text "
       "in your program. I think this is easier than "
       "using backslashes, but be careful with spacing.");
```

sizeof的限制

有没有尝试过示例代码？它基于一种被广泛使用的技巧，就是把数组的总长度除以一个元素的长度来获取一个自动或静态数组的元素个数（http://c-faq.com/aryptr/ arraynels.html；另见*K&R*第1版第126页以及第2版第135页）。例如：

```
//This is not reliable:

#define arraysize(list) sizeof(list)/sizeof(list[0])
```

sizeof操作符（它是C语言的一个关键字，不是普通的函数）所引用的是自动分配的变量（可能是个数组或指针），而不是一个指针可能指向的数据。对于一个像double list[100]这样的自动数组，编译器必须分配100个double类型的空间，并确保有足够的空间（很可能是800字节），不至于与堆栈中后面的变量重叠。对于手工分配的内存（double *plist; plist = malloc（sizeof（double *100））;），堆栈中的指针的长度可能是8字节（当然不是100），sizeof将返回这个指针的长度，而不是它所指向的数据的长度。

当你指向一个玩具时，有些猫会扑上去翻弄这个玩具，另一些猫却会来闻你的手指。

相反，我们可能想把两个并非字符串的东西连接在一起。我们可以使用2个#：##。如果name变量的值是LL，name##_list的效果就是

LL_list，这是个合法并且可以使用的变量名。

我希望每个数组都有一个辅助变量标识它的长度。例8-3编写了一个宏，它为这个宏所关注的每个列表声明了一个以_len结尾的局部变量，甚至确保每个列表具有一个终止标记，这样就不需要知道它的长度。

这个宏有点小题大做了，因此我不推荐直接使用它，但是它演示了我们怎样根据自己所选择的命名模式生成大量的小型临时变量。

例8-3　使用预处理器创建辅助变量（preprocess.c）

```
#include <stdio.h>
#include <math.h> //NAN

#define Setup_list(name, ...)                                    \
    double *name ## _list = (double []){__VA_ARGS__, NAN}; \❶
    int name ## _len = 0;                                        \
    for (name ## _len =0;                                        \
            !isnan(name ## _list[name ## _len]);                 \
            ) name ## _len ++;

int main(){
    Setup_list(items, 1, 2, 4, 8);                               ❷
    double sum=0;
    for (double *ptr= items_list; !isnan(*ptr); ptr++)           ❸
        sum += *ptr;
    printf("total for items list: %g\n", sum);

    #define Length(in) in ## _len                                ❹

    sum=0;
    Setup_list(next_set, -1, 2.2, 4.8, 0.1);
    for (int i=0; i < Length(next_set); i++)                     ❺
        sum += next_set_list[i];
    printf("total for next set list: %g\n", sum);
}
```

❶ 左侧演示了使用##根据给定的模板产生一个变量名。右侧是第10章将要介绍的变长宏。

❷ 生成items_len和items_list。

❸ 这是个使用NaN标记的循环。

❹ 有些系统允许我们使用一种类似这样的格式查询一个数组的自身长度。

❺ 这是个使用了next_set_len长度变量的循环。

作为一种风格，过去存在一个习惯，通过把一个函数名的所有字母都大写来表示这个函数实际上是个宏，借以提醒宏的文本替换可能造成的问题。我觉得这有点大惊小怪了。我更倾向于只是把宏名的第1个字母大写，也有一些人根本不去关心大小写的问题。

宏参数是可选的

下面是一个合理的断言类型的宏，它在断言失败时将会返回：

```
#define Testclaim(assertion, returnval) if (!(assertion))  \
          {fprintf(stderr, #assertion " failed to be true. \
          Returning " #returnval "\n"); return returnval;}
```

用法示例：

```
int do_things(){
    int x, y;
    …
```

```
    Testclaim(x==y, -1);
    …
    return 0;
}
```

但是，如果有个函数并没有返回值该怎么办呢？在这种情况下，我们可以把第2个参数留空：

```
void do_other_things(){
    int x, y;
    …
    Testclaim(x==y, );
    …
    return;
}
```

这个宏的最后一行的展开结果是return;，这对于C而言是合法的，适用于返回类型为void的函数[2]。

如果有兴趣，甚至可以使用下面这种方法来实现默认值：

```
#define Blankcheck(a) {int aval = (#a[0]=='\0') ? 2 : (a+0); \
        printf("I understand your input to be %i.\n", aval); \
        }

//Usage:

Blankcheck(0); //will set aval to zero.
Blankcheck( ); //will set aval to two.
```

8.1.2 测试宏

能够运行C语言程序的硬件五花八门，从LinuxPC到Arduino微控制器到GE冰箱。你的C代码通过测试宏来发现你的编译器和目标平台的能力，你可以通过编译命令的-D选项来传入，或者用#include包含列举了本平台下的一些能力的文件。如POSIX系统的unistd.h文件或者是Windows系统中的windows.h文件。

一旦你知道哪些宏可以被测试，你就可以用预编译器来处理不同的环境了。

Gcc和glang通过-E -dM选项（-E：只是运行预编译；-dM：输出宏的值）给出一系列定义好的宏，在我工作的平台上：

```
echo "" | clang -dM -E -xc -
```

产生了157个宏。

不太可能写下一个完整特性宏的列表，有些是为硬件定义的，有些是为C标准库定义的，还有为编译器定义的，但是表8-1列举出了一些常见并且稳定的宏以及它们的含义。我选择这些宏是因为这些宏与本书相关，那些以__STDC_开头的是C语言标准定义的。

表**8-1** 一些常见的宏及特性

宏	含义
_*POSIX*_CSOURCE	遵循IEEE 1003.1，也就是ISO/IEC 9945。通常设置为版本日期

_WINDOWS	一个Windows盒子，有一个*windows.h*头文件以及相关的各种定义
__MACOSX__	运行OS X的Mac
__STDC_HOSTED__	程序被编译成与这样的电脑兼容，电脑本身的操作系统会去调用main
__STDC_IEC_559__	遵守IEEE754，浮点标准最后变成了ISO/IEC/IEEE60559。需要注意的是，处理器可以表示NaN, INFINITY 和-INFINITY
__STDC_VERSION__	编译器实现的标准的版本：很多用199409L代表C89（1995年的固定版本），199901L代表C99，201112L代表C11
__STDC_NO_ATOMICS__	如果实现不支持_Atomic变量，而且不提供*stdatomic.h*，就把它设置为1
__STDC_NO_COMPLEX__	如果实现不支持复数类型，把它设置为1
__STDC_NO_VLA__	如果实现不支持变长数组，把它设置为1
__STDC_NO_THREADS__	如果实现不支持C标准*thread.h*以及其中定义的各种元素，把它设置为1。你也许可以用POSIX线程，OpenMP fork或者其他的替代物

Autoconf的一个核心功能就是产生宏以描述这些能力。假设我们使用Autoconf，你的config.ac文件包含这样一行宏：

```
AC_CHECK_FUNCS([strcasecmp asprintf])
```

如果你运行./configure的系统有（POSIX-标准）strcasecmp，但是没有（GNU/BSD-标准）asprintf， 那么Autoconf将会产生一个名字为config.h的头文件，包含下面这两行：

```
#define HAVE_STRCASECMP 1
/* #undef HAVE_ASPRINTF */
```

这样你就可以利用#ifdef（如果定义）或者#ifndef（如果没有定义）预处理指令，像这样：

```
#include "config.h"

#ifndef HAVE_ASPRINTF
[paste the source code for asprintf (Example 9-3) here.]
#endif
```

如果有的时候如果一个feature丢失了，那么你不能做其他事，只能停止下来，如果是这样，你就用#error预处理指令。

```
#ifndef HAVE_ASPRINTF
    #error "HAVE_ASPRINTF undefined. I simply refuse to " \
           "compile on a system without asprintf."
#endif
```

自从C11以后，也有了_Static_assert关键字，一个接受两个参数的静态校验：一个要被检验的静态表达式，以及一个传递给人的信息。一个C11兼容的assert.h头文件定义了一个在书写上更好的static_assert来扩展_Static_assert关键字。例子如下：

```
#include <limits.h> //INT_MAX
#include <assert.h>

_Static_assert(INT_MAX < 33000L, "Your compiler uses very short integers."
);

#ifndef HAVE_ASPRINTF
static_assert(0, "HAVE_ASPRINTF undefined. I still refuse to "
                 "compile on a system without asprintf.");
#endif
```

在33000L结尾处的字母L和一些上面的年月值代表给定的值应该被当成long int来读入，以防止正常的int发生溢出。

也许有比#if/#error/#end更方便的形式，但是在2011年12月出版的新标准中才被引入，所以本身就是一个移植性的问题。例如，Visual Studio 实现的设计者实现了_STATIC_ASSERT宏，它只接受一个参数，而且也不认识标准的_Static_assert[3]。

另外，#ifdef/#error/#endif和_Static_assert很大程度上是相等的：C标准指出它们都检查一个常量表达式，然后输出一个文本字符串，虽然一个是在预处理阶段，而一个是在编译阶段。考虑到现在这个时候，当有丢失的特性的时候，也许坚持用预处理来停止会更安全一些。

8.1.3 避免头文件重复包含

如果你把同一个结构定义粘贴到一个文件中，会发生什么？例如你可以把：

```
typedef struct {
    int a;
    double b;
} ab_s;
```

```
typedef struct {
    int a;
    double b;
} ab_s;
```

放到一个header.h的文件中。

一个人会轻松地认出这两个结构是相同的，但是编译器却被要求从文件中读取任何新的结构的类型，所以上诉的代码不会被编译，因为ab_s被两个独立（虽然是相同的）的类型重复声明[4]。

就算我们只声明一次，有的时候也会产生这种重复声明的错误，例如，如果你像这样两次包含头文件：

```
#include "header.h"
#include "header.h"
```

因为头文件通常包含其他的头文件，这种错误在你一长串头文件包含链中会出现。C标准为了解决这种问题，采用了一种叫作“包含保卫”的方法来预防这种错误，在这种方法中，我们需要定义一个专属于这个文件的变量，然后把这个文件的内容包裹起来。

```
#ifndef Already_included_head_h
#define Already_included_head_h 1

[paste all of header.h here]

#endif
```

第一次的时候，变量没有被定义，文件被解析。第二次的时候，变量被定义了，所以文件剩下的部分就忽略了。

这种方法自从K&R的第 2 版的时候就在用，不过现在你可以简单地

使用pragma once，在需要包含的头文件前面加上：

```
#pragma once
```

编译器就会理解这个文件不会发生二次包含。Pragma是编译器相关的，只有几个被定义在C语言标准中。但是，每一个主流的编译器，包括gcc、clang、intel和c89模式的Visual Studio以及其他一些编译器，全部理解#pragma once。

使用预处理器注释代码

#if 0和#endif之间的代码将会被忽略，所以你可以用这种方法来注释掉你的代码。与传统的/*..*/不同，这种方法可以嵌套。

```
#if 0
    ...
    #if 0
        /* code that was already ignored */
    #endif
    ...
#endif
```

但是如果嵌套不正确，如：

```
#if 0
    ...
    #ifdef This_line_has_no_matching_endif
    ...
#endif
```

因为预处理器把#endif和错误的#if配对，所以错误就会发生。

8.2 static和extern链接

在本节中，我们编写代码告诉编译器应该向链接器提供不同类型的选项。编译器在某个时刻处理one.c文件，（一般情况下）会在某个时刻产生一个one.o文件，然后链接器把那些.o文件链接在一起，生成一个库文件或可执行文件。

如果在两个不同的文件中存在变量x的2个声明，会出现什么情况呢？很可能其中一个文件的作者并不知道另一个文件的作者也选择了x这个名称，因此这两个x应该被分隔到两个不同的空间。也有可能这两位作者都注意到了它们引用了同一个变量，连接器应该把所有对x的引用都指向内存中的同一个地址。

外部链接意味着不同文件中匹配的符号应该被连接器看成同一样东西。使用extern关键字代表这是一个外部链接。

内部链接表示一个文件中的一个变量x或一个函数f()的实例属于它自身，只与相同作用域中的x或f()的实例匹配（在任何函数外面所声明的所有东西都属于文件作用域）。使用static关键字表示内部链接。

有趣的是，外部链接具有（可选的）extern关键字，但内部链接的关键字并不是看上去极为合理的intern，而是static。在第6章6.1 “自动、静态和手工内存”这一节中，我讨论了3种类型的内存模型：静态、自动和手工。出于技术原因，用static关键字既表示一种链接属性又表示一种内存模型，这两个概念可能在过去的某个阶段存在概念上的重叠，但现在却区分得很明显。

- 对于具有文件作用域的变量，static只影响链接属性。

  ——默认链接属性为外部，因此使用static关键字将之改变为内部链接。

  ——属于文件作用域的任何变量将通过静态内存模型分配内存，不管所使用的是static int x、extern int x还是普通的int x。
- 对于代码块作用域的变量，static只影响内存模型。

  ——默认的链接属性是内部，因此static关键字并不会影响链接属性。我们可以通过把变量声明为extern来改变它的链接属性。不过这种方法很少用。

  ——默认内存模型是自动，因此static关键字会把内存模型更改为静态模型。
- 对于函数，static只影响链接属性。

  ——函数只能在文件作用域中定义（gcc提供了嵌套函数作为一种扩展）。和具有文件作用域的变量一样，它们的默认链接属性为外部，但使用static关键字可以表示内部链接。

  ——它不会与内存模型产生混淆，因为函数总是静态的，就像具有文件作用域的变量一样。

声明一个函数使之能够被不同的.c文件共享的法则是把这个函数的头部放在一个.h文件，并在项目中全程包含这个头文件。然后把这个函数本身放在一个.c文件中（它将具有默认的外部链接属性）。这是个良好的法则，值得我们遵循，但是把一些单行或两行的工具函数（例如max和min）放在一个被广泛包含的.h文件中也是一种常见的做法。为此，我们可以在函数声明之前加上static关键字，例如：

```
//In common_fns.h:
static long double max(long double a, long double b){
```

```
    (a > b) ? a : b;
}
```

当我们在十余个文件中的每个文件中都#include "common_fns.h"时，编译器将为每个文件产生max函数的一个新实例。但是，由于我们已经假设这个函数具有内部链接属性，没有任何一个文件会公开函数名max，因此这个函数的十几个不同实例将会独立存在，并不会发生冲突。这种重新声明可能会在最终的可执行文件中额外添加几字节，并且增加几毫秒的编译时间，不过这在一般的环境中是无伤大雅的。

只在头文件中声明外部链接的元素

extern关键字比static关键字更简单，因为它只关乎链接，和内存模型没啥关系。有一些典型的方法建立具有外部链接的变量：

- 在一个被多处包含的头文件中，用extern关键字将你的变量声明为extern，例如extern int x。
- 在一个.c文件中，像以往那样声明变量，可以有选择的初始化。例如int x=3。就像所有的静态内存变量一样，如果你不初始化（只有int x），变量初始化为0或者是NULL。

这就是你使用外部链接变量的方法。

你也许更倾向于不把extern声明放到头文件中，只是松散地放到你的代码文件中。在file1.c文件中，你声明了int x，你意识到你需要在file2.c文件中存取它，所以你把extern int x放到这个文件的头部，目前，这也可以工作。下个月，当你在file1.c文件中改成double x。编译器的类

型检查还会发现file2.c内部是一致的，链接器会将file2.c中的变量链接到double类型的x变量存储的地方，然后这个变量会误被读成int。如果你把所有的extern声明都放到头文件中，就可以避免这个问题。如果类型在某个地方改变了，编译器会捕捉到这个不一致。

系统在内部做了很多工作，确保你可以多次声明变量，但是只为一次声明分配内存。通常，有extern的声明就是声明（一个类型信息的语句，所以编译器能进行一致性检查），不是定义（用来分配和初始化内存的指令）。但是一个不用extern关键字进行的声明是一个暂时的定义：如果一个编译器到了一个单元的结尾（下面定义）也没看到定义，那么这个暂时的定义就会变成一个定义，并被初始化为0或者NULL。当#include的文件内容被加入以后的.c文件时，标准把单元定义为一个文件。

Gcc和clang编译器读入单元时，就好像它在读入整个的的程序，这意味着如果在一个程序中，存在几个non-extern声明并且没有定义，这会把所有的暂时定义转化为一个真正的定义。即使你用--pedantic选项，gcc也并不关心你是否使用extern关键词。实际上，这意味着extern很大程度上可选的；你的编译器会读入很多int x=3的声明，然后当成一个有外部链接的单个变量的单个声明。从技术上来说这不是标准的。但是K&R把这种行为描述为“在UNIX系统中是正常的，并且可以当成ANSI89 标准的一个通用扩展”。Harbison, 1991文档说明了4个关于extern规则的不同解释。

这意味着如果两个文件中的两个变量有共同的名字，但是你想它们应该彼此独立的。这个时候如果你忘记static关键字，编译器可以把有外

部链接变量链接成一个变量。这种细微的bug非常容易发生，所以对于那些文件范围的有内部链接的变量不要忘了使用static。

8.3 const关键字

const关键字是极为常用的，但是，与const有关的规则存在一些令人惊奇的地方，并存在不一致之处。本节将指出这些问题，使读者不再对它们感到惊讶，并且能够按照建议的方式以良好的风格使用const。

我们在很早之前就知道，传递给函数的是输入数据的一份复制，但是我们可以通过向函数传递指向输入数据的指针，允许函数修改输入数据。当我们看到函数的输入是普通的非指针数据时，就明白调用者所拥有的源变量并不会被修改。当我们看到一个指针输入时，情况就不太明晰了。列表和字符串自然都是指针，因此指针输入有可能是需要被修改的数据，也有可能仅仅是字符串。

const关键字是一种能够让代码更容易阅读的修饰手法。它是一种类型限定符，表示输入指针所指向的数据在整个函数的执行过程中不会被修改。知道数据不应该被修改是个非常实用的信息，因此要尽可能地使用这个关键字。

第一个警告：编译器并不会锁定被指向的数据，以避免所有类型的修改。一个加上了const标记的名字所表示的数据可能通过另一个名字被修改。在例8-4中，a和b指向同一个数据，但是由于在set_elmt的头文件中，a并没有使用const标记，因此仍然可以修改b数组的一个元素，参见图8-1。

例8-4　在一个名字中用const标注的数据可以通过另一个名字被修改（constchange.c）

```
void set_elmt(int *a, int const *b){
    a[0] = 3;
}

int main(){
    int a[10] = {};      ❶
    int const *b = a;
    set_elmt(a, b);
}                        ❷
```

❶ 把数组初始化为全零。

❷ 这是一个什么也不做的程序，其目的只是在没有错误的情况下通过编译并运行。如果我们想要证明b[0]确实发生了变化，可以在调试器中运行这个程序，并在最后一行代码中设置断点，然后打印b的值。

因此，const只是一种修饰手法，而不是施加于数据上的一把锁。

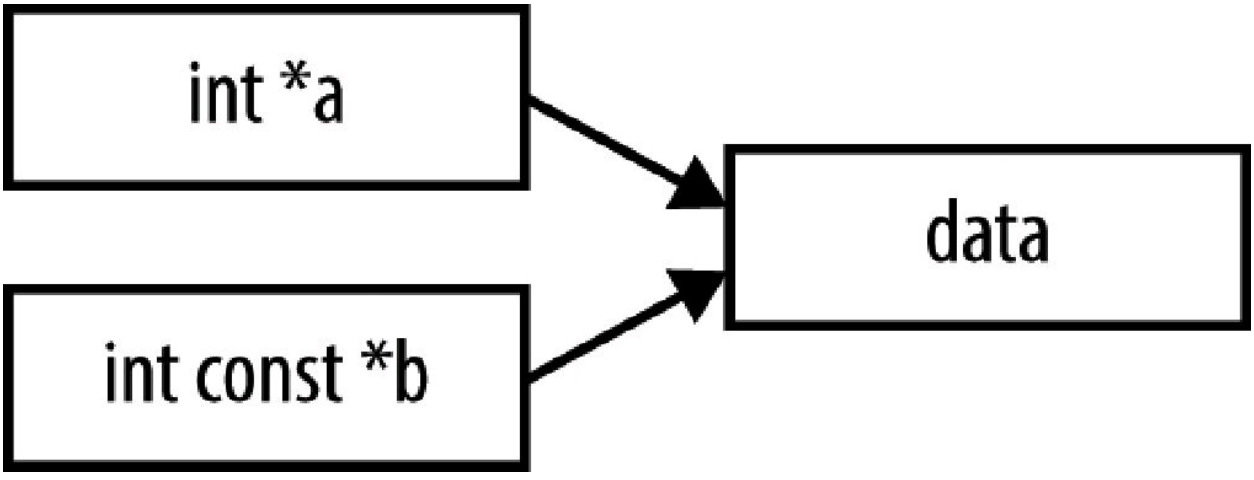


图8-1　我们可以通过a修改数据，即使b使用了const。这是合法的操作

8.3.1　名词-形容词形式

理解声明的技巧在于从右向左阅读。因此：

- int const = 一个常量整数。
- int const * = 一个（变量）指针指向一个常量整数。
- int * const = 一个常量指针指向一个（变量）整数。
- int * const * = 一个指向常量指针的指针，常量指针指向一个整数。
- int const * * = 一个指向指针的指针，指向常量整数。
- int const *const * =一个指向常量指针的指针，常量指针指向常量整数。

我们可以看到const总是作用于它左边的文本，就像*一样。

我们可以切换类型名和const的位置，写成int const或const int（尽管在涉及const和*的情况下无法采用这种切换方法）。我更倾向于int const形式，因为它与更复杂的结构保持一致，并遵循了从右向左的原则。const int也是一种使用习惯，也许是因为它更接近英语的阅读习惯，也许是因为人们已经习惯了。这两种形式都是可行的。

> **关于restrict和inline**
>
> 我所编写的有些示例代码使用了restrict和inline关键字，有些则没有使用它们。因此，我可以演示这两个关键字所带来的速度差别。
>
> 我过去曾对restrict所带来的好处寄予厚望， 但是当我如今编写测试程序时，使用和不使用这两个关键字所带来的速度差别是微不足道的。
>
> 作为贯穿全书所提供的建议，我在编译时设置CFLAGS=-g -Wall -O3，意味着gcc在编译我的示例程序时会抛出它所知道的每个优化技巧。这些优化技巧知道在什么时候把指针作为restrict指针，什么时候把函数作为inline函数，而不需要我们明确地告诉编译器。

8.3.2 压力

在实践中，有时候会发现const会带来需要解决的压力：当有一个指针被标记为const时，但是需要把它作为输入发送给一个对应的形参并没有使用const标记的函数。这也许是因为这个函数的作者觉得这个关键字太麻烦了，或者坚持认为更短的代码是更好的代码，或者仅仅是忘记了。

在采取行动之前，必须给自己提出一个问题，在这个未使用const的被调用函数中，是否以任何方式修改了这个指针？这可能是由于某些东西发生了变化而导致的边界情况，也可能是由于其他奇怪的原因。不管怎样，了解这方面的材料还是值得的。

如果能够确信这个函数并没有违背我们通过const关键字对指针所施加的常量承诺，把这个常量指针强制转换为非常量指针以避免编译器报错就是非常合理的做法。

```
//No const in the header this time...
void set_elmt(int *a, int *b){
    a[0] = 3;
}

int main(){
    int a[10];
    int const *b = a;
    set_elmt(a, (int*)b); //...so add a type-cast to the call.
}
```

注意这里没有const

这个规则对我而言非常合理。我们可以跳过编译器的常量检查，只要明确地指定这种做法并表示其结果在自己预料之中。

如果担心自己所调用的函数无法实现常量承诺，可以更深入一步，

创建输入数据的一份完整复制，而不是创建它的一个别名。由于我们并不需要这个变量所发生的任何变化，因此可以在以后抛弃这份复制。

8.3.3 深度

假设有一个结构counter_s，并且有一个函数接受一个该结构类型的参数，形式类似于f（counter_s const *in）。这个函数能否修改这个结构的成员呢？

让我们尝试一下：例8-5生成一个具有2个指针的结构。在ratio函数中，这个结构是为作为常量传递的，但是当我们把这个结构中的其中一个指针发送给非常量限定的子函数时，编译器并不会报错。

例8-5　常量结构的成员并不是常量（conststruct.c）

```
#include <assert.h>
#include <stdlib.h> //assert

typedef struct {
    int *counter1, *counter2;
} counter_s;

void check_counter(int *ctr){ assert(*ctr !=0); }

double ratio(counter_s const *in){                    ❶
    check_counter(in->counter2);                      ❷
    return *in->counter1/(*in->counter2+0.0);
}

int main(){
    counter_s cc = {.counter1=malloc(sizeof(int)),    ❸
                    .counter2=malloc(sizeof(int))};
    *cc.counter1 = *cc.counter2 = 1;
    ratio(&cc);
}
```

❶ 输入的结构被标记为const。

❷ 我们把这个常量结构的一个成员发送给一个接受一个非常量输入的函数。编译器并不会报错。

❸ 这是通过指定的初始化值所进行的声明——稍后讨论。

在结构的定义中，我们可以用const指定某个成员，尽管这种做法所带来的麻烦往往大于好处。如果我们真正需要在类型层次结构的最底层进行保护，最好还是在文档中添加说明。

8.3.4 char const **问题

例8-6是个简单的程序，检查用户是否通过命令行提供了Iggy Pop的名称。下面是在shell上的示例用法（记住$?是刚刚运行的程序的返回值）：

```
iggy_pop_detector Iggy Pop; echo $?     #prints 1
iggy_pop_detector Chaim Weitz; echo $?  #prints 0
```

例8-6　标准的歧义导致了指向常量的指针的指针所存在的各种问题（iggy_ pop_detector.c）

```
#include <stdbool.h>
#include <strings.h> //strcasecmp (from POSIX)

bool check_name(char const **in){  ❶
    return   (!strcasecmp(in[0], "Iggy") && !strcasecmp(in[1], "Pop"))
           ||(!strcasecmp(in[0], "James") && !strcasecmp(in[1],"Osterberg"
));
}

int main(int argc, char **argv){
    if (argc < 2) return 0;
```

```
    return check_name(&argv[1]);
}
```

❶ 如果以前没有看到过布尔类型，我将在后面的一个附加内容中介绍它们。

check_name函数接受一个指向“常量字符串”的指针，因为我们不需要对输入字符串进行修改。但是当我们编译代码时，会发现编译器产生了一个警告。Clang表示：“向const char **类型的形参传递char **会丢弃嵌套指针类型中的限定符。”在一连串的指针中，我能够找到的所有编译器都会把所谓的顶层指针转换为const（也就是说转换为char * const *），但是当我们要求编译器对该指针所指向的东西进行转换时，它就会发出警告（char const **，也可以写作const char **）。

这样，我们需要一个显式的类型转换，把check_name（&argv[1]）替换为：

```
check_name((char const**)&argv[1]);
```

这种完全合理的类型转换为什么不会自动发生呢？我们需要在问题发生之前进行一些独特的环境设置，而问题在于这部分的语法规则并不一致。接下来的解释比较难懂，所以如果你忽略了它，我是可以理解的。

例8-7中的代码在图中创建了3个链接：从constptr -> fixed的直接链接，以及从constptr -> var和var -> fixed的间接链接。在代码中，我们可以看到有两个赋值是明确进行转换的：constptr -> var和constptr -> fixed。但是由于*constptr == var，因此第2个链接隐式地创建了var ->

fixed链接。当我们进行赋值*var=30时，就相当于进行了赋值fixed = 30。

例8-7　我们可以通过另一个名字修改数据，即使它在一个名字下是常量，这种做法被认为是不应该的（constfusion.c）

```
#include <stdio.h>

int main(){
    int *var;
    int const **constptr = &var; // the line that sets up the failure
    int const fixed = 20;
    *constptr = &fixed;          // 100% valid
    *var = 30;
    printf("x=%i y=%i\n", fixed, *var);
}
```

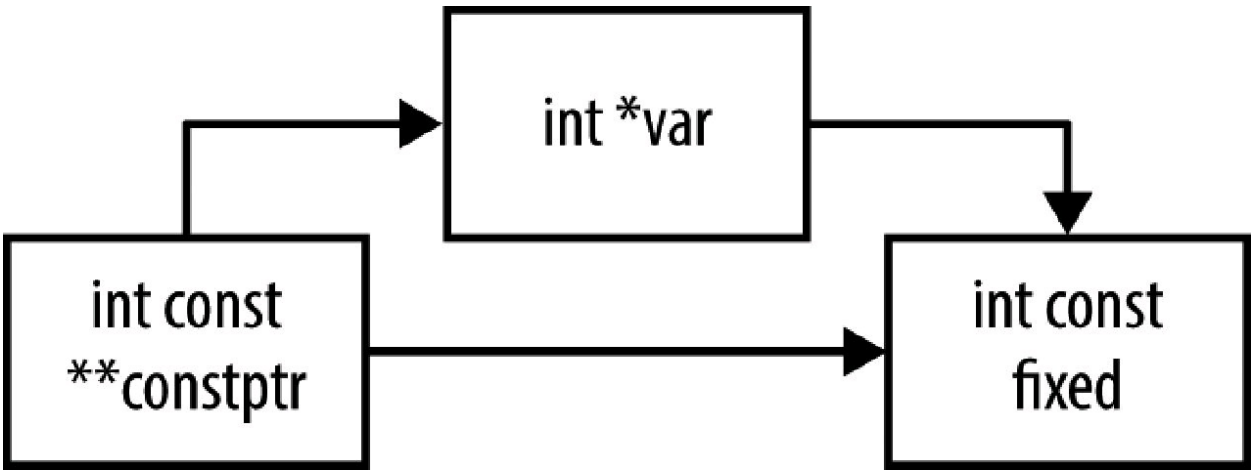


图8-2　示例8-7中各变量之间的连接关系

我们没办法让int *var直接指向int const fixed。示例程序通过一些手段让var实际上指向了fixed，但是并没有在程序中明确地指明。

> 自己动手：是不是可以产生一个类似这样的const失败，但是不被允许的类型转换恰好发生在函数调用的过程中，如同Iggy Pop

> 检测器一样？

如前所述，在一个名字下用const所标记的数据可以通过另一种不同的名字进行修改。因此，事实上用另外一个名字修改一个常量数据的能力还是令人吃惊的[5]。

我列举了const所存在的一些问题，便于读者克服它们。按照文艺的说法，并不是说这些做法都是有问题的，而是建议只要合适，应该尽可能在函数声明中添加const，而不是简单地抱怨前辈们没有提供正确的函数头部。不管怎样，总有一日会有其他人使用你的代码，你肯定不希望他们抱怨无法使用const关键字，而原因只是因为你并没有正确地声明函数的头部。

> 真和假
>
> C最初没有布尔（true/false）类型，而是采用了一个约定，即0或NULL表示false，其他值表示true。因此，if（ptr!=NULL）和if（ptr）是相同的，如果不习惯第2种形式，要习惯使用它，因为你的C程序员同伴（包括我自己）期望你能够毫不犹豫地读懂它。（例外：程序返回值的习惯是零表示成功，非零值表示失败。）
>
> C99引入了_Bool类型，它从技术上说是不必要的，因为我们总是可以使用整数来表示真/假值。但是站在阅读代码的角度，布尔类型清楚地说明变量只能接受真/假值，因此对它的用途提供了一些提示。
>
> 标准委员会选择_Bool这个字符串的原因是它属于为语言的扩展所保留的字符串空间，但这种写法显然有点笨拙。stdbool.h头文件定义了3个宏以提高它的可读性：bool的展开结果是_Bool，这样就可以避免在声明中使用麻烦的下划线。true的展开结果是1，false的展开结果是0。
>
> 就像bool类型更容易被人理解一样，true和false宏也可以清晰地表示赋值的目的：如果我忘了outcome被声明为bool类型，则outcome=true这样的语句可以清晰地提醒outcome的用途，

> 则outcome=1显然无此功效。
>
> 但是，实际上没有理由将任何表达式与true或false进行比较：我们都习惯了if （x）的意思就是“如果x为true，则……”，而不需要明确地写出==true。而且，假设int x = 2，if （x）的结果正如我们所预期的那样，而if （x==true）却并非如此。

---

[1] 还有一种选择，那就是把语句块包含在if（1){…} else（void)0中，它同样会吸收掉分号。这个方法也可以，但是如果这个宏包含在另外一个if-else语句中的时候，如果你使用-Wall， 那么它就会触发一个警告，对于用户来说也是不透明的。

[2] 关于空白宏参数的有效性，参考C99和C11，§ 6.10.3（4），允许“不包含预处理标记的参数”。

[3] 参考*Microsoft Developer Network*。

[4] 如果类型是相同的，那么重复的typedef是没有问题的。如C11§6.7（3）：“一个typedef 名字可以被重定义用来代表相同的类型，只要这个类型不是可变的修改后的类型。”

[5] 这里的代码是对C99和C11 §6.5.16.1（6）的例子的改写，其中与const ptr=&var类似的代码行被标记为违反了约束条件。为什么gcc和Clang会把它标记为警告而不是终止？因为它在技术上是正确的：C99和C11 §6.3.2.3（2）在描述关于像const这样的类型限定符时，解释了“对于任何限定符q，指向一个非q所限定类型的指针可以转换为一个指向由q所限定的该类型的指针……”

第9章　简单的文本处理

一个字母字符串是个长度不确定的数组，这个数组是自动分配的（在栈stack上分配），并且不能改变大小，这正是C在处理文本时所存在的核心问题。幸运的是，在我们之前有许多人已经面对这个问题，并且提供了至少一部分的解决方案。许多C和POSIX标准的函数足够满足日常处理字符串的许多需求。

另外，C是在20世纪70年代设计的，出现在非英语的计算机语言发明之前。同样，只要使用正确的函数（加上对语言编码的正确理解），C原先只考虑英语的情况也不会成为真正的问题。

9.1　使用asprintf，使字符串的处理不再那么痛苦

asprintf函数分配必要的字符串空间并填充字符串。这意味着我们实际上完全不需要操心字符串内存的分配问题了。

asprintf并不是C标准的组成部分，但可以在支持GNU或BSD标准库的系统中使用，而这两个标准库的普及率相当之高。而且，GNU的libiberty库提供了asprintf的一个版本，我们可以把它复制、粘贴到自己的代码中，或者在链接器中通过-liberty命令开关来从库中调用这个函数。这个库通常会和一些没有原生asprintf函数的系统一起分发，例如为Window准备的MSYS。如果从库中复制和粘贴源代码并不可行，我会演示如何用vsnprintf函数快速地实现asprintf函数。

旧式的字符串填充方法会逼得人们想要杀人（或自杀，取决于性格），因为它们首先必须获取想要填充的字符串的长度，接着分配空间，然后再实际写入空间。另外，还不能忘了添加额外的null终止符！

例9-1演示了一种创建字符串的痛苦方法，其意图是使用C的system命令运行一个外部工具。strings外部工具在一个二进制文件中搜索可打印的普通文本。get_strings函数所接受的argv[0] 参数就是程序本身的名称，因此这个程序将通过搜索自身来创建字符串，这似乎有点滑稽，但对于示例代码我们也不能要求太多。

例9-1　创建字符串的乏味方法（sadstrings.c）

```
#include <stdio.h>
#include <string.h> //strlen
#include <stdlib.h> //malloc, free, system

void get_strings(char const *in){
    char *cmd;
    int len = strlen("strings ") + strlen(in) + 1;   ❶
    cmd = malloc(len);                               ❷
    snprintf(cmd, len, "strings %s", in);
    if (system(cmd)) fprintf(stderr, "something went wrong running %s.\n",
 cmd);
    free(cmd);
}

int main(int argc, char **argv){
    get_strings(argv[0]);
}
```

❶ 预先确定长度完全是浪费时间。

❷ 在C标准中sizeof（char）==1，因此我们至少不需要费力写成malloc（ len*sizeof（char））。

例9-2使用了asprintf，它会为我们调用malloc，这意味着我们不再需要计算字符串长度这个步骤了。

例9-2　这个版本只是从例9-1中删除了2行，但这2行却是最令人感觉痛苦的（getstrings.c）

```
#define _GNU_SOURCE //cause stdio.h to include asprintf
#include <stdio.h>
#include <stdlib.h> //free

void get_strings(char const *in){
    char *cmd;
    asprintf(&cmd, "strings %s", in);
    if (system(cmd)) fprintf(stderr, "something went wrong running %s.\n",
 cmd);
    free(cmd);
}

int main(int argc, char **argv){
    get_strings(argv[0]);
}
```

asprintf的实际调用看上去与sprintf调用很像，区别在于前者需要传入的是字符串在内存中的位置而不是字符串本身，因为这个函数会分配新的空间，并且这个位置将写入我们所输入的char **类型参数。

如果出于什么原因，GNU的asprintf不能用，就像printf语句那样来计算长度又容易引起错误，计算机能为我们做什么呢？答案已经存在了，C99中vsnprintf函数返回应该写入的字符的个数，函数这里假设*n*大于所要写入的字符数（不包括末尾的截止符），或者是编码错误发生时返回的负值。Snprintf函数的返回值与此类似。

所以如果在只有一字节的字符串上来运行vsnprintf函数，我们就可以得到应该写入的字符串的长度的数值。然后我们可以申请这个字符串

长度的内存，然后使用vsnprintf。我们运行这个函数两次，所以运行时间多了一倍，但是对于安全和方便来说，这是值得的。

例9-3演示了一个通过运行vsnprintf两次来实现asprintf的程序，我把HAVE_ ASPRINT包含进来以便使autoconf友好。如下所示。

例9-3　一个asprintf的另外一种实现

```
#ifndef HAVE_ASPRINTF
#define HAVE_ASPRINTF
#include <stdio.h> //vsnprintf
#include <stdlib.h> //malloc
#include <stdarg.h> //va_start et al

/* The declaration, to put into a .h file. The __attribute___ tells the co
mpiler to check printf-style support it; just remove it if yours doesn't.
*/

int asprintf(char **str, char* fmt, ...) __attribute__ ((format (printf,2,
3)));

int asprintf(char **str, char* fmt, ...){
    va_list argp;
    va_start(argp, fmt);
    char one_char[1];
    int len = vsnprintf(one_char, 1, fmt, argp);
    if (len < 1){
        fprintf(stderr, "An encoding error occurred. Setting the input poi
nter to NULL.\n");
        *str = NULL;
        return len;
    }
    va_end(argp);

    *str = malloc(len+1);
    if (!str) {
        fprintf(stderr, "Couldn't allocate %i bytes.\n", len+1);
        return -1;
    }
    va_start(argp, fmt);
    vsnprintf(*str, len+1, fmt, argp);
    va_end(argp);
```

```
    return len;
}
#endif

#ifdef Test_asprintf
int main(){
    char *s;
    asprintf(&s, "hello, %s.", "—Reader—");
    printf("%s\n", s);

    asprintf(&s, "%c", '\0');
    printf("blank string: [%s]\n", s);

    int i = 0;
    asprintf(&s, "%i", i++);
    printf("Zero: %s\n", s);
}
#endif
```

9.1.1 安全

假设有一个预先确定长度的字符串str，我们使用sprintf向它写入未知长度的数据，那么这些数据可能会越过边界，写入与str相邻的内存，这是一个经典的安全漏洞。由于这个原因，sprintf很快被snprintf所取代，后者可以限制写入数据的数量。

使用asprintf可以有效地防止这个问题，因为它会分配需要写入的足量内存。它并不完美：最后总会有些受损的或不适当的输入字符串，在中间的某处出现“\0”，但数据的总量最终很可能会超过自由内存的数量，或者写入到str的额外数据的地方，而这些地方可能保存着像密码这样的敏感信息。

如果可用的记忆体不足，asprintf将返回-1，因此在涉及用户输入的场合，细心的程序员会使用类似Stopif宏（10.2“可变参数宏”一节中介绍）这样的东西，其形式类似：

```
Stopif(asprintf(&str, "%s", user_input)==-1, return -1, "asprintf failed.")
```

但是，如果把一个未经检查的字符串发送给asprintf，已经是有点晚了。对来自未受信任的输入的字符串应预先进行检查，保证它具有安全长度。这个函数既使对于合理长度的字符串仍然有可能失败，因为计算机的内存可能已经被耗尽或者被小妖精吃掉了。

C11（附录K）也提供了所有的通常的格式输出函数，它们都有_s后缀，如printf_s， snprintf_s等。它们都比那些没有_s的更安全。输入的字符串可能不是NULL, 如果你试图写多于RINT_MAX的字节到一个字符串中（RINT_MAX一般是size_t最大容量的一半），函数会以一个违反运行限制的错误返回，但是在标准C库中支持这些函数也是可选的。

9.1.2 常量字符串

下面这个程序创建了2个字符串并把它们输出到屏幕上：

```
#include <stdio.h>

int main(){
    char *s1 = "Thread";

    char *s2;
    asprintf(&s2, "Floss");

    printf("%s\n", s1);
    printf("%s\n", s2);
}
```

这两种形式都产生了一个包含单词的字符串。但是，C编译器会用截然不同的方法来处理它们，令不明真相的人们大跌眼镜。

读者是否尝试过前面那段显示被嵌入到程序二进制文件的字符串的示例代码？在现在这个例子中，“Thread”就是一个这样的嵌入字符串，并且s1可能指向这个程序的可执行文件中的一个位置。这是多么的高效，我们不需要花费运行时间让系统对字符进行计数或者浪费内存重复存储在二进制文件中已经存在的信息。如果是在20世纪70年代，这确实会带来显著的区别。

但是站在代码阅读者的角度，预先创建好的s1和根据需要分配的s2在行为上是相同的，但是我们无法修改或释放s1。我们可以向这个例子添加下面几行代码，并观察它们的效果：

```
s2[0]='f'; //Switch Floss to lowercase.
s1[0]='t'; //Segfault.

free(s2); //Clean up.
free(s1); //Segfault.
```

系统可能会直接指向嵌入可执行文件的字符串，或者可能把这个字符串复制到一个只读的数据区段中。C标准绝不会指定这种实现细节，但它会指定s1的内容是绝对只读的。

常量字符串和变量字符串的区别是微妙的，并且容易滋生错误，这使得硬编码字符串的适用环境比较有限。我无法想象脚本语言会要求我们关注这种差别。

但是有一个简单的解决方法：strdup。这是个POSIX标准的函数，其名称是字符串复制（string duplicate）的简写。它的操作如下：

```
char *s3 = strdup("Thread");
```

“Thread”字符串仍然是以硬编码的形式写入程序，但s3是这个常量对象的一份复制，因此可以根据需要自由地修改。通过使用strdup，就能以相同的方法处理所有的字符串，而不必操心哪些是常量，哪些是指针。

如果无法使用POSIX标准并担心自己的计算机上没有strdup的一份复制，可以很方便地为自己编写一个版本，这里又一次使用了asprintf：

```
#ifndef HAVE_STRDUP
char *strdup(char const* in){
    if (!in) return NULL;
    char *out;
    asprintf(&out, "%s", in);
    return out;
}
#endif
```

HAVE_STRDUP宏来自哪里呢？如果使用的是Autotools，就把下面这一行：

```
AC_CHECK_FUNCS([asprintf strdup])
```

放到configure.ac中，就可以在配置脚本中生成一段内容，根据是否适当地定义了HAVE_STRDUP和HAVE_ASPRINTF，生成一个configure.h文件。

9.1.3 用asprintf扩展字符串

下面是个基本形式的例子，使用asprintf把另一小段文本添加到一个字符串的尾部。

```
asprintf(&q, "%s and another clause %s", q, addme);
```

我用它来生成数据库查询。我要把字符串做一连串的连接，就像下面这个人工痕迹浓重的例子一样：

```
int col_number=3, person_number=27;
char *q =strdup("select ");
asprintf(&q, "%scol%i \n", q, col_number);
asprintf(&q, "%sfrom tab \n", q);
asprintf(&q, "%swhere person_id = %i", q, person_number);
```

最后所得到的结果是：

```
select col3
from tab
where person_id = 27
```

当你要构建一个又长又痛苦的字符串的时候，这个方法相对要简单一些。当查询字符串的分句非常复杂时，这种方法显然尤其重要。

但是，它会导致内存泄漏，因为q的原地址中的对象在asprintf向q提供一个新位置时并不会被释放。对于一次性的字符串生成，这个问题无关紧要，就算我们在内存中丢下上百万个像查询字符串这样长度的字符串，也不会对系统产生明显的影响。

如果所生成的字符串数量是未知的，并且每个字符串的长度也是未知的，那么就需要一种类似例9-4这样的格式。

例9-4　一个清晰地扩展字符串的宏（sasprintf.c）

```
#include <stdio.h>
#include <stdlib.h> //free

//Safer asprintf macro
#define Sasprintf(write_to, ...) {            \
    char *tmp_string_for_extend = (write_to); \
```

```
    asprintf(&(write_to), __VA_ARGS__);       \
    free(tmp_string_for_extend);              \
}

//sample usage:
int main(){
    int i=3;
    char *q = NULL;
    Sasprintf(q, "select * from tab");
    Sasprintf(q, "%s where col%i is not null", q, i);
    printf("%s\n", q);
}
```

Sasprintf宏加上偶尔使用strdup，差不多能够应付100%的字符串处理需要了。除了一个小问题以及偶尔需要使用free之外，我们根本无须考虑内存问题。

这个小问题是如果我们忘了把q初始化为NULL或者忘了使用strdup，那么在第1次使用Sasprintf宏时将会释放q的未初始化位置所保存的无意义数据，这会产生一个段错误。

例如，下面的代码将会失败，把声明包装在strdup中可以避免失败：

```
char *q = "select * from"; //fails—needs strdup().
Sasprintf(q, "%s %s where col%i is not null", q, tablename, i);
```

从理论上说，这种类型的字符串连接在广泛使用时会导致性能下降，因为字符串的第一部分会被不断地改写。在这种情况下，可以用当前的C作为C的原型语言：当且仅当这个技巧被证明确实太慢时，可以花点时间用传统的snprintf函数来代替它。

9.1.4 strtok的赞歌

标记解析（Tokenizing）是最简单也是最常见的解析问题，也就是根据分隔符把一个字符串分割为几个部分。这个定义覆盖了所有这种类型的任务。

- 根据空白分隔符（例如" \t\n\r"之一）分割单词。
- 假设有个像"/usr/include:/usr/local/include:."这样的路径，在冒号处将其分开，形成单独的目录。
- 根据一个简单的换行分隔符"\n"把一个字符串分割为不同的行。
- 可以使用一个配置文件，包含value = key格式的行，在这种情况下分隔符就是"="。
- 在数据文件中以逗号分隔的值当然是以逗号为分隔符。

我们可以采取两个层次的分割来分别进行处理。例如读取一个完整的配置文件，首先根据换行符进行分割，然后在每行根据=进行分割。

提示

如果你不仅仅需要根据分隔符来分割符号，还需要一些复杂的字符串功能，那么你需要正则表达式。13.2.1“解析正则表达式”一部分的内容讨论了POSIX标准的正则表达式解析器如何抽取出子字符串。

标记解析出现得非常频繁，有一个C标准库函数strtok（字符串标记化）专门用于完成这个任务。它是那些虽然小巧但能够干净利落地完成

任务的函数之一。

strtok的基本工作方式是对我们所输入的字符串进行迭代，直到遇到第1个分隔符，然后用一个'\0'覆盖这个分隔符。现在这个输入字符串的第1部分已经是个表示第1个标记的合法字符串，strtok返回一个指向这个子字符串头部的指针供我们使用。这个函数在内部保存源字符串的信息，因此当我们再次调用strtok时，它可以搜索到下一个标记的尾部，将之设置为‘\0’，并以一个合法的字符串的形式返回这个标记。

每个子字符串的指针都指向原字符串内的位置，因此标记化操作所执行的数据写入是极少的（只是那些\0），并且不会进行复制操作。它的直接影响是输入字符串被损坏，由于子字符串是指向源字符串的指针，因此无法释放这个输入字符串，必须等到完成了所有子字符串的使用以后，你才能去释放原始的字符串。（或者，可以使用strdup复制出这些子字符串。）

strtok函数通过一个静态的内部指针保存我们第1次输入的字符串的剩余部分，意味着它被限制为一次只能标记化一个字符串（通过一组分隔符），无法在涉及线程的情况下使用。因此目前strtok已经不推荐使用了。

推荐使用strtok_r或strtok_s，它们是strtok的线程友好版本。POSIX标准提供了strtok_r，C11标准提供了strtok_s。这两个函数的用法都有点笨拙，因为第1次调用与后续的调用在形式上是不一样的。

- 第1次调用函数时，把需要解析的字符串作为它的第1个参数。
- 在后续的调用中，把第1个参数设置为NULL。

- 最后一个参数是处理过的字符串，我们并不需要在第1次使用时对它进行初始化，在后续的调用中，它将保存到目前为止所解析的字符串。

下面是个行计数器（事实上是非空白行的计数器，参见后面的警告）。在脚本语言中，标记解析往往只需要一行程序，但是以下已经是strtok_r最精简的用法了。注意用于发送源字符串的if ? then:else只出现在第1次呼叫时传入字符串。

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

int count_lines(char *instring){
    int counter = 0;
    char *scratch, *txt, *delimiter = "\n";
    while ((txt = strtok_r(!counter ? instring : NULL, delimiter, &scratch
)))
          counter++;
    return counter;
}
```

后面的9.2“Unicode”一节将提供一个完整的例子，第11章的11.5“引用计数”中的Cetology也是这样的一个例子。

C11标准的strtok_s函数的工作方式就像strtok_r一样，但接受一个额外的参数（第2个），它提供了输入字符串的长度，并在后续的调用过程中不断缩短，表示每次调用时剩余字符串的长度。如果输入字符串并不是以\0分隔的，这个额外的参数就非常实用。我们可以用下面的代码重新完成前面那个例子：

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

//first use
size_t len = strlen(instring);
txt = strtok_s(instring, &len, delimiter, &scratch);
```

```
//subsequent use:
txt = strtok_s(NULL, &len, delimiter, &scratch);
```

警告

连续的2个或更多个分隔符被当作单个分隔符，意味着空白标记被简单地忽略。例如，如果分隔符是":"并要求strtok_r或strtok_s分解“/bin:/ usr/bin::/opt/bin”，将会按顺序得到3个目录，::被当作:。这也是前面那个行计数器实际上是个非空行计数器的原因，因为字符串中的连续两个换行符（例如“one \n\n three \n four”，表示第2行为空白）将被strtok以及它的变型当成单个换行符。

忽略连续的分隔符往往是我们想要的结果（例如上面这个路径例子），但有时候情况并非如此，我们可能需要考虑怎样检测连续的分隔符。如果需要被分割的字符串是自己所编写的，就要保证在生成字符串时使用一个记号，表示有意的空白标记。编写一个函数，对字符串进行连续分隔符的预先检查并不是件困难的事情（或者可以尝试BSD/GNU标准的strsep函数）。对于来自用户的输入，可以添加严厉的警告，表示不允许出现连续的分隔符，告诉他们这样做会出现什么结果，就像上面这个行计数器会忽略空白行一样。

例9-6展示了一个小型的字符串工具库，它相当实用，包括了本书前面所讨论的一些宏。

这里有2个关键函数：string_from_file把一个完整的文件读取到一个字符串。这样就避免了与读取并处理更小的文件块相关的所有麻烦。如

果读者日常所处理的文本文件都是些几个GB的巨无霸，自然用不着这个函数。不过文本文件的大小绝对不会超过几MB的情况下，就完全没有必要以增量的方式循环读取文件的一小块。我将在本书的几个示例程序中使用这个函数。

第2个关键函数是ok_array_new，它对一个字符串进行标记化，并把输出写入 ok_array结构中。

例9-5是头文件。

例9-5　一小组字符串工具的头文件（string_utilities.h）

```
#include <string.h>
#define _GNU_SOURCE //asks stdio.h to include asprintf
#include <stdio.h>

//Safe asprintf macro
#define Sasprintf(write_to, ...) {          \                    ❶
    char *tmp_string_for_extend = write_to; \
    asprintf(&(write_to), __VA_ARGS__);     \
    free(tmp_string_for_extend);            \
}

char *string_from_file(char const *filename);

typedef struct ok_array {
    char **elements;
    char *base_string;
    int length;
} ok_array;                                                      ❷

ok_array *ok_array_new(char *instring, char const *delimiters);  ❸

void ok_array_free(ok_array *ok_in);
```

❶ 这是前面的Sasprintf宏，为方便起见复制于此处。

❷ 这是一个标记数组，在调用ok_array_new对一个字符串进行标记解析时所得。

❸ 这是strtok_r的包装部分，将生成ok_array。

例9-6让GLib把一个文件读取到一个字符串，并使用strtok_r函数把一个单独的字符串变成一个字符串数组。我们将在例9-6、例12-2和例12-3中看到一些用法示例。

例9-6　一些实用的字符串工具（string_utilities.c）

```
#include <glib.h>
#include <string.h>
#include "string_utilities.h"
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> //abort

char *string_from_file(char const *filename){
    char *out;
    GError *e = NULL;
    GIOChannel *f = g_io_channel_new_file(filename, "r", &e);    ❶
    if (!f) {
        fprintf(stderr, "failed to open file '%s'.\n", filename);
        return NULL;
    }
    if (g_io_channel_read_to_end(f, &out, NULL, &e) != G_IO_STATUS_NORMAL)
{
        fprintf(stderr, "found file '%s' but couldn't read it.\n", filenam
e);
        return NULL;
    }
    return out;
}

ok_array *ok_array_new(char *instring, char const *delimiters){   ❷
    ok_array *out= malloc(sizeof(ok_array));
    *out = (ok_array){.base_string=instring};
    char *scratch = NULL;
    char *txt = strtok_r(instring, delimiters, &scratch);
    if (!txt) return NULL;
```

```
    while (txt) {
        out->elements = realloc(out->elements, sizeof(char*)*++(out->lengt
h));
        out->elements[out->length-1] = txt;
        txt = strtok_r(NULL, delimiters, &scratch);
    }
    return out;
}

/* Frees the original string, because strtok_r mangled it, so it
   isn't useful for any other purpose. */
void ok_array_free(ok_array *ok_in){
    if (ok_in == NULL) return;
    free(ok_in->base_string);
    free(ok_in->elements);
    free(ok_in);
}

#ifdef test_ok_array
int main (){                                                          ❸
    char *delimiters = " '~!@#$%^&*()_-+={[]}|\\;:\",<>./?\n";
    ok_array *o = ok_array_new(strdup("Hello, reader. This is text."), del
imiters);
    assert(o->length==5);
    assert(!strcmp(o->elements[1], "reader"));
    assert(!strcmp(o->elements[4], "text"));
    ok_array_free(o);
    printf("OK.\n");
}
#endif
```

❶ 尽管这种做法并不在所有场合下都是可行的，但我已经醉心于一次把一个完整的文本文件读取到内存中，这是一个很好的使程序员绕过心烦问题的例子。如果预计到需要读取到内存的文件过于巨大，可以使用mmap（q.v.）来实现相同的效果。

❷ 这是strtok_r的包装器。如果读者是从头看到这里的，应该已经熟悉这个while循环，它几乎是必需的。这个函数把从strtok_r所返回的结果记录到一个ok_array结构。

❸ 如果test_ok_array并未被设置，则它是一个用于其他地方的库。如果它被设置（CFLAGS=-Dtest_ok_array），则它是个用于测试ok_array_new是否正确工作的程序，方法是根据非字母字符将示例字符串进行分割。

9.2 Unicode

在只有美国有计算机的时代，ASCII为标准的美式QWERTY键盘上出现的所有普通字母和符号定义了一个数值码，称之为本地英语字符集。一个C字符的长度是8位（二进制数字），等于1字节，可以表示256个不同的值。ASCII定义了128个字符，因此可以用单个char来表示，并且还有1位的节余。也就是说，每个ASCII字符的第8位将是0，这个结果在后来发挥了出乎意料的作用。

Unicode遵循了与此相同的基本前提，为每个用于人类通信的字符置一个单独的十六进制数值（一般是从0000到FFFF）[1]。按照习惯，这些码是以U+0000的形式书写的。这项任务更有雄心，并且更具挑战性，因为它需要对所有普通的西方字符、数以万计的汉字和日语字符以及类似乌加里特和德撒律等语言所需要的所有必要的字形进行编目，涵盖了全世界古往今来的所有字形。

接下来的一个问题是Unicode是怎样进行编码的。谈到这个问题，分歧就开始出现了。最基本的问题是设置几字节作为分析单位。UTF-32（UTF代表UCS转换格式，UCS代表统一字符集）指定了32位（即4字节）作为基本单位，这意味着每个字符都可以用1个单位进行编码，但它所付出的代价是巨量的空白填充，因为本地英语字符只需要7个位

就足够了。UTF-16使用2字节作为基本单位，可以用1个单位表示绝大部分的字符，但有些字符则需要2个单位才能表示。UTF-8使用1字节为单位，意味着许多字符需要用多个单位才能表示。

我喜欢把UTF编码看成一种简单的加密方案。对于每个编码点，在UTF-8中有一个对应的字节序列，在UTF-16中也有一个对应的字节序列，在UTF-32也有一个对应的字节序列，而这几个字节序列并不一定相关。除了后面所讨论的一个特例之外，我们没有理由认为一个编码点与它的任何一个加密值在数值上是相同的，甚至通过某种明显的方式存在关联。但是我知道优秀的解码器可以很轻松且无歧义地在各种UTF编码方案和正确的Unicode编码点之间进行转换。

那么，在机器的世界上是如何选择的呢？在网络上，赢家非常明显：超过73%的网站使用了UTF-8。另外，Mac和Linux操作系统在默认情况下使用UTF-8表示任何文本，因此我们有很大机会看到Mac或Linux中的未标注文本是采用UTF-8的。

全球大约有10%的网站仍然没有采用Unicode，而是采用了一种相对过时的格式——ISO/IEC 8859（具有名称类似Latin-1的代码页）。非POSIX操作系统的Winodws则使用了UTF-16。

Unicode的显示取决于操作系统，为此它需要完成大量的操作。例如，在打印本地英语字符集时，每个字符在文本行中获得一个位置，但是以希伯莱的בּ（b）为例，却是ב（U+05D1）和（U+05BC）的组合。元音被添加到辅音上，进一步构建了这个字符：  = ba （U+05D1 + U+05BC + U+05B8）。至于在UTF-8中需要用多少（在此例中为6）字

节来表示这3个编码点则是另一个层面的问题。现在，当我们讨论字符串的长度时，我们所表示的究竟是编码点的数量、字符串在屏幕上的宽度还是表示这个字符串所需要的字节数？

因此，作为一个需要与各种语言的人们进行交流的程序的作者，我们的责任是什么？我们需要：

- 确定宿主系统所使用的编码方式，这样就不至于用错误的编码来读取输入，并可以发送回可以被宿主系统正确解码的输出。
- 成功地存储文本，并保证数据完好无损。
- 认识到一个字符并不占据固定数量的字节，因此我们所编写的任何以基地址加偏移量的代码（假设有个Unicode字符串us，像us++这样的写法）可能会产生编码点的碎片。
- 用一个便利的工具函数完成任何类型的文本理解：toupper和tolower只适用于本地英语字符，因此我们需要替代品。

为了实现这些责任，就需要挑选正确的内部编码以防止字符串的损坏，并拥有一个良好的函数库在需要解码的时候提供帮助。

9.2.1 C代码的编码

内部编码的选择尤其简单。UTF-8可以说是为C程序员量身定做的。

- UTF-8的单位是8位，也就是一个char。把一个UTF-8字符串写成一个char *字符串是完全合法的，就像本地英语文本一样。
- 前128个Unicode码完全与ASCII码匹配。例如，*A*在ASCII码中是

41（十六进制），它的Unicode代码点是U+0041。因此，如果Unicode文本恰好完全是由本地英语文本所组成的，就可以对它们使用普通的面向ASCII的工具，或者使用UTF-8工具。有个技巧是如果一个char的第8位是0，则这个char表示一个ASCII字符串。如果它是1，则这个char是个一连串的多字节字符。因此，UTF-8中非ASCII的Unicode字符绝不会与任何ASCII字符匹配。

- U+0000是个合法的编码点，C程序员喜欢把它写成'\0'。由于\0也表示ASCII的零，因为这个规则是上面那条规则的一种特殊情况。这是非常重要的，因为在尾部带有\0的UTF-8字符串正是我们所需要的合法的char 字符串。这个单元的大部分空间会被填充。这意味着前*8*位很有可能由全零所组成，意味着把*UTF-16*或*UTF-32*文本赋值给一个*char* 变量时，很可能会出现一个充满null字节的字符串。

因此，作为C程序员，我们必须考虑清楚：UTF-8编码的文本可以像以前一直使用的char *字符串类型一样进行存储和复制。既然一个字符的长度可能有几字节，因此要注意避免更改任何字节的顺序，而且保证绝对不会分割一个多字节字符。如果没有采取这样的做法，如果字符串是本地英语字符串，那也不会出现问题。因此，下面是UTF-8安全的标准库函数的不完整列表。

- strdup和strndup。
- strcat和strncat。
- strcpy和strncpy。
- POSIX的basename和dirname。
- strcmp和strncmp，但只在用它们作为零或非零函数来判断两个字符串是否相等时。如果想要进行有意义的分类，就需要一个定序函

数，参见下一节。

- strstr。
- printf族系的函数，包括sprintf，其中%s仍然作为字符串的标记符使用。
- strtok_r、strtok_s和strsep，前提是用" \t\n\r：|;,"分隔符之一以ASCII字符类单位进行分割。
- strlen和strnlen，但要认识到我们所得到的是字节数，并不是Unicode编码点的数量或者字符串在屏幕上的宽度。对于这两个数据，需要一个新的库函数，将在下一节讨论。

这些都是纯粹以字节为操作单位的函数，但我们对文本所执行的大多数操作要求对它进行解码，因此需要使用库函数。

9.2.2 Unicode函数库

我们的首要任务是把来自世界上其他地方的信息转换为UTF-8，这样就可以在内部使用数据。也就是说，我们需要使用看守函数把外来的字符串编码为UTF-8，并把UTF-8的字符串解码为输出字符串，供另一端的接收者所使用，这样就可以在一种合理的编码模式下完成所有的内部工作。

这也正是Libxml（将在本书第325页13.5“Libxml和cURL”一节遇到）的工作方式：一个合适的XML文档在头部陈述了它的编码方案（如果缺少编码声明，这个库会执行一组猜测规则），因此Libxml知道需要进行什么转换。Libxml把文档解析为一种内部格式，然后我们就可以查询和编辑这种内部格式。排除了错误之后，我们就可以保证这种内

部格式将是UTF-8格式，因为Libxml并不需要处理其他编码方案。

如果你做自己的翻译，那么你需要POSIX标准的iconv函数，这是一个超乎想象的复杂的函数，有如此多的编码要处理，GNU提供了一个可移植的libiconv函数，以防止你的电脑上没有这个函数。

提示

POSIX标准还指定了一个命令行的iconv程序，这是C函数的shell的友好包装器。

GLib提供了iconv的一些包装函数，我们需要关注的是g*locale_to_utf8*和*g_locale* from_utf8。当我们阅读GLib手册时，将会看到一个非常长的关于Unicode操作工具的章节。我们还将看到这些工具分为两种类型：一种操作于UTF-8，另一种操作于UTF-32（GLib是通过gunichar存储UTF-32字符的）。

记住，8字节并不足以用一个单位表示所有的字符，因此单个字符的长度是在1个到6个单位之间。UTF-8是以多字节编码为基础进行计数的，因此我们将遇到的问题是怎样获取字符串的真实长度（使用字符计数或屏幕宽度长度定义）、获取下一个完整的字符、获取一个子字符串或者获取一个用于排序的比较标准。

UTF-32具有足够的填充空间，可以用相同数量的字节块表示任何

字符，因此它被称为宽字符。我们经常会看到关于多字节字符和宽字符之间的转换的参考材料，这正是它们所讨论的主题。

对于一个UTF-32的字符（GLib的gunichar），我们可以毫不困难地完成与字符内容相关的操作，例如，获取它的类型（字母、数字或其他）、把它转换为大写或小写等。

如果阅读C标准，就会注意到它包含了一种宽字符类型，并且包含了与它相关的各种函数。wchar_t类型来自C89，因此出现在第一个Unicode标准发布之前。我不清楚它是不是真的还有用处。标准并没有确定wchar_t的宽度，因此它可能是32位，也可能是16位（或其他）。Windows上的编译器喜欢把它设置为16位，以迎合Microsoft对UTF-16的偏好，但UTF-16仍然是一种多字节的编码方案，因此我们需要另一种类型来保证真正的固定宽度的编码。C11提供了char16_t和char32_t作为固定宽度的字符。但是对于这两种类型，我们还没有太多相关的代码。

9.2.3 示例代码

例9-7展示了一个程序，它接收一个文件，并把它分割为各个“单词”，按照我的意思就是使用strtok_r根据空白和换行符进行分割，这也是相当普遍的做法。对于每个单词，我使用了GLib把第1个字符从多字节的UTF-8转换为宽字符的UTF-32，并确定第1个字符是字母、数字还是CJK类型的宽符号。（CJK代表中文/日文/朝鲜文，通常每个字符在输出的时候需要更大的空间。）

string_from_file函数把整个输入文件读取到一个字符串，然后local string_to_utf8把机器上的本地形式转换为UTF-8。strtok_r的用法的一个

值得注意的地方就是不需要对它多加关注。如果根据空白和换行符进行分割，就可以保证不会把一个多字节字符从中分断。

我把输出写入到一个HTML文件，因为接着就可以指定UTF-8，而不必担心输出端的编码。对于一台使用UTF-16的主机，也可以在浏览器中打开输出文件。

由于这个程序使用了GLib和string_utilities，因此我的makefile类似下面这样：

```
CFLAGS=='pkg-config --cflags glib-2.0' -g -Wall -O3
LDADD='pkg-config --libs glib-2.0'
CC=c99
objects=string_utilities.o

unicode: $(objects)
```

关于Unicode字符处理的另一个例子，可以参见例10-21，它对一个目录中所有采用UTF-8编码的文件中的每个字符进行列举。

例9-7　接收一个文本文件，并打印一些与它的字符有关的实用信息（unicode.c）

```
#include <glib.h>
#include <locale.h> //setlocale
#include "string_utilities.h"
#include "stopif.h"

//Frees instring for you—we can't use it for anything else.
char *localstring_to_utf8(char *instring){  ❶
    GError *e=NULL;
    setlocale(LC_ALL, ""); //get the OS's locale.
    char *out = g_locale_to_utf8(instring, -1, NULL, NULL, &e);
    free(instring); //done with the original
    Stopif(!g_utf8_validate(out, -1, NULL), free(out); return NULL,
           "Trouble: I couldn't convert your file to a valid UTF-8 string.
```

```
");
    return out;
}

int main(int argc, char **argv){
    Stopif(argc==1, return 1, "Please give a filename as an argument. "
                      "I will print useful info about it to uout.html.");

    char *ucs = localstring_to_utf8(string_from_file(argv[1]));
    Stopif(!ucs, return 1, "Exiting.");
    FILE *out = fopen("uout.html", "w");
    Stopif(!out, return 1, "Couldn't open uout.html for writing.");
    fprintf(out, "<head><meta http-equiv=\"Content-Type\" "
                 "content=\"text/html; charset=UTF-8\" />\n");
    fprintf(out, "This document has %li characters.<br>",
                 g_utf8_strlen(ucs, -1));                ❷
    fprintf(out, "Its Unicode encoding required %zu bytes.<br>", strlen(uc
s));
    fprintf(out, "Here it is, with each space-delimited element on a line
"
                 "(with commentary on the first character):<br>");

    ok_array *spaced = ok_array_new(ucs, " \n");         ❸
    for (int i=0; i< spaced->length; i++, (spaced->elements)++){
        fprintf(out, "%s", *spaced->elements);
        gunichar c = g_utf8_get_char(*spaced->elements); ❹
        if (g_unichar_isalpha(c)) fprintf(out, " (a letter)");
        if (g_unichar_isdigit(c)) fprintf(out, " (a digit)");
        if (g_unichar_iswide(c)) fprintf(out, " (wide, CJK)");
        fprintf(out, "<br>");
    }
    fclose(out);
    printf("Info printed to uout.html. Have a look at it in your browser.\n
");
}
```

❶ 这是一个入口看守函数，它把来自外部的字符串转换为UTF-8。输出的时候不需要看守函数，因为我把输出写入到一个HTML文件，浏览器知道怎样处理UTF-8。如果要编写输出看守函数，它应该与入口看守函数相似，区别在于使用g_locale_from_utf8。

❷ strlen是那种认为1个字符等于1字节的函数，因此我们需要它的

一个替代品。

❸ 使用本章前面的ok_array_new函数根据空白和换行符进行分割。

❹ 这是一些基于每个字符的操作，只适合于把多字节的UTF-8转换为一种固定宽度（宽字符）的编码形式之后。

Gettext

我们的程序很可能为读者输出大量的信息，例如错误信息和输入提示信息。真正用户友好的软件会将这些文本翻译成用户可以理解的语言。GNU的Gettext提供了一个框架，对这类翻译进行了组织。Gettext的手册很容易理解，因此读者可以阅读该手册并了解细节。下面提供了这个过程的粗略概述，帮助读者对这个系统有所了解。

- 把代码中的每个"Human message"实例替换为_（"Human message"）。下划线是个宏，最终将展开为一个函数调用，根据用户的运行时文件选择正确的字符串。
- 运行xgettext产生需要翻译的字符串的索引，其形式是可移植的对象模板文件（.pot）。
- 把.pot发送给全球范围内使用不同语言的同事，他们可以向你发送提供了把这些字符串翻译为他们所使用语言的.po文件。
- 在configure.ac中添加AM_GNU_GETTEXT（包括任何可选的用于指定在什么地方寻找.po文件以及其他类似细节的宏）。

---

[1] 从0000到FFFF这个范围是基本多语言平面（BMP），包括绝大多数（但不是全部）现代语言所使用的字符。后来的编码点（可以想象是从10000到10FFFF）位于补充平面中，包括数学符号（像表示实数的符号R）以及一个统一的CJK表意文字集。如果读者是中国数以百万计的苗族人之一，或者是印度数以亿计的讲索拉语或查克玛语的人之一，你的语言就位于这个平面。是的，绝大多数文本可以用MBP表示，但是如

果以为Uniode中的所有文本都在FFFF的范围之内，其立足点就是错误的。

第10章　更好的结构

本章所讨论的是接受结构作为输入的函数，看看它们能够为我们带来什么。

我们首先讨论ISO C99标准所引入的3个新加入的语法：复合常量（Compound Literal）、变长参数宏（Variable-length Macro）和指定初始化器（Designated initializer）。本章将对这些特性进行深入的探讨，并研究这些元素组合在一起时能够产生的各种混合效果。

使用复合常量，我们就可以更方便地向函数发送参数列表。接着，变长参数宏可以在用户面前隐藏复合常量的语法，以便使用能够接受任意个参数的函数：f（1, 2）或f（1, 2, 3, 4）将是同等合法的。

我们可以采用类似的技巧实现许多其他语言中的foreach关键字，或者对单输入函数进行向量化，使之能够对几个输入进行操作。

指定初始化器可以进一步简化对结构的操作，几乎可以完全抛弃旧式的方法。我们不必再使用难以理解并且容易出错的像person_struct p = {"Joe", 22, 75, 20}这样的记法，而是可以使用类似下面这样含义清晰的写法：person_struct p = {.name= "Joe", .age=22, .weight_kg=75, .education_years=20}。

随着结构的初始化不再成为问题，从函数返回一个结构也不再给我们造成困扰。我们可以更清晰地说明函数接口。

向函数发送结构也成为一种更可行的选项。把所有东西包装在另一个变长参数宏中，就可以编写接受可变数量的命名参数的函数，甚至为那些用户并未指定值的函数参数赋予默认值。一个贷款计算器例子会提供一个函数，amortization（.amount= 200000,.rate=4.5, .years=30）和amortization（.rate=4.5, .amount=200000）都是合法的用法。因为第二个调用并没有给出贷款时长，所以函数使用默认的30年。

本章最后提供一些示例程序，证明输入和输出结构可以使我们的工作变得更为简单，包括处理基于void指针的函数接口以及处理具有令人生畏的接口的遗留代码，后者必须包装在某种合适的结构中才更便于使用。

10.1 复合常量

我们可以非常方便地把一个常量值发送给C函数：假设已有声明double a_value，C毫无疑问能够理解f（a_value）。

但是，如果我们想发送的是一个元素列表，例如，像{20.38, a_value, 9.8}这样的复合常量，就需要警惕语法了：必须在复合常量前面加上类型转换，不然会让解析器感到困惑。因此这个列表应该看上去像（double[]）{20.38, a_value, 9.8}，对应的调用则如：

```
f((double[]) {20.38, a_value, 9.8});
```

复合常量是自动分配内存的，这意味着不必对它们使用malloc或free。在复合常量所在的作用域结束时，它就会自动消失。

例10-1首先显示了一个相当典型的函数sum，它接受一个double类型的数组，对所有的元素求和，直到遇到第1个NaN（“Not a Number，表示不是个数”，参见“用NaN标记异常数值”）元素。如果输入数组中不包含NaN，其结果将是灾难性的。我们必须采用一些安全措施。这个例子的main函数采用两种方式来调用它：通过一个临时变量的传统方法以及采用复合常量的方法。

例10-1　我们可以使用复合常量来绕过临时变量（sum_to_nan.c）

```
#include <math.h> //NAN
#include <stdio.h>

double sum(double in[]){                                    ❶
    double out=0;
    for (int i=0; !isnan(in[i]); i++) out += in[i];
    return out;
}

int main(){
    double list[] = {1.1, 2.2, 3.3, NAN};                   ❷
    printf("sum: %g\n", sum(list));
    printf("sum: %g\n", sum((double[]){1.1, 2.2, 3.3, NAN})); ❸
}
```

❶ 这个不起眼的函数将把输入数组中的元素相加，直到遇见第1个NaN标记。

❷ 这是接受数组为参数的函数的典型用法，它在单行代码中通过一个随用随弃的变量声明了这个列表，然后把它发送到接下来的那个函数中。

❸ 在这里，我们丢弃了那个中间变量，并使用了一个复合常量来创建一个数组，并直接把它发送给那个函数。

这是复合常量的最简单用法。本章的剩余部分将利用它们实现所有用途。另外，读者硬盘上使用了随用随弃的列表的代码是不是都可以通过复合常量实现流水线操作呢？

提示

这种形式是设置一个数组，而不是一个指向数组的指针，因此我们将使用（double[]）类型，而不是（double*）类型。

通过复合常量进行初始化

让我们深入探讨一个细微的区别，它有助于我们更牢固地理解复合常量的用法。

我们很可能习惯用下面的形式声明一个数组：

```
double list[] = {1.1, 2.2, 3.3, NAN};
```

我们分配了一个命名数组list。如果调用sizeof（list），得到的结果相当于当前计算机上4 * sizeof（double）的结果。也就是说，list是个数组（我们在6.1“自动，静态和手工内存”一节讨论过）。

我们还可以通过一个复合常量来执行声明，可以通过（double[]）执行类型转换予以确认：

```
double *list = (double[]){1.1, 2.2, 3.3, NAN};
```

系统在这里首先生成一个匿名列表，把它放在这个函数的内存帧中，然后声明一个指针list指向这个匿名列表。因此list是个别名，sizeof（list）等于sizeof（double*）。例8-2证明了这一点。

10.2 可变参数宏

依我所见，C的可变长度的函数很容易产生问题（详见本章后面10.11“灵活的函数输入”一节）。但是可变参数宏却非常简单。它的关键字是**VA_ARGS**，展开的结果是给定的元素集合。

在例10-2中，我重新回顾了例2-5，后者是printf函数的一个变型，它在一个断言失败时打印出一条消息。

例10-2　一个用于处理错误的宏，取自例2-5（stopif.h）

```
#include <stdio.h>
#include <stdlib.h> //abort

/** Set this to \c 's' to stop the program on an error.
    Otherwise, functions return a value on failure.*/
char error_mode;

/** To where should I write errors? If this is \c NULL, write to \c stderr.
 */
FILE *error_log;

#define Stopif(assertion, error_action, ...) {                    \
        if (assertion){ \
            fprintf(error_log ? error_log : stderr, __VA_ARGS__); \
            fprintf(error_log ? error_log : stderr, "\n");        \
            if (error_mode=='s') abort();                         \
            else                 {error_action;}                  \
        } }

    //sample usage:
    Stopif(x<0 || x>1, return -1, "x has value %g, "
                      "but it should be between zero and one.", x);
```

不管用户在省略号的位置放了什么东西，都会被插入到**VA_ARGS**标记的位置。

为了演示可变参数宏能够向我们提供多大的便利，例10-3重新改写了for循环的语法。第2个参数之后的所有东西（不管包含了多少个逗号）都将被读作...参数并粘贴到**VA_ARGS**标记的位置。

例10-3　宏的...包含了for循环的整个循环体（varad.c）

```
#include <stdio.h>

#define forloop(i, loopmax, ...) for(int i=0; i< loopmax; i++) \
                                        {__VA_ARGS__}

int main(){
    int sum=0;
    forloop(i, 10,
            sum += i;
            printf("sum to %i: %i\n", i, sum);
    )
}
```

我不会在现实世界的代码中实际使用例10-3，但是与之大体相同只在细微处存在差别的代码块却是屡见不鲜，因此有时候使用可变参数宏来消除冗余是合理的。

10.3　安全终止的列表

复合常量和可变参数宏是关系密切的伙伴，因为我们可以使用宏来创建列表和结构。我们将在稍后讨论结构的创建，首先从列表开始。

在前几页，我们看到了接受一个列表并对第1个NaN之前的所有元素进行求和的函数。使用这个函数时，并不需要知道输入列表的长度，

但是列表中必须要保证最终存在一个NaN标记。如果不存在这个标记，将会导致段错误。我们可以在调用sum时使用一个可变参数的宏来保证这个列表的尾部存在一个NaN标记，如例10-4所示。

例10-4　使用一个可变参数宏生成一个复合常量（safe_sum.c）

```
#include <math.h> //NAN
#include <stdio.h>

double sum_array(double in[]){                                  ❶
    double out=0;
    for (int i=0; !isnan(in[i]); i++) out += in[i];
    return out;
}

#define sum(...) sum_array((double[]){__VA_ARGS__, NAN})❷

int main(){
    double two_and_two = sum(2, 2);                             ❸
    printf("2+2 = %g\n", two_and_two);
    printf("(2+2)*3 = %g\n", sum(two_and_two, two_and_two, two_and_two));
    printf("sum(asst) = %g\n", sum(3.1415, two_and_two, 3, 8, 98.4));
}
```

❶ 函数名发生了变化，但它的功能与此前的数组求和函数相同。

❷ 这一行是实际操作所在：这个可变参数宏把它的输入注入一个复合常量。因此这个宏接受一个松散的double值列表，但发送给函数的却是单个列表，并保证以NAN结尾。

❸ 现在，main可以把任意长度的松散的数值列表发送给sum，让这个宏来负责添加最后的NAN标记。

现在就产生了一个现代风格的函数。它接受任意数量的输入，而不需要预先把它们包装到一个数组中，因为这个宏已经利用一个复合常量

为我们完成了这项任务。

事实上，这个宏版本只对松散的数字有效，而不适用于已经创建为数组的数据。如果已经有了一个数组，并且保证数组的尾部存在NAN，就可以直接调用sum_array。

10.4 多列表

现在如果你想发送两个任意长度的列表，该怎么办？例如，假设你决定你的程序应该用两种方法来发送错误：在屏幕上显示一个对人友好的信息，并且将一个机器可读的错误码发送到日志文件（这里使用stderr）。有一个函数可以接受像printf风格的可变长参数比两个不同的输出函数要好。但是我们如何让编译器知道一个参数的集合结束了，下一个开始了呢？

就像我们一直那样做的，我们可以使用括号对参数进行分组。以my_macor（f（a,b）,c）的方式来调用my_macro。第一个宏参数是f（a,b），括号中的逗号并不是宏参数的分隔符，因为那样会分割括号，也没什么意义。

这样我们就有了一个可以工作的例子，它可以马上打印两个错误信息。

```
#define fileprintf(...) fprintf(stderr, __VA_ARGS__)
#define doubleprintf(human, machine) do {printf human; fileprintf machine;
} while(0)

//usage:
if (x < 0) doubleprintf(("x is negative (%g)\n", x), ("NEGVAL: x=%g\n", x)
);
```

宏被扩展成：

```
do {printf ("x is negative (%g)\n", x); fileprintf ("NEGVAL: x=%g\n", x);}
while(0);
```

我把fileprintf宏加入以便提供两个语句的一致性。如果你不这么做，你必须在把给人读的参数放到括号里，而输入到日志文件的参数却不需要括号：

```
#define doubleprintf(human, ...)do{printf human;\
                                   fprintf (stderr, __VA_ARGS__);} while(0)
//and so:
if (x < 0) doubleprintf(("x is negative (%g)\n", x), "NEGVAL: x=%g\n", x);
```

这是合法的语法，但是我不喜欢这种用户界面的样式，因为对称性的事物就应该看起来是对称的。

如果用户完全忘记了括号会发生什么？它不会编译：如果你把一些内容放到printf后面，但是并没有放到括号里的话，编译器就一定会给你错误信息的。一方面，你可以得到错误信息；另一方面，你也不太可能一不小心忘记了括号，这种错误也不会发布到最后的产品中去。

给出另外一个例子，例10-5显示一个产品表：给出两列*R*和*C*，每一个单元（*i*,*j*）保持R_i和C_j的乘积。程序的核心是matrix_cross宏，以及相对友好的界面。

例10-5　发送两个变长列表给一个函数（times_table.c）

```
#include <math.h> //NAN
#include <stdio.h>

#define make_a_list(...) (double[]){__VA_ARGS__, NAN}
```

```
#define matrix_cross(list1, list2) matrix_cross_base(make_a_list list1, \
    make_a_list list2)

void matrix_cross_base(double *list1, double *list2){
    int count1 = 0, count2 = 0;
    while (!isnan(list1[count1])) count1++;
    while (!isnan(list2[count2])) count2++;

    for (int i=0; i<count1; i++){
        for (int j=0; j<count2; j++)
            printf("%g\t", list1[i]*list2[j]);
        printf("\n");
    }
    printf("\n\n");
}

int main(){
    matrix_cross((1, 2, 4, 8), (5, 11.11, 15));

    matrix_cross((17, 19, 23), (1, 2, 3, 5, 7, 11, 13));

    matrix_cross((1, 2, 3, 5, 7, 11, 13), (1)); //a column vector
}
```

10.5 Foreach

早些时候，你看到可以在使用数组或结构的任何地方使用复合常量。例如，下面是一个通过复合常量声明的字符串数组：

```
char **strings = (char*[]){"Yarn", "twine"};
```

现在，让我们将它放在一个for循环中。循环的第一部分声明了这个字符串数组，因此我们可以使用前面的代码。接着，我们遍历这个数组，直到最终的NULL标记。为了提高代码的可读性，我通过typedef定义了一个string类型：

```
#include <stdio.h>

typedef char* string;
```

```
int main(){
    string str = "thread";
    for (string *list = (string[]){"yarn", str, "rope", NULL}; *list; list
++)
        printf("%s\n", *list);
}
```

看上去仍然有些杂乱，因此我们在一个宏中隐藏所有的语法细节。这样，main函数可以简洁如下：

```
#include <stdio.h>
//I'll do it without the typedef this time.

#define Foreach_string(iterator, ...)\
    for (char **iterator = (char*[]){__VA_ARGS__, NULL}; *iterator; iterat
or++)

int main(){
    char *str = "thread";
    Foreach_string(i, "yarn", str, "rope"){
        printf("%s\n", *i);
    }
}
```

10.6 函数的向量化

free函数接受并只能接受1个参数。因此，我们常常需在一个函数的末尾执行一长串的清理工作，类似下面这样的格式：

```
free(ptr1);
free(ptr2);
free(ptr3);
free(ptr4);
```

这是非常令人厌烦之事！任何有自尊心的LISP程序员都不会允许这样的冗余写法存在，他们会编写像下面这样的向量化的free函数：

```
free_all(ptr1, ptr2, ptr3, ptr4);
```

如果读者是从头阅读本书的，那么应该可以完全理解下面这些句子：我们可以编写一个可变参数宏，通过复合常量生成一个数组（以一个终止符结尾），然后运行一个for循环，把这个函数应用于这个数组中的每一个元素。例10-6对这些思路进行了汇总。

例10-6　把接受任何指针类型的参数的任何函数进行向量化的机制（vectorize.c）

```
#include <stdio.h>
#include <stdlib.h> //malloc, free

#define Fn_apply(type, fn, ...) { \
    void *stopper_for_apply = (int[]){0};                              \❶
    type **list_for_apply = (type*[]){__VA_ARGS__, stopper_for_apply}; \❷
    for (int i=0; list_for_apply[i] != stopper_for_apply; i++)    \
         fn(list_for_apply[i]);                                        \
}

#define Free_all(...) Fn_apply(void, free, __VA_ARGS__);

int main(){
    double *x= malloc(10);
    double *y= malloc(100);
    double *z= malloc(1000);

    Free_all(x, y, z);
}
```

❶ 为了提高安全性，这个宏接受一个类型名。我把它放在函数名的前面，因为在函数声明中，先类型后名字的顺序比较容易记忆。

❷ 我们需要一个终止符，保证不与任何使用中的指针匹配，包括任何NULL指针。因此，我们使用复合常量形式定义一个包含整数的数组，并定义一个指向这个数组的指针。注意，for循环的终止条件所观察

的是指针本身，而不是它所指向的东西。

有了这个机制之后，我们可以使用这个向量化的宏包装任何接受一个指针的函数。对于GSL，我们可以定义：

```
#define Gsl_vector_free_all(...) \
        Fn_apply(gsl_vector, gsl_vector_free, __VA_ARGS__);
#define Gsl_matrix_free_all(...) \
        Fn_apply(gsl_matrix, gsl_matrix_free, __VA_ARGS__);
```

我们仍然会得到编译时的类型检查（除非把指针类型设置为void），这样就保证了这个宏的输入是相同类型的指针列表。为了接受一组多个不同的元素，就需要进一步的技巧，即指定的初始化器。

10.7 指定的初始化器

我打算通过例子来定义这个术语。下面是个简短的程序：在屏幕上打印出3×3的网格，每个点用星号表示。我们想要指定星号应该出现在右上角还是左中央，或者通过一个direction_s结构来设置。

例10-7的重点是在main函数中，我们使用指定的初始化器声明了3个结构，即在初始化器中指定每个结构成员的名称。

例10-7　使用指定的初始化器指定结构成员（boxes.c）

```
#include <stdio.h>

typedef struct {
    char *name;
    int left, right, up, down;
} direction_s;

void this_row(direction_s d); //these functions are below
```

```
void draw_box(direction_s d);

int main(){
    direction_s D = {.name="left", .left=1};              ❶
    draw_box(D);

    D = (direction_s) {"upper right", .up=1, .right=1};  ❷
    draw_box(D);

    draw_box((direction_s){});                            ❸
}

void this_row(direction_s d){                             ❹
    printf( d.left ? "*..\n"
            : d.right ? "..*\n"
            : ".*.\n");
}

void draw_box(direction_s d){
    printf("%s:\n", (d.name ? d.name : "a box"));
    d.up               ? this_row(d) : printf("...\n");
    (!d.up && !d.down) ? this_row(d) : printf("...\n");
    d.down             ? this_row(d) : printf("...\n");
    printf("\n");
}
```

❶ 这是我们的第1个指定的初始化器。由于并未指定.right、.up和.down，因此它们被初始化为0。

❷ 名称出现在最前面看上去很自然，因此我们把它作为第1个初始化器，即使不使用名称，也不会产生歧义。

❸ 这是极端情况，所有元素都被初始化为0。

❹ 这行之后的所有代码是在屏幕上打印一个格子网格，因此不存在需要解释的地方了。

填充结构的旧式方法是记住结构元素结构成员的顺序，并在不使用成员标签的情况下对所有元素进行初始化，因此不带标签的upright声明

将是：

```
direction_s upright = {NULL, 0, 1, 1, 0};
```

这种方法很不容易理解，并使很多人恨上了C。除了那些元素顺序确实非常自然而明显的极为罕见的情况之外，应该考虑丢弃这种不带标签的形式。

- 读者有没有注意到，在创建upper right结构时，相对于结构声明中的顺序，我在指定元素时所采用的顺序并不一致。要求记住任意有序集合的元素顺序实在是太过分了，应该由编译器来完成这项任务。
- 未声明的元素被初始化为0。不会留下没有被定义的元素。
- 我们可以混合指定的初始化值和未指定的初始化值。在例10-7中，让名字出现在最前面显得非常自然（像"upper right"这样的字符串并不是整数），因此当名字并没有加上显式的标签时，这种声明仍然是很容易被理解的。规则是让编译器按照顺延原则处理剩余的值。

```
typedef struct{
    int one;
    double two, three, four;
} n_s;

n_s justone = {10, .three=8}; //10 with no label gets dropped into
                              //the first slot: .one=10
n_s threefour={.two=8, 3, 4}; //By the pick up where you left off rule, 3
gets put in
                              //the next slot after .two: .three=3 and .fo
ur=4
```

- 我已经介绍了对数组使用复合常量。结构与数组相似，只不过它的

元素具有名称，并且元素的长度不一。我们也可以把复合常量应用于结构，就像在示例代码中对upper right和center结构所操作的那样。和以前一样，我们需要在花括号前添加一个类型转换（typename）。main中的第1个例子是个直接声明，因此并不需要复合常量初始化器语法，后面那个赋值通过复合常量创建了一个匿名结构，并把这个匿名结构复制到D，然后把它发送到一个函数。

自己动手：重写自己的代码中的每个结构的声明，使用指定的初始化器。我的意思就是这样。旧式的不带标签的初始化值方法是非常恐怖的。另外还要注意把下面这样的写法：

```
direction_s D;

D.left = 1;

D.right = 0;

D.up = 1;

D.down = 0;
```

改写为下面的语句

```
direction_s D = {.left=1, .up=1};
```

10.8 用零初始化数组和结构

如果你在一个函数内部声明了一个变量，C并不会自动把它初始化为零（对于自动变量而言，这个行为显得有点奇怪）。我猜测这种机制的基本原理是提高速度：在创建函数帧时，清空所有字节是非常耗时

的，如果达上百万次调用这个函数，累积的时间是相当可观的，要知道这是早在1985年的时候。

但是到了当今这个时代，让变量处于未定义状态只会带来麻烦。

对于简单的数值数据，在声明变量的代码行将它初始化为零。对于指针（包括字符串），把它设置为NULL。只要能够记住（好的编译器会在你使用一个未初始化的变量时发出警告），这类操作是极为简单的。

对于具有常量长度的结构和数组，我只想说明，如果使用了指定初始化器但保留一些元素为空，这些空白的元素将被自动设置为0。因此我们可以通过对其赋一个完全的空白值，从而把整个结构设置为零。下面这个什么也不做的程序就演示了这个概念：

```
typedef struct {
    int la, de, da;
} ladeda_s;

int main(){
    ladeda_s emptystruct = {};
    int ll[20] = {};
}
```

这是不是既轻松又贴心？

现在要面对令人痛心的那一面了：假设有一个可变长度的数组（即长度由一个运行时变量所设置），把所有元素清空为零的唯一方法就是通过memset函数：

```
int main(){
    int length=20;
```

```
    int ll[length];
    memset(ll, 0, 20*sizeof(int));
}
```

这个要比初始化固定长度的数组麻烦一些，但只能如此。

> 自己动手：为自己编写一个宏，声明一个可变长度的数组，并把它的所有元素设置为零。在输入中需要列出类型、名称和长度。

对于稀疏但并不完全为空的数组，可以使用指定初始化器来完成初始化：

```
//By the pick up where you left off rule, equivalent to {0, 0, 1.1, 0, 0, 2.2, 3.3}:
double list1[7] = {[2]=1.1, [5]=2.2, 3.3}
```

10.9 typedef可以化繁为简

指定初始化器使结构焕发新生。既然结构的用法不再给我们带来痛苦，本章的剩余部分在很大程度上就是重新考虑结构能够为我们带来什么。

但是首先，我们必须声明结构的格式。下面是我所使用的一个格式示例：

```
typedef struct newstruct_s {
    int a, b;
    double c, d;
} newstruct_s;
```

它声明了一个新类型（newstruct_s），正好是一个给定格式（struct newstruct_s）的结构。我们经常可以发现许多作者对结构标签和typeded使用了不同的名称，例如typedef struct _nst { ... } newstruct_s;。这种做法是不必要的：结构标签具有与其他标识符不同的独立名字空间[K&R第2版，§A8.3（第213页）；C99和C11，§6.2.3（1）]，因此并不会使编译器产生歧义。我发现重复的名称并不会给阅读代码的人造成歧义，而且省去了发明另一个命名的麻烦。

警告

POSIX标准保留了以_t结尾的名字作为未来可能添加到标准的类型。站在正式的角度，C标准只保留了int..._t和unit..._t，但每个新标准通过可选的头文件允许保留所有以_t结尾的新类型。很多人丝毫没有顾及他们的代码可能在数年后C22问世时发生潜在的名字冲突，仍然频繁地使用_t结尾的名字。在本书中，结构的名字以_s结尾。

我们可以通过两种方法声明这种类型的结构：

```
newstruct_s ns1;
struct newstruct_s ns2;
```

只有少数几种情况下应该使用struct newstruct_s，而不是简单的newstruct_s。

- 如果一个结构有个元素的类型就是这个结构本身（例如链表结构中

的next指针就指向另一个链表结构）。例如：

```
typedef struct newstruct_s {
    int a, b;
    double c, d;
    struct newstruct_s *next;
} newstruct_s;
```

- 在C11标准中，匿名结构要求使用struct newstruct_s格式，出自10.1.2“C，更少的缝隙”一节。
- 有些人只是喜欢使用struct newstruct_s这种格式，所以有必要讨论一下程序风格的问题了。

风格说明

我非常惊讶地发现有些人觉得typedef只会带来混淆。例如，来自Linux内核风格文件的说法：“当你在源代码中看到vps_t a;时，能知道它表示什么吗？反之，如果你看到的是struct virtual_container *a;，就可以明白a到底是什么东西。”对这种说法的自然回应是：使代码变得清晰的是后面那个更长的名称（特别是以container结尾），而不是前面所加的struct这个单词。

但是对typedef的这种反感肯定是有原因的。进一步的调查发现了一些使用typedef来定义不同的单位。例如：

```
typedef double inches;
typedef double meters;

inches length1;
meters length2;
```

现在，当我们每次使用inches时，都必须查阅它到底是什么东西？

（是unsigned int？还是doulbe？）它甚至没有提供任何的错误保护。上百行代码之后，当我们进行下面的赋值：

```
length1 = length2;
```

我们已经忘记了相关的类型声明，典型的C编译器并不会把这种做法看成错误的。如果需要关注单位，可以把它们与变量名连在一起，这样可以清楚地发现错误：

```
double length1_inches, length2_meters;

//100 lines later:

length1_inches = length2_meters; //this line is self-evidently wrong.
```

对于那些具有全局性质并且它们的内部细节应该被用户所知的结构，像使用其他全局元素一样尽可能地使用typedef还是合理的，因为查找它们的声明就像查找一个变量的声明一样让人心烦，加上struct 的同时会增加用户的认知负担。

这就是说，很难找到不是严重依赖于使用typedef声明的全局结构的产品库，像GSL的gsl_vectors和gsl_matrixes，或者GLib的散列、树等大量其他对象。即使是Linus Torvalds所编写的作为Linux内核的版本控制系统的Git源代码，也存在一些用typedef精心声明的结构。

另外，typedef的作用域与其他任何声明的作用域是相同的。这意味着我们可以在单个文件中用typedef声明一些东西，这样就不必担心它们与这个文件之外的名字空间发生混淆。我们甚至可以找到理由让typedef发生在单个函数的内部。我们可能已经注意到，到目前为止的绝大多数typedef声明都是局部的，这意味着读者向前浏览数行就可以查到它们的

定义。当typedef声明是全局性质的时候，它们总是设法隐藏在一个包装器中，这意味着读者完全没有必要查找它们的定义。因此，我们在使用结构时可以不造成任何认知负担。

10.10 从函数返回多个数据项

数学函数并不一定要映射到一维空间。例如，映射到一个2D点（*x*, *y*）的函数是很常见的函数。

Python允许我们使用列表返回多个值，类似：

```
#Given the standard paper size name, return its width, height
def width_length(papertype):
    if (papertype=="A4"):
        return [210, 297]
    if (papertype=="Letter"):
        return [216, 279]
    if (papertype=="Legal"):
        return [216, 356]

[a, b] = width_length("A4");
 print("width= %i, height=%i" %(a, b))
```

在C中，我们总是可以返回一个结构，因此可以根据需要返回许多子元素，这也是我在前面称赞随用随弃的结构的原因：从函数中返回特定的结构并不是件复杂的事情。

让我们面对这个问题：相对于提供了语法支持返回列表的语言，C在这方面的语法显然有点啰嗦。但是，正如例10-8所演示的那样，清楚地表示一个函数返回一个R2形式的值并不是不可能的。

例10-8　如果需要从一个函数中返回多个值，可以返回一个结构

（papersize.c）

```
#include <stdio.h>
#include <strings.h> //strcasecmp (from POSIX)
#include <math.h>    //NaN

typedef struct {
    double width, height;
} size_s;

size_s width_height(char *papertype){
  return
    !strcasecmp(papertype, "A4")    ? (size_s) {.width=210, .height=297}
  : !strcasecmp(papertype, "Letter")? (size_s) {.width=216, .height=279}
  : !strcasecmp(papertype, "Legal")  ? (size_s) {.width=216, .height=356}
                                     : (size_s) {.width=NAN, .height=NAN};
}

int main(){
    size_s a4size = width_height("a4");
    printf("width= %g, height=%g\n", a4size.width, a4size.height);
}
```

提示

这个代码例子使用了condition? iftrue：else这种格式，这是单个表达式，因此可以出现在return的后面。注意一系列的缩进是怎么清晰地表示一系列的case的（包括最后一条捕捉所有其他情况的分句）。我喜欢把这类代码格式化为一个清晰的小表格，当然也会有人觉得这种是可怕的风格。

另一种方法是使用指针，这也是极为常见的，并不会被看成不良的形式。但是，它确实混淆了什么是输入以及什么是输出，使得具有额外

typedef声明的版本看上去非常庞大：

```
//Return height and width via pointer:
void width_height(char *papertype, double *width, double *height);

//or return width directly and height via pointer:
double width_height(char *papertype, double *height);
```

报告错误

Pete Goodliffe讨论了从一个函数返回一个错误代码的各种方法，它似乎对于这些选项都不是特别满意。

- 在有些情况下，返回值可能是某个特定的信号值，例如表示整数的−1 或者表示浮点数的NaN（但变量的整个范围都是合法也是足够常见的）。
- 我们可以设置一个全局性的错误标志，但在2006年，Goodliffe还无法推荐使用C11的_Thread_local关键字，允许多个线程在并行运行时能够让这个错误标志正确地发挥作用。尽管一般情况下无法使用一个表示程序错误的全局标志，但是可以根据一个具有_Thread_local文件作用域的变量，合理地编写一小组紧密协作的函数。
- 第3个选项是“返回一个复合数据类型（或数对），既包含返回值也包含错误代码。这种方法在流行的C类语言中显得笨拙，因此极少被使用”。

到本书的这个节点，读者应该已经理解从函数返回一个结构的许多优点，并且现代的C提供了大量的工具（typedef、指定的初始化值）消除了原先所存在的一些笨拙操作。

提示

当我们编写一个新的结构时，要考虑添加一个错误元素或状态元素。这样的话，当这个新结构从一个函数返回的任何时候，我们就可以通过一种内置的通信方法了解这个结构是否是可用的了。

例10-9把物理公式转变为一个检查函数，以回答下面这个问题：假设一个具有特定质量的理想物体处于自由落体状态，并持续了特定的秒数，求它的能量。

我这里使用了很多宏，因为我发现C程序员更喜欢编写用于错误处理的宏，这也许是因为没人希望让错误检查代码占据大部分的主程序的缘故。

例10-9　如果函数需要返回一个值和一个错误，可以使用一个结构来完成这个任务（errortuple.c）

```
#include <stdio.h>
#include <math.h> //NaN, pow

#define make_err_s(intype, shortname)   \          ❶
    typedef struct {                    \
        intype value;                   \
        char const *error;              \          ❷
    } shortname##_err_s;

make_err_s(double, double)
make_err_s(int, int)
```

```
make_err_s(char *, string)

double_err_s free_fall_energy(double time, double mass){
    double_err_s out = {}; //initialize to all zeros.
    out.error = time < 0      ? "negative time"         ❸
                : mass < 0    ? "negative mass"
                : isnan(time) ? "NaN time"
                : isnan(mass) ? "NaN mass"
                              : NULL;
    if (out.error) return out;                          ❹

    double velocity = 9.8*time;
    out.value = mass*pow(velocity, 2)/2.;
    return out;
}

#define Check_err(checkme, return_val) \                ❺
    if (checkme.error) {fprintf(stderr, "error: %s\n", checkme.error); ret
urn return_val;}

int main(){
    double notime=0, fraction=0;
    double_err_s energy = free_fall_energy(1, 1);   ❻
    Check_err(energy, 1);
    printf("Energy after one second: %g Joules\n", energy.value);

    energy = free_fall_energy(2, 1);
    Check_err(energy, 1);
    printf("Energy after two seconds: %g Joules\n", energy.value);
    energy = free_fall_energy(notime/fraction, 1);
    Check_err(energy, 1);
    printf("Energy after 0/0 seconds: %g Joules\n", energy.value);
}
```

❶ 如果你喜欢返回一个“值/错误”结构的话，那么对于每种类型都需要这样一个结构。因此我觉得最好还是编写一个宏，可以为每种基本类型生成一种结构类型。参见前面的用法以生成double_err_s、int_err_s和string_err_s。如果你觉得这种做法有过于设计的嫌疑，那就不使用它好了。

❷ 为什么让错误成为一个字符串而不是一个整数呢？错误信息一

般情况是个常量字符串，因此在内存管理方面不会出现问题，同时也没人喜欢使用含义不清的枚举值。关于这方面的讨论，参见第7章的7.4“枚举和字符串”一节。

❸ 另一个返回值的表格。这种类型的东西对于函数执行之前的输入检查是极为常用的。注意out.error元素指向所列出的其中一个文字字符串。由于没有进行字符串的复制，因此不会分配或释放内存。为进一步证明这一点，我把error指针定义为char const。

❹ 或者，使用第2章的2.4“错误检查”一节的Stopif宏：Stopif（out.error, return out, out.error）。

❺ 用宏在返回时检查错误是一种常见的C惯用法。由于这个错误是个字符串，因此这个宏可以直接把它打印到stderr（或者打印到一个错误日志文件）。

❻ 它的用法正如我们所预期的那样。代码的作者常常抱怨用户太容易忽略函数所返回的错误代码了。把输出值放在一个结构中可以起到很好的提醒作用，表明输出中包含了一个错误代码，函数的用户应该对此加以重视。

10.11 灵活的函数输入

变参函数是一种接受可变数量的输入的函数。最著名的例子是printf函数，printf（"Hi."）和printf（"%f %f %i\n", first, second, third）都是合法的，尽管前者只有1个输入而后者具有4个输入。

简而言之，C的变参函数提供了正好足够的能力来实现printf，仅此而已。我们必须有个初始的固定参数，期望第1个参数或多或少能够提供后续元素的类型目录，至少能够提供参数的数量。在前面的例子中，第1个参数（"%f %f %i \n"）表示接下来的两个数据项应该是浮点变量，最后一个则是整数。

这里不存在类型安全：如果实际传递了一个像1这样的int值，却以为自己传递了一个像1.0这样的float值时，其结果是未定义的。如果函数期望接受3个参数，但实际上只提供了2个，很可能会产生一个段错误。由于类似这样的问题，CERT（计算机安全应急响应组）认为变参函数存在安全风险（严重性：高。可能性：很可能）[1]。

前面，我们介绍了提供安全性的第1种方法：编写一个包装器宏，在列表的尾部添加一个终止标志，这样就可以保证底层的函数不会接受一个永远不会终止的列表。复合常量也会检查输入类型，如果发送了错误类型的输入，就无法通过编译。

本节还将讨论另外两种实现具有一些类型检查安全性的变参函数的方法。最后一个方法允许我们对参数进行命名，这可以减少出错的机会。我同意CERT关于自由形式的变参函数风险过大的说法，因此只在自己的代码中使用本书介绍的变参函数的实现方式。

本节所讨论的第1种安全格式用到了编译器对printf的检查，我们只是对已经熟悉的形式进行扩展。第2种格式使用了一个可变参数宏对输入进行预处理，在函数头部使用指定初始化值语法。

10.11.1 把函数声明为printf风格

首先，我们遵循传统的路线，使用C89的变参函数功能。介绍这种方法的原因是我们在有些场合下可能无法使用宏，这往往是由社会原因而不是技术原因所导致的。只有很少的情况无法使用可变参数宏代替变参函数。

为了保证C89的变参函数的安全性，我们将使用C99所增加的一个特性：**attribute**允许使用编译器特殊的功能[2]。

```
#include "config.h"
#ifndef HAVE__ATTRIBUTE__
#define __attribute__(...)
#endif
```

它能用在变量、结构或函数的声明行（如果函数在使用之前还没有被声明，就需要先进行声明），

gcc和Clang允许我们设置一个属性，把一个函数声明为printf的风格，这意味着编译器将进行类型检查，并在需要发送一个double值而实际发送的是int或double*值时发出警告。

假设我们需要system的一个版本允许printf风格的输入。在例10-10中，system_w_printf函数接受printf风格的输入，把它们写入到一个字符串，并发送到标准system命令中。这个函数使用了vasprintf这个能够支持va_list的asprintf相容版本。它们都是BSD/GNU标准的函数。如果需要坚持遵循C99，可以用类似snprintf的vsnprintf来代替它们（这时需要加入#include <stdarg.h>）。

main函数中只有一个简单的示例用法：它从命令行接受第1个输入，并在此基础上运行ls。

例10-10　处理可变长度的输入的旧式方法（olden_varargs.c）

```
#define _GNU_SOURCE //cause stdio.h to include vasprintf
#include <stdio.h> //printf, vasprintf
#include <stdarg.h> //va_start, va_end
#include <stdlib.h> //system, free
#include <assert.h>

int system_w_printf(char const *fmt, ...) __attribute__((format (printf,1,
2))); ❶

int system_w_printf(char const *fmt, ...){
    char *cmd;
    va_list argp;
    va_start(argp, fmt);
    vasprintf(&cmd, fmt, argp);
    va_end(argp);
    int out= system(cmd);
    free(cmd);
    return out;
}

int main(int argc, char **argv){
    assert(argc == 2);                                          ❷
    return system_w_printf("ls %s", argv[1]);
}
```

❶ 标记为类似printf的函数，其中第1个输入是格式指示符，其他参数列表是从第2个输入开始的。

❷ 我承认自己在这里偷懒了。应该只是在程序设计师能完全控制的内部使用assert检查，而不是检查用户的输入。关于适用于输入测试的宏，参见第2章2.4“错误检查”一节。

这种方法优于可变参数宏之处在于从宏获取返回值的方式是相当笨拙的。但是，例10-11的宏版本更短更简单，如果编译器会对printf族系的函数执行类型检查，它在这里也会如此（不需要任何gcc/Clang特定的属性）。

例10-11　宏版本更简短（macro_varargs.c）

```
#define _GNU_SOURCE //cause stdio.h to include vasprintf
#include <stdio.h>  //printf, vasprintf
#include <stdlib.h> //system
#include <assert.h>

#define System_w_printf(outval, ...) {          \
    char *string_for_systemf;                   \
    asprintf(&string_for_systemf, __VA_ARGS__); \
    outval = system(string_for_systemf);        \
    free(string_for_systemf);                   \
}

int main(int argc, char **argv){
    assert(argc == 2);
    int out;
    System_w_printf(out, "ls %s", argv[1]);
    return out;
}
```

10.11.2　可选参数和命名参数

我们已经介绍了怎样通过复合常量加上可变参数宏更清晰地向一个函数发送一个相同类型参数的列表。如果对这个概念仍然比较模糊，可以参阅本章10.3“安全终止的列表”一节。

在许多情况下结构就像数组一样，只不过它保存的是不同类型的数据。因此，我们似乎可以采用相同的套路：编写一个包装器宏，用简洁的界面把所有的元素包装到一个结构中，然后把整个结构发送给函数。例10-12就是这样做的。

在这个例子中创建了一个函数，接受可变数量的命名参数。这个函数的定义分为3个部分：一个随用随弃的结构，用户绝不会通过名称来引用它（但是如果这个函数将成为全局函数，这个结构也必须在全局空

间中定义）；把参数插入到一个结构（这个结构将被传递给底层函数）的宏；底层函数。

例10-12　一个接受可变数量的命名参数的函数，不是用户设置的参数具有默认值（ideal.c）

```
#include <stdio.h>

typedef struct {                                                    ❶
    double pressure, moles, temp;
} ideal_struct;

/** Find the volume (in cubic meters) via the ideal gas law: V =nRT/P
Inputs:
pressure in atmospheres (default 1)
moles of material (default 1)
temperature in Kelvins (default freezing = 273.15)
   */
#define ideal_pressure(...) ideal_pressure_base((ideal_struct){.pressure=1
,\❷
                                    .moles=1, .temp=273.15, __VA_ARGS__})
double ideal_pressure_base(ideal_struct in){                        ❸
    return 8.314 * in.moles*in.temp/in.pressure;
}

int main(){
    printf("volume given defaults: %g\n", ideal_pressure() );
    printf("volume given boiling temp: %g\n", ideal_pressure(.temp=373.15)
); ❹
    printf("volume given two moles: %g\n", ideal_pressure(.moles=2) );
    printf("volume given two boiling moles: %g\n",
                                ideal_pressure(.moles=2, .temp=373.15));
}
```

❶ 首先，我们需要声明持有函数的输入结构。

❷ 这个宏的输入参数将被插入匿名结构，用户在括号中所设置的值将用作指定初始化器。

❸ 函数本身接受一个结构类型（ideal_struct）参数，而不是通常的自由输入列表。

❹ 用户输入一串指定初始化器，未列出的参数将得到一个默认值，然后ideal_pressure_base将从输入结构中取得所需要的数据。

最后一行的函数调用（不要告诉用户它实际上是个宏）是这样展开的：

```
ideal_pressure_base((ideal_struct){.pressure=1, .moles=1, .temp=273.15,
                                                .moles=2, .temp=373.15})
```

规则是如果一个数据项被多次初始化，则以最后一次初始化为准。因此，.pressure被保留为默认值1，而另两个输入被设置为用户指定的值。

警告

如果使用-Wall选项，Clang就会对moles和temp的重复初始化发出警告，因为编译器作者认为重复的初始化更可能是出于错误，而不是对默认值的故意选择。在编译器开关中添加-Wno-initializer-overrides选项可以关掉这个警告。gcc只有在要求-Wextra警告的情况下才会把这个行为标识为错误。如果想使用这个选项，可以打开-Wextra -Woverride-init。

自己动手：在这个例子中，这个随用随弃的结构可能并不是一次性的，因为很可能需要在多个方向使用这个公式：

- pressure = 8.314 moles * temp/volume。
- moles = pressure *volume /（8.314 temp）。
- temp = pressure *volume /（8.314 moles）。

改写这个结构，添加一个volume元素，并使用这个结构编写函数，实现这些额外的公式。

然后利用这些函数产生一个统一的函数，它通过结构指定pressure、moles、temp和volume（第4个可以是NaN，或者针对结构添加一个what_to_solve元素）其中的3个成员，并应用正确的函数计算出第4个值。

既然参数是可选的，我们可以在半年后再添加一个新的参数，而不会破坏期间使用了这个函数的每个程序。我们可以很方便地从一个简单的工作函数开始，并根据需要创建额外的特性。但是，我们应该从一开始就记住一个教训：这种做法很容易失控，很可能会创建具有十几个甚至几十个输入的函数，而每个输入仅仅用于处理1～2个罕见的情况。

10.11.3 使无聊的函数焕发光彩

目前，我们所讨论的例子重点在于演示简单的程序结构，我们并没有介绍很多细节。但是，如果要解决现实中的问题，就必须把所有的东西集成为一个实用且健壮的程序，毕竟简短的示例程序显然无法满足所有的需求。因此，从现在开始的例子的篇幅会更长一点，包括更多现实

的考虑。

例10-13是个不那么让人愉快的无聊函数。对于一项分期偿还的贷款，每月的还款额是固定的，但是还款中利息所占的百分比在一开始时（当绝大部分贷款尚未偿还时）非常巨大，但在贷款趋向完成时逐渐减少为零。它所涉及的数学计算是乏味的（尤其是当我们增加了选项，可以增加每月所支付的本金，或者可以提前完成贷款的支付），如果读者忽略了这个函数的具体计算部分，也是可以原谅的。我们在这里所关注的是这个函数的接口，它接受10个可以按照任何顺序出现的输入。使用这个函数进行任何类型的金融查询不仅是件痛苦的事情，而且很容易产生错误。

也就是说，amortize函数看上去极像C世界里所漂荡的许多历史遗留函数。它最明显的特点就是完全无视读者的感受，颇有点朋克摇滚的感觉。因此，按照遍地开花的时装杂志的风格，这个复杂的函数必须用一个良好的包装器进行“整容”。如果存在历史遗留代码，我们就无法更改这个函数的接口（其他程序可能依赖于此）。因此在上面那个用于生成命名的可选参数的理想气体例子的基础之上，我们将需要添加一个预备函数，在宏输出和固定的遗留函数输入之间架起一座桥梁。

例10-13　一个难以使用的函数，它具有太多的输入并且没有进行错误检查（amortize.c）

```
#include <math.h> //pow.
#include <stdio.h>
#include "amortize.h"

double amortize(double amt, double rate, double inflation, int months,
                int selloff_month, double extra_payoff, int verbose,
                double *interest_pv, double *duration, double*monthly_paym
```

```
ent){
    double total_interest = 0;
    *interest_pv = 0;
    double mrate = rate/1200;

    //The monthly rate is fixed, but the proportion going to interest chan
ges.
    *monthly_payment = amt * mrate/(1-pow(1+mrate, -months)) + extra_payof
f;
    if (verbose) printf("Your total monthly payment: %g\n\n", *monthly_pay
ment);
    int end_month = (selloff_month && selloff_month < months )
                           ? selloff_month
                           : months;
    if (verbose) printf("yr/mon\t Princ.\t\tInt.\t| PV Princ.\t PV Int.\t
Ratio\n");
    int m;
    for (m=0; m < end_month && amt > 0; m++){
        double interest_payment = amt*mrate;
        double principal_payment = *monthly_payment - interest_payment;
        if (amt <= 0)
            principal_payment =
            interest_payment = 0;
        amt -= principal_payment;
        double deflator = pow(1+ inflation/100, -m/12.);
        *interest_pv += interest_payment * deflator;
        total_interest += interest_payment;
        if (verbose) printf("%i/%i\t%7.2f\t\t%7.2f\t| %7.2f\t %7.2f\t%7.2f
\n",
              m/12, m-12*(m/12)+1, principal_payment, interest_payment,
              principal_payment*deflator, interest_payment*deflator,
              principal_payment/(principal_payment+interest_payment)*100);
    }
    *duration = m/12.;
    return total_interest;
}
```

例10-14和例10-15为amortize函数创建了一个用户友好的接口。头文件的大部分采用了Doxygen风格的文档，因为在具有这么多输入的情况下，不采用文档的形式将它们全部记录下来显然是愚蠢的。我们必须告诉用户默认值将是什么，用户是否应该省略某个输入。

例10-14　头文件（大多具有文档描述）添加了一个宏和一个预备

函数的头部（amortize.h）

```
double amortize(double amt, double rate, double inflation, int months,
            int selloff_month, double extra_payoff, int verbose,
            double *interest_pv, double *duration, double *monthly_payment
);

typedef struct {
    double amount, years, rate, selloff_year, extra_payoff, inflation; ❶
    int months, selloff_month;
    _Bool show_table;
    double interest, interest_pv, monthly_payment, years_to_payoff;
    char *error;
} amortization_s;

/** Calculate the inflation-adjusted amount of interest you would pay ❷
  over the life of an amortized loan, such as a mortgage.

\li \c amount The dollar value of the loan. No default--if unspecified,
              print an error and return zeros.
\li \c months The number of months in the loan. Default: zero, but see yea
rs.
\li \c years If you do not specify months, you can specify the number of
             years. E.g., 10.5=ten years, six months.
             Default: 30 (a typical U.S. mortgage).
\li \c rate  The interest rate of the loan, expressed in annual
             percentage rate (APR). Default: 4.5 (i.e., 4.5%), which
             is typical for the current (US 2012) housing market.
\li \c inflation The inflation rate as an annual percent, for calculating
                 the present value of money. Default: 0, meaning no
                 present-value adjustment. A rate of about 3 has been typic
al
                 for the last few decades in the US.
\li \c selloff_month At this month, the loan is paid off (e.g., you resell
                     the house). Default: zero (meaning no selloff).
\li \c selloff_year If selloff_month==0 and this is positive, the year of
                    selloff. Default: zero (meaning no selloff).
\li \c extra_payoff Additional monthly principal payment. Default: zero.
\li \c show_table If nonzero, display a table of payments. If zero, displa
y
                  nothing (just return the total interest). Default: 1

All inputs but \c extra_payoff and \c inflation must be nonnegative.

\return an \c amortization_s structure, with all of the above values set a
s
```

```
       per your input, plus:

\li \c interest Total cash paid in interest.
\li \c interest_pv Total interest paid, with present-value adjustment for
inflation.
\li \c monthly_payment The fixed monthly payment (for a mortgage, taxes an
d
                        interest get added to this)
\li \c years_to_payoff Normally the duration or selloff date, but if you m
ake early
                          payments, the loan is paid off sooner.
\li \c error              If <tt>error != NULL</tt>, something went wrong an
d the results
                       are invalid.
*/
#define amortization(...) amortize_prep((amortization_s){.show_table=1, \
                                                          __VA_ARGS__})❸
amortization_s amortize_prep(amortization_s in);
```

❶ 宏所使用的这个结构用于把数据传输给预备函数。它的作用域必须与这个宏和预备函数本身相同。有些元素是amortize函数中不存在的输入元素，但是这样可以使用户变得轻松。有些元素是需要被填充的输出结果。

❷ Doxygen格式的文档。当文档占据了接口文件的大部分空间时，它的价值就非常大。注意每个输入都列出了它的默认值。

❸ 这个宏把用户的输入（很可能是类似amortization （.amount=2e6, .rate=3.0）这样的东西）填充到一个amortization_s，作为后者的指定初始化值。宏中必须把show_table设置为默认值，因为如果没有它，就没有办法区别显式地设置了.show_table=0的用户和完全省略了.show_table。因此，如果我们想让一个变量具有非零的默认值，而用户将其设置为零也是很合理的做法的时候，就必须使用这种格式。

命名参数设置的这3个组成部分是显而易见的：一个用于结构的

typedef声明；一个接受命名元素并填充结构的宏；一个接受单个结构为输入的函数。但是，这个被调用的函数是个预备函数，在宏和底层函数之间架起桥梁。它的声明出现在头文件中，它的主体部分位于例10-15中。

例10-15　接口的非公共部分（amort_interface.c）

```
#include "stopif.h"
#include <stdio.h>
#include "amortize.h"

amortization_s amortize_prep(amortization_s in){              ❶
    Stopif(!in.amount || in.amount < 0 || in.rate < 0         ❷
            || in.months < 0 || in.years < 0 || in.selloff_month < 0
            || in.selloff_year < 0,
            return (amortization_s){.error="Invalid input"},
            "Invalid input. Returning zeros.");

int months = in.months;
if (!months){
    if (in.years) months = in.years * 12;
    else          months = 12 * 30; //home loan
}

int selloff_month = in.selloff_month;
if (!selloff_month && in.selloff_year)
    selloff_month = in.selloff_year * 12;

amortization_s out = in;
out.rate = in.rate ? in.rate : 4.5;                           ❸
out.interest = amortize(in.amount, out.rate, in.inflation,
        months, selloff_month, in.extra_payoff, in.show_table,
        &(out.interest_pv), &(out.years_to_payoff), &(out.monthly_payment)
);
    return out;
}
```

❶ 这是amortize函数应该具备的预备函数：它设置智能的默认值，并检查输入错误。现在amortize函数可以直奔主题，因为所有的预备工

作都已经在这里完成。

❷ 参见第2章的2.4“错误检查”一节关于Stopif宏的讨论。按照那一节的讨论，在这一行所进行的检查更注重于防止段错误和合理性检查，而不是自动验证错误的输入。

❸ 由于rate是个简单常量，我们也可以在amortization宏中设置它，show_table也是这种情况。这些都可以自由选择。

因为我们无法直接修改amortize的接口，预备函数的直接用途是接受一个结构，并利用这个结构的元素调用amortize函数。但是，既然我们有了一个专门用于准备函数输入的函数，我们事实上可以进行错误检查和默认值设置。例如，我们现在可以向用户提供选项，指定以月或年为单位的还款期数，并使用这个预备函数在输入越界或者明显不合理时抛出错误。

对于像这样的函数，默认值尤其重要。顺便说一下，由于我们中的大多数人事实上并不知道（并且也没有太大的兴趣知道）合理的通货膨胀率是多少。如果一台计算机既可以向用户提供对方可能并不具备的领域知识，同时能够悄悄地提供默认值供用户直接使用，无疑会让用户感到满意。

amortize函数返回几个不同的值。根据10.10“从函数返回多个数据项”一节所述，把它们放在一个单独的结构中是一种漂亮的替代方案，这比让amortize返回一个值，同时通过指针传回其他值的方法要好。另外，为了让通过可变参数宏实现的指定初始化器技巧发挥作用，我们必须通过另一个结构发挥中间作用。为什么不把这两个结构组合在一起

呢？其结果是一个输出结构，它保留了所有的输入信息。

完成这一切接口界面后，我们就有了一个具有良好文档、易于使用并执行错误检查的函数，例10-16的程序可以在大量不同的场景下运行而不会出现困扰。它用了前面的amortize.c和amort_interface.c，并且前者使用了来自math库的pow函数，因此我们的makefile看上去应该类似：

```
P=amort_use
objects=amort_interface.o amortize.o
CFLAGS=-g -Wall -O3 #the usual
LDLIBS=-lm
CC=c99

$(P):$(objects)
```

例10-16　现在可以使用amortization宏/函数编写易于阅读的假设分析场景（amort_use.c）

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

int main(){
    printf("A typical loan:\n");
    amortization_s nopayments = amortization(.amount=200000, .inflation=3)
;
    printf("You flushed real $%g down the toilet, or $%g in present value.
\n",
            nopayments.interest, nopayments.interest_pv);

    amortization_s a_hundred=amortization(.amount=200000, .inflation=3,
                                          .show_table=0, .extra_payoff=100);
    printf("Paying an extra $100/month, you lose only $%g (PV), "
            "and the loan is paid off in %g years.\n",
            a_hundred.interest_pv, a_hundred.years_to_payoff);

    printf("If you sell off in ten years, you pay $%g in interest (PV).\n"
,
                    amortization(.amount=200000, .inflation=3,
```

```
                                        .show_table=0, .selloff_year=10).interest
_pv);                                                                    ❶
}
```

❶ amortization函数返回一个结构。在前两次使用时，将传回结构赋值给一个变量，并使用这个变量来存取结构成员。但是如果我们并不需要这个中间变量，就不用声明了。这行代码的作用是从函数的返回值中提取我们所需要的一个结构元素。如果这个函数返回一块由malloc分配的内存，就没有办法做到这一点，因为我们必须要通过变量才能释放分配的内存。但是需要注意的是，这一章所讨论的就是传递结构而不是指向结构的指针。

我们编写了很多行的代码，对原先的函数进行了包装。但是，这个样板结构以及用于设置命名参数的宏并不多。剩余部分包括文档以及值得增加的输入检查。从整体上说，我们已经让一个其接口几乎没办法让人使用的函数变得用户友好，就像使用分期还贷计算器那样。

10.12 void指针以及它所指向的结构

本节将讨论通用过程和通用结构的实现。本节中的一个例子把一些函数应用于一个目录层次结构中的每个文件，允许用户把文件名打印到屏幕上、搜索一个字符串或者其他想要做的事情。另一个例子将使用GLib的散列结构记录一个文件中出现的每个字符的数量，这意味着把一个Unicode字符键值与一个整数值进行关联。当然，GLib提供了一个散列结构，可以接受任何类型的键和值，因此Unicode字符计数器就是通用容器的一个应用。

所有这些功能的实现都要归功于void指针，它可以指向任何东西。

hash函数和目录处理函数对于实际被指向的值并不关心，它们只是简单地处理传递进来的值。类型安全变成了我们的责任，但是结构可以帮助我们保持类型安全并且让我们更容易编写和操作通用过程。

10.12.1 具有通用输入的函数

回调函数（函数*B*）就是被传递给另一个函数（函数*A*）以便这个函数（函数*A*）内部使用的函数（函数*B*）。接下来，我将讨论一个通用过程，对一个目录进行递归，对这个目录中所找到的每个文件执行一些操作。这个回调函数将被传递给目录遍历过程，后者将其作用于每个文件。

图10-1描述了这个问题。通过一个直接的函数调用，编译器知道了所传递的数据的类型，也知道函数所需要的数据的类型。如果它们并不匹配，编译器就会报错或发出警告。但是，通用过程不应该决定函数的格式或者函数所使用的数据。第12章的12.5“pthread”一节使用了pthread_create，它（省略了不相关的部分）可能以下面的形式被声明：

```
typedef void *(*void_ptr_to_void_ptr)(void *in);
int pthread_create(..., void *ptr, void_ptr_to_void_ptr fn);
```

如果我们执行一个类似pthread_create（..., indata, myfunc）这样的调用，则indata的类型信息就会丢失，因为它将被转换为一个void指针。我们可以期望在pthread_create中的某个地方会出现myfunc（indata）形式的调用。如果indata的类型是double*，并且myfunc接受一个char*参数，就会出现编译器无法预防的灾难。

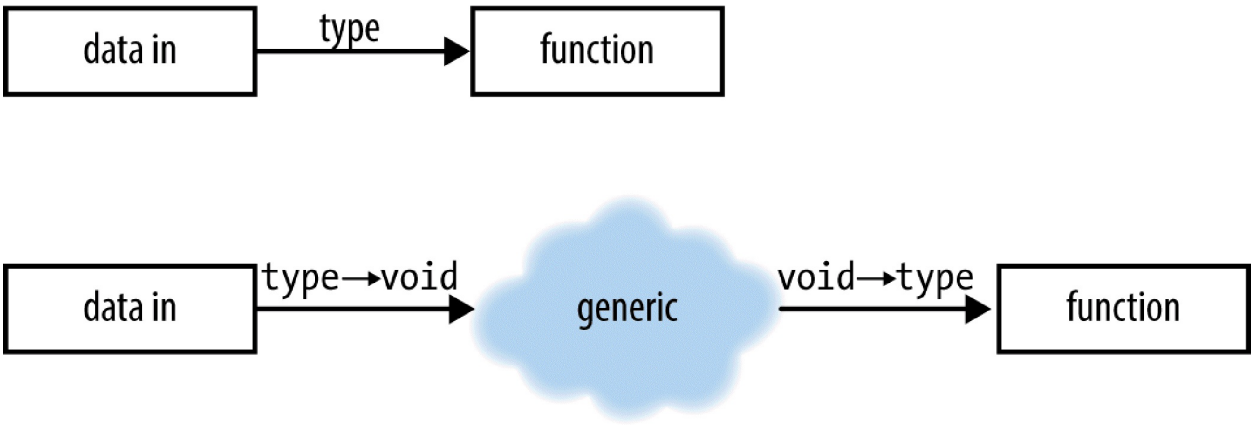


图10-1　直接调用一个函数以及让通过泛型来执行调用

例10-17是一个函数实现的头文件，这个函数把一些函数应用于一个给定目录中的每个目录和文件。它包含了说明process_dir函数行为的Doxygen文档。理所当然，这些文档的长度大致与代码的长度相同。

例10-17　一个通用的目录递归函数的头文件（process_dir.h）

```
struct filestruct;
typedef void (*level_fn)(struct filestruct path);

typedef struct filestruct{
    char *name, *fullname;
    level_fn directory_action, file_action;
    int depth, error;
    void *data;
} filestruct;                                                    ❶

/** I get the contents of the given directory, run \c file_action on each
    file, and for each directory run \c dir_action and recurse into the di
rectory.
    Note that this makes the traversal depth first.

    Your functions will take in a \c filestruct, qv. Note that there is an
 \c error
    element, which you can set to one to indicate an error.

    Inputs are designated initializers, and may include:
    \li \c .name The current file or directory name
    \li \c .fullname The path of the current file or directory
```

```
    \li \c .directory_action A function that takes in a \c filestruct.
            I will call it with an appropriately-set \c filestruct
            for every directory (just before the files in the directory
            are processed).
    \li \c .file_action Like the \c directory_action, but the function
            I will call for every non-directory file.
    \li \c .data A void pointer to be passed in to your functions.

    \return 0=OK, otherwise the count of directories that failed + errors
thrown
            by your scripts.

    Sample usage:
\code
    void dirp(filestruct in){ printf("Directory: <%s>\n", in.name); }
    void filep(filestruct in){ printf("File: %s\n", in.name); }

    //list files, but not directories, in current dir:
    process_dir(.file_action=filep);

    //show everything in my home directory:
    process_dir(.name="/home/b", .file_action=filep, .directory_action=dir
p);
\endcode
*/
#define process_dir(...) process_dir_r((filestruct){__VA_ARGS__}) ❷

int process_dir_r(filestruct level);                                    ❸
```

❶ 这里再次出现：接受命名参数的函数的3个组成部分。但即便存在取巧的地方，这个结构在传递void指针时仍能在本质上保持类型安全。

❷ 这个宏用来自用户的指定初始化器填充一个复合常量结构。

❸ 这个函数接受process_dir宏所创建的结构。用户不会直接调用这个函数。

将其与图10-1进行比较，这个头文件已经提示了类型安全问题的一个部分解决方案：定义一个确定的类型filestruct，并要求回调函数接受

这种类型的结构为参数。在这个结构的最后仍然隐藏着一个void指针。我可以把这个void指针留在这个结构的外面，就像下面这样：

```
typedef void (*level_fn)(struct filestruct path, void *indata);
```

既然总要定义一个临时的结构作为process_dir函数的助手，那么将void指针纳入结构就不会有什么问题。而且，既然我们有了一个与process_dir相关联的结构，就可以像10.11.2“可选参数和命名参数”一节所描述的那样，用它来实现“一个宏把指定的初始化器作为一个函数的输入”这种形式。结构使一切变得简单。

例10-18展示了process_dir的一种用法——在图10-1的云之前和之后的部分。这些回调函数相当简单，就是打印一些空格和文件/目录名。这个例子甚至不存在任何类型上的不安全，因为回调函数的输入被定义为一个特定类型的结构。

下面是个示例输出，一个目录具有2个文件和一个称为cfiles的子目录，后者又包含了3个文件：

```
Tree for sample_dir:
├ cfiles
└┐
    │ c.c
    │ a.c
    │ b.c
│ a_file
│ another_file
```

例10-18　一个显示当前目录结构树的程序（show_tree.c）

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

```
void print_dir(filestruct in){
    for (int i=0; i< in.depth-1; i++) printf(" ");
    printf("├ %s\n", in.name);
    for (int i=0; i< in.depth-1; i++) printf(" ");
    printf("└──┐\n");
}

void print_file(filestruct in){
    for (int i=0; i< in.depth; i++) printf(" ");
    printf("│ %s\n", in.name);
}

int main(int argc, char **argv){
    char *start = (argc>1) ? argv[1] : ".";
    printf("Tree for %s:\n", start ? start: "the current directory");
    process_dir(.name=start, .file_action=print_file, .directory_action=pr
int_dir);
}
```

正如我们所看到的那样，main函数把print_dir和print_file函数传递给process_dir，并信任process_dir将会在正确的时候调用它们，向它们提供适当的输入。

process_dir函数本身的内容在例10-19中。这个函数的大多数工作集中在根据当前被处理的文件或目录生成一个正确的结构。指定的目录通过opendir函数打开。接着对readdir函数的每次调用将提取这个目录的下一项，可能是一个文件、目录、链接或其他东西。作为输入的filestruct被当前项的信息所更新。取决于当前的目录项所描述的是目录还是文件，会以一个新准备的filestruct为参数调用适当的回调函数。如果是个目录，这个函数就用当前目录的信息执行递归调用。

例10-19　对一个目录进行递归，对所找到的每个文件应用file_action，对找到的每个目录应用directory_action（process_dir.c）

```
#include "process_dir.h"
#include <dirent.h> //struct dirent
```

```
#include <stdlib.h> //free

int process_dir_r(filestruct level){
    if (!level.fullname){
        if (level.name) level.fullname=level.name;
        else            level.fullname=".";
    }
    int errct=0;

    DIR *current=opendir(level.fullname);                          ❶
    if (!current) return 1;
    struct dirent *entry;
    while((entry=readdir(current))) {
        if (entry->d_name[0]=='.') continue;
        filestruct next_level = level;                             ❷
        next_level.name = entry->d_name;
        asprintf(&next_level.fullname, "%s/%s", level.fullname, entry->d_n
ame);

        if (entry->d_type==DT_DIR){
            next_level.depth ++;
            if (level.directory_action) level.directory_action(next_level)
;❸
            errct+= process_dir_r(next_level);
        }
        else if (entry->d_type==DT_REG && level.file_action){
             level.file_action(next_level);                        ❹
             errct+= next_level.error;
        }
        free(next_level.fullname);                                 ❺
    }
    closedir(current);
    return errct;
}
```

❶ opendir、readdir和closedir函数都是POSIX标准的函数。

❷ 对于目录中的每个项，创建输入参数filestruct的一份新复制，并适当地对它进行更新。

❸ 对于更新后的filestruct，调用针对每个目录的函数，并递归到子目录。

❹ 对于更新后的filestruct，调用针对每个文件的函数。

❺ 在每个步骤中所生成的filestruct并不是指针，不需要手工分配内存，因此它们并不需要内存管理代码。但是，asprintf会隐式分配fullname，因此必须被释放才能得到清理。

这种设置成功地实现了恰到好处的封装：打印函数并不需要关注POSIX的目录处理，process_dir.c并不知道输入函数执行了哪些操作。回调函数通过使用结构使得整个流程结合在一起。

10.12.2 通用结构

链表、散列、树和其他数据结构在各种场合中都具有使用价值，因此为它们提供void指针以处理各种类型的数据是合理的。作为用户，我们可以在进出两端对类型进行检查。

本节将展示一个典型的教科书例子：一个字符频率散列。散列是一种保存键/值对的容器，目的是允许用户通过键快速地查找对应的值。

在处理一个目录中的文件之前，我们需要对GLib库的通用散列进行自定义，使用一个Unicode键和一个保存单个整数的值，作为这个程序将要使用的形式。有了这个组件（它已经是个处理回调函数的良好例子）之后，就很容易在程序中实现用于文件遍历功能的回调函数。

正如我们将要看到的那样，equal_chars和printone函数将作为回调函数，由那些与这种散列结构相关联的函数所使用，因此这种散列结构将向这两个回调函数发送void指针。因此，这两个函数的第1行声明了正确类型的变量，有效地把void指针输入转换为一种正确的类型。

例10-20是头文件，显示了例10-21的公共用法。

例10-20　unictr.c的头文件（unictr.h）

```
#include <glib.h>
void hash_a_character(gunichar uc, GHashTable *hash);
void printone(void *key_in, void *val_in, void *xx);
GHashTable *new_unicode_counting_hash();
```

例10-21　围绕以Unicode字符为键的散列和计算各字符出现的频度的函数（unictr.c）

```
#include "string_utilities.h"
#include "process_dir.h"
#include "unictr.h"
#include <glib.h>
#include <stdlib.h> //calloc, malloc

typedef struct {                                                    ❶
    int count;
} count_s;

void hash_a_character(gunichar uc, GHashTable *hash){
    count_s *ct = g_hash_table_lookup(hash, &uc);
    if (!ct){
        ct = calloc(1, sizeof(count_s));
        gunichar *newchar = malloc(sizeof(gunichar));
        *newchar = uc;
        g_hash_table_insert(hash, newchar, ct);
    }
    ct->count++;
}

void printone(void *key_in, void *val_in, void *ignored){           ❷
    gunichar const *key= key_in;                                    ❸
    count_s const *val= val_in;
    char utf8[7];                                                   ❹
    utf8[g_unichar_to_utf8(*key, utf8)]='\0';
    printf("%s\t%i\n", utf8, val->count);
}

static gboolean equal_chars(void const * a_in, void const * b_in){ ❺
    const gunichar *a= a_in;                                        ❻
```

```
    const gunichar *b= b_in;
    return (*a==*b);
}
GHashTable *new_unicode_counting_hash(){
    return g_hash_table_new(g_str_hash, equal_chars);
}
```

❶ 是的，这是个保存了单个整数的结构。也许有一天，它会发挥至关重要的作用。

❷ 它将成为g_hash_table_foreach的回调函数，因此它将接受分别表示键和值的void指针为参数，这个函数并不需要使用第三个void指针。

❸ 如果一个函数接受一个void指针，它的第1行需要用正确的类型设置一个变量，然后把void指针转换为实用的类型。不要把这个任务拖到后面，必须在一开始就完成这个任务，因为这样可以确认进行了正确的类型转换。

❹ 6个char足以表示一个Unicode字符的任何UTF-8编码。再添加1字节表示缀尾的'\0'，7个char足以表示任何一个单字符的字符串。

❺ 由于一个散列的键和值可以是任何类型，因此GLib要求我们提供比较函数来确定两个键是否相同。后面的new_unicode_counting_hash将把这个函数发送到散列以创建函数。

❻ 我是不是提到过一个接受void指针为参数的函数应该在第1行就把这个void指针转换为正确类型的变量？如果这样做了，就可以实现类型安全。

既然已经有了一组函数支持Unicode字符的散列，例10-22就使用这

些函数以及前面的process_dir，对一个目录中的UTF-8可读的文件中的所有字符进行计数。

它使用了与前面相同的process_dir函数，因此这个通用过程以及它的用法应该并不陌生。处理单个文件的回调函数hash_a_file接受一个filestruct，但这个filestruct内部隐藏着一个void指针。这里所使用的函数使用这个void指针指向一个GLib散列结构。因此，hash_a_file函数的第1行把这个void指针转换为它所指向的结构，因此保证了类型安全。

每个组件可以单独进行调试，只要知道它的输入是什么以及在什么时候接受输入。我们也可以在组件之间对散列进行追踪，并证实它通过输入的filestruct的.data元素被发送给process_dir，并且hash_a_file再次把.data转换为一个GHashTable，然后把它发送给hash_a_character，后者将像我们前面所看到的那样对它进行修改或添加。接着，g_hash_table_foreach使用printone回调函数打印这个散列结构中的每个元素。

例10-22　一个字符频率计数器；用法：charct your_dir |sort -k 2–n（charct.c）

```
#define _GNU_SOURCE            //get stdio.h to define asprintf
#include "string_utilities.h" //string_from_file
#include "process_dir.h"
#include "unictr.h"
#include <glib.h>
#include <stdlib.h>          //free

void hash_a_file(filestruct path){
    GHashTable *hash = path.data;                                    ❶
    char *sf = string_from_file(path.fullname);
    if (!sf) return;
    char *sf_copy = sf;
    if (g_utf8_validate(sf, -1, NULL)){
```

```
        for (gunichar uc; (uc = g_utf8_get_char(sf))!='\0'; ❷
            sf = g_utf8_next_char(sf))
                hash_a_character(uc, hash);
    }
    free(sf_copy);
}

int main(int argc, char **argv){
    GHashTable *hash;
    hash = new_unicode_counting_hash();
    char *start=NULL;
    if (argc>1) asprintf(&start, "%s", argv[1]);
    printf("Hashing %s\n", start ? start: "the current directory");
    process_dir(.name=start, .file_action=hash_a_file, .data=hash);
    g_hash_table_foreach(hash, printone, NULL);
}
```

❶ 记住filestruct包含了一个void指针——data。因此这个函数的第1行当然将声明一个正确类型的变量，作为输入的void指针的转换目标。

❷ UTF-8字符是可变长度的，因此我们需要一个特殊的函数获取一个字符串的当前字符或者进入到下一个字符。

我是个笨手笨脚的人，几乎会犯每种可能出现的错误，但是我几乎不会把错误类型的结构放在列表、树等数据结构中。下面是我所总结的保证类型安全的规则。

- 如果我有一个基于void指针的称为active_groups的链表和另一个称为persons的链表，很显然当人们阅读像 g_list_append（active_groups, next_person）这样的代码时，会发现它把错误类型的结构与错误的列表进行了匹配，无须编译器报错。因此，我的第一个成功秘密是在进行乏味的工作时使用含义清晰的名称。
- 在代码中使图10-1的两端尽可能地靠近，这样当我们修改了其中之

一时，就可以很方便地修改另外一个。

- 我此前已经提到过这个规则，一个接受void指针为参数的函数在第1行就应该声明一个正确类型的变量，并有效地将void指针转换为正确的类型，就像printone和equal_chars一样。把它放在最前面可以有效地提高执行正确转换的可能性，一旦完成了转换，编译器就可以提供了类型安全的检查了。
- 为特定的泛型过程或结构建立对应的结构是极为合理的做法。

  ——如果没有特制的结构，当我们修改输入类型时，每次都必须记住把void指针转换位置，然后对它进行修改并将其转换为新类型，编译器对此并不会提供帮助。如果我们所发送的是一个保存数据的结构，那么唯一需要做的事情就是修改结构的定义。

  ——与此类似，当我们认识到需要向回调函数传递一段额外的信息时，就可以发现传递结构的好处了，我们唯一需要做的就是在这个结构的定义中添加一个元素。

  ——如果只是传递一个整数，似乎不值得传递一个新的结构，但实际上这是最具风险的情况。假设我们有一个通用过程，它接受一个回调函数和一个需要发送给这个回调函数的void指针，当我们像下面这样向它传递一个回调函数和一个指针时：

```
void callback (void voidin){
    double input = voidin;
    ...
}

int i=23;
```

```
generic_procedure(callback, &i);
```

有没有觉得这种无伤大雅的代码存在类型灾难？不管一个int值的位模式是怎么样的，当它被回调函数作为double值读取时，它的结果肯定不会是23。声明一个新结构似乎有点小题大做，但它可以防止一个很容易发生并且显得非常自然的错误：

```
typedef struct {
    int level;
} one_lonely_integer;
```

——我发现，如果在有些代码片段中只要定义一种类型就可以适合所有的处理时，可以减轻我们的认知负担。当我进行类型转换并发现它显然就是适合当前情况的类型时，就可以确认自己是正确的。以后我也不必怀疑应该再次检查char *而不是char **、wchar_t *或其他才是正确的类型。

本章讨论了许多向函数发送结构以及返回结构的方法，这些方法可能非常简单：用一个良好的宏，输入结构可以用默认值进行填充并提供命名函数输入；输出结构可以使用复合常量随时创建；如果函数必须复制结构（就像在递归中），只要一个等号就行了；返回一个空白结构更是小儿科，只要使用什么也不设置的指定初始化器就可以了。把一个专门创建的结构与一个函数相关联就解决了与使用泛型过程或容器相关联的许多问题，遇到泛型就是使用这些技巧的好时机。结构甚至可以提供

空间让你保存错误代码，这样就避免了把它们硬塞到函数的参数中。只要编写一个简单的类型定义，就可以收到丰厚的回报。

---

[1] 参考CERT网站。

[2] 如果你担心用户的编译器不支持__attribue__，autotools 可以帮助你。将AX_C__ATTRIBUTE__宏粘贴到你项目目录中的aclocal.m4文件中，并且将AX_C_ATTRIBUTE_加入到configure.ac文件中，如果发现用户的编译器不支持__attribute__，那么用预编译器起将__attribute__定义成空。

第11章 C语言面向对象编程

无论是在C语言或在其他语言中，典型的库的一般格式是：

- 一小组数据结构，代表库所针对领域的关键概念。
- 一组函数（经常被称为接口函数），用于处理那些数据结构。

比如说，一个XML库，会有一个代表XML文档的数据结构，以及类似文档的各类视图，加上一些关联数据结构和XML磁盘文件、查询结构的成员等功能函数。一个数据库管理库，一般有代表表单的数据结构，外加一些与数据库对话和剖析它发送的数据的函数。

这是一种组织程序或者库的特别体贴的方式。通过这种方式，一个作者可以根据他手头的问题挑选出适当的名词和动作。

面向对象编程（OOP）中的第一个乐趣就是定义术语。尽管我不想浪费时间（以及招致猛烈的争论）来给你一个精确的定义，前面关于一个面向对象的库的描述应该给了你一点基本的概念了：一些核心数据结构，每个都伴随一组函数用以处理那些核心数据结构。

有些专家认为某一个特性是OOP核心概念，你总会发现不同的专家对此有不同看法。不管怎样，除了基本的结构加函数这个定义以外，OOP还有以下的一些扩展。

- 继承，一个结构被扩展，以便加入新的元素。

- 虚函数，对一个类所有的对象都有默认的行为，但是对于不同对象的实例有定制的行为（在继承树的后代中）。
- 范围的精确控制，把结构的成员分为私有的和公共的。
- 运算符重载，同样的运算符在处理不同的类型的时候，含义是不同的。
- 引用计数，允许对象只在它不被使用的时候才被释放掉。

本章的一部分会考虑如何在C语言中实现这些特性。这些都不太难：引用计数只需要维护一个计数器，函数（不是操作符）重载使用_Generic关键字，这也是这个关键字设计的缘由。虚函数可以通过分发函数来实现，分发函数有选择地备份一个key/value候选函数的表格。

这带给我们另外一个问题，如果这些基于结构+函数的扩展能如此简单地实现，也就是需要几行代码，为什么我们不一直用它呢？

连接语言和认知的Sapir-Whorf假说有不同的描述；条款1认为一些语言强迫我们去想一些其他语言不强迫我们去考虑的事情。很多语言强迫我们去考虑性别，当你对于一个人来造句，但是不考虑他的性别标签如he、she、his和her时，你会发现有点不适应和不舒服。C要求你比其他语言更加考虑内存的分配（有些非C语言的人看待C代码就是一些对内存进行改变的语句）。而那些实现了访问控制的语言强迫你考虑何时何地一个变量是可见的。即使语言在技术上允许你将你的对象成员全部设置为公开变量，但是如果你真的这么做，有些人会认为你疯了，并且提醒你语言的规则要求你必须考虑精细的范围控制。

使用C语言让我们处在一个有利的位置，如果我们使用标准的OOP语言，例如C++和Java：我们就不会这么有利了，我们可以实现一些结

构+函数的扩展，但是我们从不强迫这么做，当它们并不会真正给我们带来一些益处的时候，我们可以不去管它。

11.1 扩展结构和字典

本节的前面，我给出了一个如何组织库结构的例子：一个结构外加上作用于这个结构的一组函数。但是，本节的目的是如何实现扩展：如何将一个新的成员加入到结构中，如何加入一个函数，以便能让这个函数同时在旧的结构和新的结构上都能正常工作？

1936年，为了解决一个数学问题（Entscheidungsproblem），Alonso Church发明了一个lambda代数，一种正规的描述函数和变量的方法。在1937年，为了解决同样的问题，Alan Turing透过特定的计算机模型描述了正规语言，计算机通过磁带保存数据，并通过可以移动位置的读写头在磁带上读写数据。随后，Church的lambda代数和Turing的机器被证明是对等的——任何你用其中一个表达的计算，你也可以用另一个模型来表示。从那时起沿用至今，Church和Turing的结构一直都是我们构造数据的理论基础。

Lambda代数非常依赖命名列表；在类lambda的伪代码中，我们可以把一个人的信息这样表达：

```
(person (
    (name "Sinead")
    (age 28)
    (height 173)
))
```

在Turing机模型中，我们会为这个结构分配一块磁带。头几块应该

是一个名字（name），随后的几块保存年龄（age），如此类推。几乎是一个世纪过去，Turing的磁带模型仍然是对计算机不错的描述：参见6.3.4“你需要知道的各种指针运算”，其中这种基地址加偏移量的形式正是C处理数据结构的方式。我们可以这样写：

```
typedef struct {
    char * name;
    double age, height;
} person;

person sinead = {.name="Sinead", .age=28, .height=173};
```

sinead将指向一块内存，而sinead.height将指向紧挨着name和age之后的磁带位置（当然是在各种填充和对齐之后）。

这里对比列表方法和内存块方法之间的不同：

- 从处理速度的角度看，这种基地址加偏移量的方法是最快的。告诉计算机从一个确定地址按照一定偏移量跳转到别的位置仍然是可以执行的最快的操作了。你的C编译器甚至在编译时就把标识翻译成了偏移量。相反地，在列表中寻找某个元素需要搜索过程：给定“age”标识，列表中的哪个成员与之对应？且它的数据又在内存的什么位置？每个系统都有加速搜索的技术，但是一个搜索总是比一个简单的基地址加偏移量的工作量多。
- 向列表中添加一个新的成员比在结构中添加一个新成员简单得多，结构基本上在编译时就确定了。
- C 编译器可以在编译的时候告诉你hieght是一个笔误，因为它可以检视数据结构的定义并发现那里没有这个成员。因为一个列表是可扩展的，在运行这个程序并检查这个列表之前， 我们并不会知道

没有hieght成员。

最后两条差异展示了一些压力：我们想要可扩展性，以便我们可以向结构添加一个成员；我们需要注册机制，以便不属于结构的事物能够被当作错误提示出来。这里必须坚持一种平衡，而在为一个现存的列表实现受控的可扩展性的做法，每个人都不一样。

C++、Java和它们的兄弟都能够从现有的类型扩展出新的类型，新的类型中包含原有类型的成员。你仍然有“基地址加偏移量”方式的速度，以及编译时检查，但是你需要手工写好多代码；然而C有struct和它的简单得有点荒谬的作用域规则（参见11.3“作用域”），Java有implements、extends、final、instanceof、class、this、interface、private、public和protected等。

Perl、Python以及很多类LISP语言是基于命名列表的，所以是一种实现结构的自然方式。扩展列表只需要向其添加成员。这样做的好处是：通过添加一个新的命名条目来实现完整的扩展性；坏处是：像之前一样，我们没有得到注册功能，虽然你可以通过各种花招来改进命名检索，但还是与一个简单的基地址加偏移量的速度差距很大。这一家族的很多语言有一个类定义系统，所以你可以注册一些列表条目并检查未来的使用是否符合这个定义，也就是说，如果做得恰到好处，可以提供一个非常好的在可检查性和扩展性之间的妥协。

回到简朴、古老的C语言，它的struct是最快的利用结构成员的途径，我们牺牲了运行时的可扩展性，但是我们获取了编译时检查。如果你想要一个可以在运行时随需扩展的灵活的列表，你将需要一个列表结构，比如GLib的hash表，或者下面会介绍的字典。

11.1.1 实现一个字典

因为我们有了基于结构的C，所以字典是一个容易构建的结构。这个过程是一个非常好的构建一些对象的机会。然而请注意，其他的作者已经完成了这个功能，并把它做得无懈可击；例如，可参见GLib的键/值数据表或者GHashTable。这里的重点就是用复合结构加上简单的数组来构建一个简易的字典对象。

我们马上开始一个简单的键/值对的例子。它的机制将实现在keyval.c中。例11-1中的头文件列出了所有的结构和它的接口函数。

例11-1　头文件，或者说是键/值类的公共部分（keyval.h）

```
typedef struct keyval{
   char *key;
   void *value;
} keyval;

keyval *keyval_new(char *key, void *value);
keyval *keyval_copy(keyval const *in);
void keyval_free(keyval *in);
int keyval_matches(keyval const *in, char const *key);
```

对传统的面向对象编程语言有经验的读者会发现这段代码非常熟悉。通常建立一个新对象的第一个实例的第一反应就是写下new/copy/free函数，这个例子正是这样做的。在这之后，典型的是几个结构相关的函数，比如用来检测在keyval当中的键是否与输入参数key符合的keyval_matches函数。

有了new/copy/free函数就意味着你不用太担心内存管理了：在new和copy函数中，用malloc来分配内存；在free函数中，用free来释放结

构，而且在配置好这些函数后，使用这个对象的代码将不再用malloc和free，而是信任keyval_new、keyval_copy和keyval_free等函数来正确管理内存。

例11-2　典型的键/值对象模板：一个带有new/copy/free的结构（keyval.c）

```
#include <stdlib.h> //malloc
#include <strings.h> //strcasecmp (from POSIX)
#include "keyval.h"

keyval *keyval_new(char *key, void *value){
    keyval *out = malloc(sizeof(keyval));
    *out = (keyval){.key = key, .value=value};     ❶
    return out;
}

/** Copy a key/value pair. The new pair has pointers to
  the values in the old pair, not copies of their data. */
keyval *keyval_copy(keyval const *in){
    keyval *out = malloc(sizeof(keyval));
    *out = *in;                                    ❷
    return out;
}

void keyval_free(keyval *in){ free(in); }

int keyval_matches(keyval const *in, char const *key){
    return !strcasecmp(in->key, key);
}
```

❶ 指定初始化值以便填充一个结构。

❷ 记住，你可以用一个等号来复制结构的内容。如果我们想要复制结构体内的指针的内容（不是指针自己），需要在这之后添加更多的代码。

现在我们有了一个代表单个的键/值对的对象，可以继续用这些对象的列表来建立一个字典。例11-3提供了头文件。

例11-3　字典结构的公共部分（dict.h）

```
#include "keyval.h"

extern void *dictionary_not_found;    ❶

typedef struct dictionary{
   keyval **pairs;
   int length;
} dictionary;

dictionary *dictionary_new (void);
dictionary *dictionary_copy(dictionary *in);
void dictionary_free(dictionary *in);
void dictionary_add(dictionary *in, char *key, void *value);
void *dictionary_find(dictionary const *in, char const *key);
```

❶ 这是在字典里找不到一个键的时候的一个标记。它必须是公共的。

就像你所看到的，你有了同样的new/copy/free函数，加上一些其他的字典特有的函数，以及随后说明的标记。例11-4提供了私有的实现部分。

例11-4　字典对象的实现（dict.c）

```
#include <stdio.h>
#include <stdlib.h>
#include "dict.h"

void *dictionary_not_found;

dictionary *dictionary_new (void){
    static int dnf;                                          ❶
    if (!dictionary_not_found) dictionary_not_found = &dnf;
```

```
    dictionary *out= malloc(sizeof(dictionary));
    *out= (dictionary){ };
    return out;
}

static void dictionary_add_keyval(dictionary *in, keyval *kv){    ❷
    in->length++;
    in->pairs = realloc(in->pairs, sizeof(keyval*)*in->length);
    in->pairs[in->length-1] = kv;
}

void dictionary_add(dictionary *in, char *key, void *value){
    if (!key){fprintf(stderr, "NULL is not a valid key.\n"); abort();} ❸
    dictionary_add_keyval(in, keyval_new(key, value));
}

void *dictionary_find(dictionary const *in, char const *key){
    for (int i=0; i< in->length; i++)
        if (keyval_matches(in->pairs[i], key))
            return in->pairs[i]->value;
    return dictionary_not_found;
}

dictionary *dictionary_copy(dictionary *in){
    dictionary *out = dictionary_new();
    for (int i=0; i< in->length; i++)
        dictionary_add_keyval(out, keyval_copy(in->pairs[i]));
    return out;
}

void dictionary_free(dictionary *in){
    for (int i=0; i< in->length; i++)
        keyval_free(in->pairs[i]);
    free(in);
}
```

❶ 在键/值表中有一个NULL是合理的，所以我们需要一个独特的标记来表示一个缺少的值。我不知道dnf被保存在内存的什么地方，但是它的地址一定是唯一的。

❷ 记住，被标记为static的函数不能在文件之外使用，所以这是一个关于这个函数是私有实现的附加的提醒。

❸ 我必须承认：像这样使用abort是一种不好的形式。更好的办法是使用stopif.h中的宏（见例10-2）。我这么做是为了展示一个测试框架的特性。

现在我们有了一个字典，例11-5可以使用它而不用考虑内存管理，因为new/copy /free/add函数会负责内存管理，我们也不需要引用键/值对，因为对字典的用户来说， 键/值对的抽象层级太低了。

例11-5　字典对象的用法示例；不用深入了解结构的内部，因为接口函数提供了我们所需要的一切（dict_use.c）

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

int main(){
    int zero = 0;
    float one = 1.0;
    char two[] = "two";
    dictionary *d = dictionary_new();
    dictionary_add(d, "an int", &zero);
    dictionary_add(d, "a float", &one);
    dictionary_add(d, "a string", &two);
    printf("The integer I recorded was: %i\n", *(int*)dictionary_find(d, "
an int"));
    printf("The string was: %s\n", (char*)dictionary_find(d, "a string"));
    dictionary_free(d);
}
```

那么写一个结构和它的new/copy/free与其他的附加的函数可以给我们封装问题的合理层次：字典不需要去考虑键/值对的内部机理，而应用程序不需要担心字典的内部细节。

这个样本程序代码并不比其他代码差，但是这里当然有一些对new/copy/free函数的重复。而且随着下面的例子，你还会继续看到这个

样板程序代码。

有时，我甚至给自己写一个自动产生这些的宏。例如，copy函数只是在处理内部指针的时候有所不同，所以我们写一个宏来自动化所有无关内部指针的模板：

```
#define def_object_copy(tname, ...)          \
    void * tname##_copy(tname *in) {         \
        tname *out = malloc(sizeof(tname)); \
        *out = *in;                          \
        __VA_ARGS__;                         \
        return out;                          \
    }

def_object_copy(keyval) // Expands to the previous declarations of keyval_
copy.
```

但是这种冗余性是不值得担心的。有时多写一点代码真的可以使程序更加可读和健壮，尽管可能有违数学美学对最小化重复文字的要求。

11.1.2 C，更少的缝隙

在C语言中，如果你想扩展一个结构，那么你需要另外一个结构来包装这个结构。假设我们有一个如下定义的类型：

```
typedef struct {
    ...
} list_element_s;
```

这个结构已经定义好了而且不能被改变，但是我们希望添加一个类型标志。于是我们将需要一个新的结构：

```
typedef struct {
    list_element_s elmt;
    char typemarker;
```

```
} list_element_w_type_s;
```

好处就是：这也简单得有点愚蠢了，而且你仍然具有速度上的好处。坏处就是：现在，每当你要引用成员的名字，你将需要写出完整的路径，your_typed_list->elmt->name，类似C++/Java的扩展中你可以直接写成：your_typed_list->name。要是再多加一些层次，就开始变得有点烦人了。不过你已经在6.3“不使用malloc的指针”中看到一个别名如何帮助解决这个问题了。

C11允许在结构中包含一个匿名成员，这样使得在一个结构内的结构更加容易使用。虽然这个是在2011年12月才加入标准的，但Microsoft很早就有这个扩展了。我将分别展示强模式和弱模式两种模式；gcc和clang允许通过-fms-extensions的命令行选项来使用强模式。如果你没有使用上面的命令行选项，那么你的支持C11标准的编译器将使用弱模式。

强模式的语法是：在新结构声明中的某处包含另一个结构，参见例11-6中的point，不用命名这个成员。例11-6仅仅用一个结构的名字——struct point，而不是类似struct point element name的命名方式。这样所有的在新结构中被包含的结构的成员就像是在新结构中声明的一样。

例11-6把一个2D点扩展为3D点。目前唯一值得注意的是，因为threepoint结构对point进行了如此无缝的扩展，以至于threepoint的用户甚至不会知道它的定义是基于另一个结构之上的。

例11-6　一个在包装结构中的匿名子结构，无缝地融入包装结构（seamlessone.c）

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

typedef struct point {
   double x, y;
} point;

typedef struct {
    struct point;                                   ❶
    double z;
} threepoint;

double threelength (threepoint p){
    return sqrt(p.x*p.x + p.y*p.y + p.z*p.z); ❷
}

int main(){
    threepoint p = {.x=3, .y=0, .z=4};          ❸
    printf("p is %g units from the origin\n", threelength(p));
}
```

❶ 这里是匿名的。而非匿名的版本还是有类似struct point twopt的名字。

❷ 在point中的x和y行为就和threepoint中添加的z一样。

❸ 甚至声明本身都没有给出x和y是从一个现存的结构中继承的。

最初的对象point，可能伴随了几个也可能有用的接口函数，比如一个测量原点和给定点之间的距离的length函数。既然在较大结构中没有指定子结构的名称，我们要怎么用这个函数？

解决方法是用一个匿名的联合，包含一个命名的point和一个未命名的point。作为两个相同的结构的联合，这两个结构共享所有的成员，且唯一的区别就在于是否命名：当你需要调用使用原始的结构作为一个输入的函数的时候使用命名版本，而在无缝地融合在更大的结构中时使用

匿名版本。例11-7用这个技巧重写了例11-6。

例11-7　point结构被无缝地融入一个thresspoint，且我们仍然有一个函数的名字，用以使用操作一个point的结构的函数（seamlesstwo.c）

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

typedef struct point {
   double x, y;
} point;

typedef struct {
    union {
        struct point;              ❶
        point p2;                  ❷
    };
    double z;
} threepoint;

double length (point p){
    return sqrt(p.x*p.x + p.y*p.y);
}

double threelength (threepoint p){
    return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
}

int main(){
    threepoint p = {.x=3, .y=0, .z=4};❸
    printf("p is %g units from the origin\n", threelength(p));
    double xylength = length(p.p2);   ❹
    printf("Its projection onto the XY plane "
           "is %g units from the origin\n", xylength);
}
```

❶ 这是一个匿名结构。

❷ 这是一个命名结构。作为一个联合的部分，它与匿名结构是相同的，唯一的差别是是否有名字。

❸ point结构仍然被无缝地包含在threepoint结构中，但是……

❹ ……p2是一个命名成员，就像它通常表现的那样，所以我们可以用它来调用按照原来的结构所写的接口函数。

在声明了threepoint p之后，我们可以通过p.x来引用*x*坐标（以匿名结构），也可以用p.p2.x（以命名结构）。这个例子的最后一行显示了投射到*xy*平面的长度，并且是用length（p.p2）实现的。从此，我们成功地扩展了结构并且仍然可以使用与原有的结构关联的所有函数。

从多个结构中继承也许能工作，也许不能工作。如果两个结构都包含一个成员*x*，那么编译器也许会抛出一个错误，我们没有办法来重新改变一个存在的结构中的乘员的名字，或者只是抽取出其中的一个子集。但是如果在你的不能修改的旧的代码中有一个结构，其中包含了10个成员，而你只是想再在其中增加一个新的成员，那么上面的方法就很适合你了。

警告

你有没有注意到这是我在本书中第一次使用union关键字？联合是另一种解释起来破费口舌的东西——它像一个结构，但是所有的成员占用同一个空间——然后关于如何不把你自己吊死的警告就要花费好几页。内存是便宜的，而且对于编写应用，我们不需要去在意内存的对齐，所以坚持使用结构将减少出错的可能，即便是每次只使用一个成员。

如果不使用-fms-extensions标志，那么就是弱模式了。它不允许我们使用匿名的结构标识符来引用我们以前定义过的结构，相反，它要求结构必须在本地定义。这样，我们需要复制和粘贴整个P2 struct的定义了。

```
typedef struct {
    union {
        struct {
           double x, y;
        };
        point p2;
    };
    double z;
} threepoint;
```

在示例代码库中，你会发现seamlessthree.c文件，它用这种方式来编译，结果与seamlesstwo.c是一样的。

弱模式也许对我们这里讨论的扩展并不是非常有用，因为现在你必须保持两份独立的结构定义同步。但是在一定的场合下，它还是有一定的用处的：

- 这本书的大部分内容都是讨论如何处理大量的陈旧的C代码。只要能修改那些旧代码的人在2003年退休了，就再也没人敢去动那些旧代码了，这个时候，你把旧代码中的结构复制处理就是比较安全的了。
- 如果所有的代码都在你的控制之下，那么你有一个选项，可以通过宏来去掉这份重复。

```
#define pointcontents { \
   double x, y;        \
}
```

```
typedef struct pointcontents point;

typedef struct {
    union {
        struct pointcontents;
        point p2;
    };
    double z;
} threepoint;
```

这不仅仅是方便，也保证了原始代码和扩展代码的一致性，减少对新标准的依赖，同时让编译器去帮助你确保类型安全。

基于指向对象的指针编码

第10章讲述的大部分技巧都是关于数据结构的，而不是关于指向数据结构的指针的，但是本章所有的例子都是声明和使用结构的指针。

当你使用一个简单的结构的时候，/new/copy/free函数就会写成：

new

在你使用结构的第一行就用指定的初始化值去初始化。另外，还可以在编译时声明结构，所以它们可以立即被用户所用而不用先调用一个构造函数。

copy

直接使用等号就好了。

free

不需要；它会在作用域之外自动释放。

就是因为有了指针，我们把事情变得更加困难了。而且根据我们已经看到的，在用指向对象的指针作为我们设计基础的时候，是存在很多共识的。

使用指针的好处如下所示。

- 复制一个单独的指针比复制一个完整的结构代价要小得多，所以每当你调用以结构作为输入的函数的时候都会节省几毫秒。当然，这只会在调用几十亿次函数调用后才能

感觉到。

- 关于数据结构的库（比如说，你的树和链表）都是围绕一个指针的方式来编写的。
- 现在你正在填充一个数或者列表，由系统在它创建的作用域结束后自动释放结构可能并不是你想要的。
- 许多接受结构输入的函数会在函数内部修改结构的内容，也就是必须传入指针才能够完成预期的目的。部分函数使用结构参数，部分使用结构指针作为参数，这样混合使用会造成困扰（笔者曾经这样写过函数库的界面，现在感到十分后悔），最好每次都传送指针。
- 如果结构的内容包括一个指向某处数据的指针，那么使用一个普通结构的好处就消失了：如果你想要一个深度复制（其中被指向的数据被复制，而不仅仅是指针），那么你需要一个copy函数，而且你也可能想要一个free函数来确保内部数据被释放了。

在使用结构方面，并不存在一套方案适合所有情况。随着你的项目变得更大，而且一个随用随弃的结构成为你组织数据的核心，这会彰显使用指针的好处，使用非指针的好处则逐渐衰减。

11.2 你结构中的函数

目前，每一个头文件都是一个结构，后面跟着一组函数。其实结构之中可以包含一个函数，就像包含其他的数据成员一样简单。下面我们把除了object_new函数以外所有的函数都包含到一个结构中。

```
typedef struct keyval{
    char *key;
    void *value;
    keyval *(*keyval_copy)(keyval const *in);
    void (*keyval_free)(keyval *in);
    int (*keyval_matches)(keyval const *in, char const *key);
} keyval;

keyval *keyval_new(char *key, void *value);
```

提示

假设我们有一个指向函数的指针，fn，意味着*fn是一个函数而且fn是它在内存中的指针。那么（*fn）（x）显然就是一个函数调用，但是fn（x）是什么意思呢？在这个例子中，C 把调用这个指向函数的指针对应成对那个函数的调用。这种情况称为指针退化。这是为什么我在本文中把函数和指向函数的指针视作等同的原因。

这可以说是一种风格的选择，只是影响了我们如何在文档中查找函数，以及函数在纸面上的外观而已。对文档来说，比起copy_keyval风格，我更喜欢keyval_copy的命名方式：利用这种风格，文档可以在一个地方列举出全部的与keyval_相关的函数。

这种方式的真正好处在于你可以很容易地修改和对象每个实例都关联的函数。例11-8展现了一个简单的列表结构，它并没有显式地说明它能够保存一个广告、歌词、菜谱或者其他的什么东西。如果对于不同的东西能按照不同的格式来打印就好了。所以我们有几种类型不同的打印函数。

例11-8　一个通用的结构，包含内建的打印方法（print_typedef.h）

```
#ifndef textlist_s_h
#define textlist_s_h

typedef struct textlist_s {
    char *title;
    char **items;
```

```
    int len;
    void (*print)(struct textlist_s*);
} textlist_s;

#endif
```

例11-9声明并使用了两个上面定义的对象，那些独立的打印方法作为对象的一部分来定义。

例11-9　把函数放到结构的里面，可以清楚地表明哪个函数应该搭配哪个结构

```
#include <stdio.h>
#include "print_typedef.h"
static void print_ad(textlist_s *in){
    printf("BUY THIS %s!!!! Features:\n", in->title);
    for (int i=0; i< in->len; i++)
        printf("· %s\n", in->items[i]);
}

static void print_song(textlist_s *in){
    printf("♬ %s ♬\nLyrics:\n\n", in->title);
    for (int i=0; i< in->len; i++)
        printf("\t%s\n", in->items[i]);
}

textlist_s save = {.title="God Save the Queen",
     .len=3, .items=(char*[]){
    "There's no future", "No future", "No future for me."},
    .print=print_song};                                    ❶

textlist_s spend = {.title="Never mind the Bollocks LP",
    .items=(char*[]){"By the Sex Pistols", "Anti-consumption themes"},
    .len=2, .print=print_ad};

#ifndef skip_main
int main(){
    save.print(&save);                                     ❷
    printf("\n-----\n\n");
    spend.print(&spend);
}
#endif
```

❶ 不要错过了，就是在这里把函数加入了save结构。同样的事情发生在下面几行后，把print_ad函数加入了spend结构。

❷ 你调用内嵌在结构中的函数，它们看起来都是相同的。你不需要记住save是一个歌词，spend是一个广告。

最后三行，我们对于完全不同的函数，使用了相同的界面。对于使用textlist_s* t的函数，你可以这样调用：t->print（&t）。

负面来说，我们再一次违反了规则，那就是做不同的事的东西应该看起来不同：如果在打印函数中存在一些细微的副作用的话，你是不会收到任何警告的。

注意static的使用，这代表着文件外面的代码不能通过名字来直接调用print_song和print_ad。但是它们可以通过save.print和spend.print来调用。

这里我们再多说几句。首先，save.print（&save）用重复的形式使用了save。系统如果知道你调用函数的那个对象实例就是函数的第一个参数，那么我们就可以写成 save.print()的方式了。函数可以使用一个this或者self的变量名，或者我们可以一个特殊的规则，把object.fn（x）重新定义为fn（object,x）的方式。

但是对不起，C语言中不存在这些。

C并没有给你定义魔法变量，它们对于送入函数的参数总是非常诚实并且透明的。正常来说，如果我们想在函数参数上做点文章，我们可以使用预编译器，它会很高兴地把f（anything）重写成f（anything

else）。但是，所有的转换只发生在括号的内部，所以对于预编译器来说，我们不能把s.prob（d）改写成s.prob（s,d）。如果你不想模仿C++语法的话，你可以这样定义宏：

```
#define Print(in) (in).print(&in)
#define Copy(in, ...) (in).copy((in), __VA_ARGS__)
#define Free(in, ...) (in).free((in), __VA_ARGS__)
```

但是现在，你的全局命名空间中充斥着Print、Copy和Free符号，也许是值得的（特别是每个函数都有对应的copy和free函数）。

你可以保持命名空间的有序性，并且通过对宏正确命名来避免命名冲突：

```
#define Typelist_print(in) (in).estimate(&in)
#define Typelist_copy(in, ...) (in).copy((in), __VA_ARGS__)
```

回到typelist_s，我们有了打印歌词和广告的方法，但是对于菜谱应该怎么办呢？或者人们又新加了喜欢的内容呢？如果人们定义了一个列表，但是没有提供对应的打印函数，这个时候会发生什么呢？

我们需要一个默认的函数，一个最简单的实现方式是使用分发函数。函数会检查输入结构中的print函数，如果它发现这个函数不是NULL，那么就使用这个函数。否则提供一个默认的打印方法。例11-10演示了分发函数，它可以正确地打印一个带有print函数的song结构。由于recipe没有对应的print函数，分发函数用默认函数来代替。

例11-10　菜单没有print方法，但是分发函数也打印了它（print_dispatch.c）

```
#define skip_main
#include "print_methods.c"
textlist_s recipe = {.title="1 egg for baking",
     .len=2, .items=(char*[]){"1 Tbsp ground flax seeds", "3 Tbsp water"}}
;

void textlist_print(textlist_s *in){
    if (in->print){
        in->print(in);
        return;
    }

    printf("Title: %s\n\nItems:\n", in->title);
    for (int i=0; i< in->len; i++)
        printf("\t%s\n", in->items[i]);
}

int main(){
    textlist_print(&save);
    printf("\n-----\n\n");
    textlist_print(&recipe);
}
```

这样分发函数就给了我们缺省函数，解决了烦人的缺少神奇的this或self变量的问题，并且以一种看起来和通常的textlist_copy或testlist_free这类接口函数相似的方式。

还有别的方法来实现。前面，我用指定的初始化函数来配置函数，所以没有被指定的成员为NULL，同时转发函数就有存在的必要。如果我们要求用户总是用一个testlist_new函数，那么我们可以把缺省函数设定在那里。就像前面那样，可以用一个简单的宏来消除save.print（&save）的冗余性。

你又一次面临选择。我们已经有了超出必要的语法工具来标准化地为派生的对象调用派生的函数。这样你只需要编写那些派生函数，而只要以一种标准的方式调用它们，它们的行为就是可预期的。

虚函数表

textlist_s结构被设计出来以后，过了一段时间，我们发现又有了新的需求。现在想把列表放到WWW上，但是需要用html来格式化这些内容。如何在结构中加入一个HTML的输出函数呢？目前这个结构中只有输出到屏幕的函数啊！

在11.1.2“C 更少的缝隙”一节，你已经看到了结构可以被扩展，我们可以使用这种方法来将新函数加入存在的结构中。

本节展示了一个替代的方法，它把新函数加入到对象的结构之外。它们是虚函数表中的一个记录。虚函数来源于面向对象编程词典。在20世纪90年代的风格中，任何在软件中的实现都叫作虚。一个虚函数表是一个hash表，简单的键/值列表。11.1.1“实现一个词典”已经演示了如何建造这样一个键/值列表，本节我将使用Glib的实现。

给定一个对象，产生一个hash（键），并且把一个函数与这个hash关联。然后当用户调用分发函数来执行某种操作的时候，分发函数首先检查hash表来获取这个函数，如果找到了就执行它，否则就执行默认函数。

这里我们需要一些组件来使得整个系统工作：

- 一个hash函数。
- 一个类型检查器。我们必须确保保存在hash表中的函数具有同样的类型标识。
- 一个键/值对，以及配套的存储和获取函数。

hash函数

一个hash函数把输入转换成一个整数值，它的目标就是避免把两个不同的输入转换成相同的整数值。

Glib提供了一些hash函数，包括g_direct_has、g_int_hash和g_str_hash。g_direct_hash可以用于指针，它只是把指针读成数字，这样如果两个指针没有指向相同的内存区域，那么它们就不会发生冲突（collision）。

对于复杂的情况，我们可以发明新的hash。这里是一个通用的hash函数，由Dan J. Bernstein发明。对于字符串中的每一个字符（或者UTF-8多字节字符中的每一个字节），它乘以33，然后加上新的字符（或字节）。这个值比较容易在unsighned int所能容纳的范围内溢出，但是溢出是这个算法的一个正常的步骤。

```
static unsigned int string_hash(char const *str){
    unsigned int hash = 5381;
    char c;
    while ((c = *str++)) hash = hash*33 + c;
    return hash;
}
```

另外，Glib也提供了自己的g_str_hash函数，所以没必要使用上面的函数，但是我们可以把它当成一个模板用以实现其他的hash函数。如果我们有一个指针的列表，那么我们可以使用这个hash：

```
static unsigned int ptr_list_hash(void const **in){
    unsigned int hash = 5381;
    void *c;
    while ((c = *in++)) hash = hash*33 + (uintptr_t)c;
    return hash;
}
```

对于面向对象的读者，我们已经几乎完成了多分发的功能了。给定两个不同的对象，我可以将它们的指针hash后，再将正确的函数保存到对应的键/值对中。

Glib中的hash表也需要相等检查，所以Glib对不同的hash表提供了对应的g_direct_equal、g_int_equal和g_str_equal函数。

对于任何hash函数，还是有一定概率会发生碰撞的，虽然对于一个精心设计的hash来说这个概率比较小。我使用上面提到过的hash函数，我知道总有一天有些人会比较不幸地遇到碰撞的事情发生。如果决定如何分配时间的话，我总有一些bug需要修改，一个特性需要实现，一个文档需要增加内容，一个用户需要交谈等。这些带来的好处都比处理hash碰撞带来的好处更多。Git也依赖于hash来保存提交记录，而且用户现在已经产生了数百万次的提交，不过现在我看Git的维护者也没有把处理碰撞这个任务放到它们的时间表中。

类型检查

我们允许用户在hash表中保存任何函数，然后分发函数会通过预先定义的模板来获得这个函数。如果一个用户写了一个接受错误类型的函数，你的分发函数会失败，然后用户会在各种社交媒体网站上批评你的代码不工作。

通常，函数调用都是显式地写到代码中的，所有的类型都在编译的时候检查。一方面，当动态地选择函数的时候，我们就丢失了这种类型安全性了；另一方面，我们可以利用它检查一个函数是否有正确的类型。

例如，我想让我的函数接受一个double*和int（例如一个列表和它的长度），并且返回一个out_type结构。我们可以这样定义类型。

```
typedef out_type (*object_fn_type)(double *, int);
```

然后定义一个什么也不做的函数：

```
void object_fn_type_check(object_fn_type in){ };
```

在下面的例子中，这会包装到一个宏，确保用户会调用它。这个函数可以帮我找回类型安全性：如果用户把一个错误类型的函数放到hash表中，那么函数在编译那个什么也不做的函数的时候，会抛出一个错误。

集成

例11-11是一个Vtable需要的头文件，提供了增加新方法的宏以及完成查询获取功能的分发函数。

例11-11　一个头文件，声明了与特定对象函数关联的Vtable

```
#include <glib.h>
#include "print_typedef.h"

extern GHashTable *print_fns;

typedef void (*print_fn_type)(textlist_s*);                          ❶

void check_print_fn(print_fn_type pf);

#define print_hash_add(object, print_fn){                        \   ❷
    check_print_fn(print_fn);                                    \
    g_hash_table_insert(print_fns, (object)->print, print_fn); \
}
void textlist_print_html(textlist_s *in);
```

❶ 这是可选的。但是一个好的typedef会让你使用函数指针的时候变得简单一些。

❷ 挑剔的用户认为类型检查函数是浪费时间，那么给他们提供一个做这个工作的宏。

例11-12提供了一个分发函数，它首先检查Vtable。它在Vtable中查找而不是在输入的结构里面查找，除了这点区别，它和前面的分发函数没有太大的区别。

例11-12　一个使用虚函数表的分发函数

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

GHashTable *print_fns;                                                    ❶

void check_print_fn(print_fn_type pf) { }
void textlist_print_html(textlist_s *in){
    if(!print_fns) print_fns=g_hash_table_new(g_direct_hash,g_direct_equal
); ❷

    print_fn_type ph = g_hash_table_lookup(print_fns, in->print);  ❸
    if (ph) {
        ph(in);
        return;
    }
    printf("<title>%s</title>\n<ul>", in->title);
    for (int i=0; i < in->len; i++)
        printf("<li>%s</li>\n", in->items[i]);
    printf("</ul>\n");
}
```

❶ 注意hash表在这里是一个私有的实现，并不是一个公开的结构，用户从来不直接使用它。

❷ 利用hash函数和相等函数来初始化Glib的hash表。一旦它们被保存在hash结构里面了，用户就不再需要显式地引用它们。这行建立了一个用户print函数的hash函数，我们可以根据需要建立很多hash函数。

❸ 输入结构的print方法可以用来识别这个结构是song：菜单，还是别的东西，所以我们可以获取正确的HTML print 方法。

最后，在例11-13中给出了最后的应用。注意用户只是使用宏来把一个特殊的函数和一个对象关联起来了，分发函数也做这个工作。

例11-13　一个关联了函数和对象的虚拟表(print_vtable_use.c)

```
#define skip_main
#include "print_methods.c"
#include "print_vtable.h"

static void song_print_html(textlist_s *in){
    printf("<title>♬ %s ♬</title>\n", in->title);
    for (int i=0; i < in->len; i++)
        printf("%s<br>\n", in->items[i]);
}

int main(){
    textlist_print_html(&save);              ❶
    printf("\n-----\n\n");

    print_hash_add(&save, song_print_html);  ❷
    textlist_print_html(&save);
}
```

❶ 目前，hash表是空的，所以这个调用会使用分发函数中的默认print方法。

❷ 这里，我们在hash表中加入了特殊的print函数，所以下一个调用中分发函数会找到并使用这个函数了。

Vtable是面向对象语言中很多特性得以实现的一个途径，而且也不难实现出来。如果你查查上面代码的数量，你会发现它还不到10行[1]。即使我们为了特定组合对象指定了一个特殊例子的函数，我们也不需要发明蹩脚的语法来完成这个任务。Vtable需要一些建造工作，但是通常可以只在后续的时候去完成，实际上，只对于特定结构的特定操作去实现它，才是更加有益处的。

11.3 作用域

所谓变量的作用域就是其可以存在和使用的代码范围。一个理智的编程方式的首选原则就是尽可能缩小变量的作用域，因为这么做就限制了你在某一点上需要记住的变量个数，这意味着这个变量被你无意写出的代码改变的风险更低了。

下面就是C语言中所有的关于变量作用域的准则。

- 变量在没有被声明前是不在其作用域内的。这一条太明显了，所以听起来有点蠢。
- 如果一个变量是在一对花括号内定义的，那么在结束花括号（右括号）之后，这个变量就在作用域之外了。特例：for循环和函数可以有变量定义在开始的花括号前的一对括弧内；在一对括弧内定义的这些变量，其作用域就等同于在花括号内定义。
- 如果一个变量不在任何花括号内，那么它的作用域就从它的声明开始到文件的结尾。

就这些了。

这里没有类作用域、原型作用域、友类作用域、命名空间作用域、动态作用域、扩展问题，或者特殊作用域的概念或操作符（当然C存在在那些花括号和有争议的static和extern连接指示符之外等问题）。字典的作用域有迷惑你了么？不要担心。如果你知道花括号在哪里，你可以决定在哪里用哪个变量。

其他的任何事情都是一个简单的推论了。例如，如果code.c有一行#include <header.h>，那么header.h的头文件被粘贴入code.c，而且变量从此随之有了作用域。

函数其实是另一个花括号作用域的例子。这里有一个示例函数，将所有的输入整数求和：

```
int sum (int max){
    int total=0;
    for (int i=0; i<= max; i++){
        total += i;
    }
    return total;
}
```

那么max和total就有了在这个函数之内的作用域，通过花括号准则和在花括号之前的括号中声明变量也相当于在花括号中声明的“半个”例外准则。同样的情况发生在for循环中，而且i的生成和死亡都是伴随着for循环的花括号。如果你有一个单行的for循环，你不需要写花括号，就像for（int i=0; i<=max; i++） total+=i;，i的作用域还是被限定在循环中。

总之，C非常棒，具有如此简单的作用域规则，只要找到结束花括号或者文件结尾。你只要用10分钟就可以把整个作用域系统教给一个无

知的学生。对于经验丰富的程序员，这个规则比函数和for循环的花括号还要普遍，所以你可以在特例情况中使用大括号将变量的作用域限制到特定范围内。就像在8.1“营造健壮和繁盛的宏”中的宏的技巧。

私有结构成员

这样我们一股脑把所有支持不断细化的作用域控制的附加规则和关键字都扔掉了。

我们能实现私有数据结构成员而不用额外的关键字吗？在典型的OOP用法中，“私有”数据并不是被编译器或者别的什么严肃的东西隐藏起来的：如果你有这个变量的地址（例如，如果你有它在数据结构中的偏移量），你可以直接指向它，在调试器中观察它，并修改它。为了给数据这种不透明性，我们也有这个技巧。

一个对象通常是通过两个文件定义的：带有细节的.c文件和用来被其他要使用这个对象的代码文件包含的.h文件。如果把.c文件想象为私有部分而把.h文件想象为公共部分，也不是不合理的。例如，假设我们试图把对象的一些成员设为私有。公共的头文件应该是：

```
typedef struct a_box_s {
    int public_size;
    void *private;
} a_box_s;
```

这个void指针基本上对其他的程序员是没有用的，因为他们不知道该把它转换成什么类型。私有的部分，a_box.c，将实现必要的类型定义：

```
typedef struct private_box_s {
```

```
    long double how_much_i_hate_my_boss;
    char **coworkers_i_have_a_crush_on;
    double fudge_factor;
} private_box_s;

//Given the typedef, we have no problem casting the private pointer to
//its desired type and making use here in a_box.c.

a_box_s *box_new(){
    a_box_s *out = malloc(sizeof(a_box_s));
    private_box_s *outp = malloc(sizeof(private_box_s));
    *out = (a_box_s){.public_size=0, .private=outp};
    return out;
}

void box_edit(a_box_s *in){
    private_box_s *pb = in->private;
    //now work with private variables, e.g.:
    pb->fudge_factor *= 2;
}
```

所以实现一个C结构的私有部分也不是那么困难，但是我很少看到有人在现实世界的库中这么用。很少有C程序员思考过这么做的真正好处。

下面是一个例子，展示了更常用的把私有成员放在一个公共结构中的方法。

```
typedef struct {
    int pub_a, pub_b;
    int private_a, private_b; //Private: please do not use these.
} public_s;
```

就是说，用文档表明什么是不可以用的，并信任你的用户不会作弊。如果你的同事不愿意听从这么简单的一个提示，那就把咖啡机锁在墙上准备加班吧，因为你已经陷入了一个编译器无法解决的问题了。

函数是特别容易被做成私有的：只要不把它们的声明放在头文件

中。或者，把static关键词放在定义的前面，这样读者就知道这个函数是私有的了。

11.4 用操作符重载进行重载

在我的印象中，多数人都会想起整数除法——比如3/2==1——多少有点恼人。如果我键入3/2，我的期望是1.5，可不成想是1。

确实，这是一个关于C和其他的整数算数语言的恼人的常识，并且更一般性地，这也展示给我们操作符重载的危险。就是指当一个操作符，比如“/”，根据不同的数据类型做一些不同的事情。对于两个整数类型，这个斜杠做除以并取整的操作，而对于其他的类型，它执行通常的除法。

回忆一下6.3“不使用malloc的指针”中的准则，行为不同的事物看起来也应该不同。这是3/2的失效：整数除法和浮点数除法的行为是不同的，但是看起来一模一样。随之肯定会引发迷惑和问题。

人类的语言是有冗余的，这其实是好事情，部分是因为它允许错误纠正。当Nina Simone说：“ne me quitte pas。”（如果逐字翻译过来就是“不要离开我，不”）如果你在开头空着也可以，因为“……me quitte pas”已经有pas来表示否定；如果你把结尾空着也可以，“ne me quitte……”已经有ne表示否定。

语法中的性别一般并没有太多现实的意义，而且有时随着选择词语的不同对象也不同。我喜欢的例子是西班牙语，其中el pene和la polla都是指同样的对象，第一个是阳性的而第二个是阴性的。性别的真正价值

在于其提供了冗余，强迫句子的各部分互相配合，这样就增加了一种清晰性。

编程语言则避免冗余。我们就表达一次否定，典型地就是仅用一个字符（!）。但是编程语言的确有性别，但我们称为类型。一般来说，你的动词和名字之间需要在配型上彼此配合（就像在阿拉伯语、希伯来语和俄语，以及其他语言中一样）。因为这种加入的冗余，当你有两个矩阵时，你需要matrix_multiply（a, b）；而你有两个复数时，你需要complex_multiply（a, b）。

操作符重载将会消除冗余性，不管你有一对矩阵、复数、自然数或者集合，都写作a*b。这里有从一个非常不错的关于减少冗余的代价的文章引用的一段：“当你看到代码i=j*5;，在C语言中你至少知道，j是被5乘并把结果存储在i中。但是你如果在C++中看到这样的代码段落，你就什么都不知道。”问题就在于你并不知道“*”意味着什么，除非你检查了j的类型，逐一检查了j的类型的继承关系来决定你的意思是哪个版本的“*”，然后你可以从头开始识别i以及它在j的类型下与=的关系。

关于重载我有一个自己的原则，用_Generic或者任何其他方法后，如果用户不知道输入的类型，那么还能不能得到正确的结果？例如对int、float、double重载能够满足这个要求。GNU科学库提供了gsl_complex类型来代表复数，但是C标准允许complex double的类型；你的重载函数应该同等对待这两种类型。

就像目前你已经看见过的例子，C的风格遵循我介绍过的性别匹配的规则，例如：

```
//add two vectors in the GNU Scientific Library
gsl_vector *v1, *v2;
gsl_vector_add(v1, v2);

//Open a GLib I/O channel for reading at a given filename.
GError *e;
GIOChannel *f = g_io_channel_new_file("indata.csv", "r", &e);
```

这增加了很多语句，而且当你有10行处理同样的结构，事情看起来就开始重复了。但是每一行都很清晰。

_Generic

C通过C11中的_Generic关键词提供了有限的重载支持。这个关键词基于输入的类型计算出一个值，这使得你可以写一些宏来把一些类型整合在一起。

当我们面临快速增长的类型的时候，我们需要类型普适的函数。一些系统提供了数目巨大的精确类型，但是每种新的类型都是我们必须支持的。例如，GNU科学计算库提供了一个复数类型、一个复数向量类型和一个向量类型——然后又有了C的复数类型。我们可以合理地把那4种类型在一起做乘法运算，这意味着我们需要16个函数。例11-14列出了其中的几个函数；如果你不是一个复数向量的狂热爱好者，你完全可以把这个例子忽略，并跳到我们整理后的结果。

例11-14　具体的实现，如果你对GSL的复数类型感兴趣（complex.c）

```
#include "cplx.h"                  //gsl_cplx_from_c99; see below.
#include <gsl/gsl_blas.h>          //gsl_blas_ddot
#include <gsl/gsl_complex_math.h> //gsl_complex_mul(_real)
```

```
gsl_vector_complex *cvec_dot_gslcplx(gsl_vector_complex *v, gsl_complex x)
{
    gsl_vector_complex *out = gsl_vector_complex_alloc(v->size);
    for (int i=0; i< v->size; i++)
        gsl_vector_complex_set(out, i,
                           gsl_complex_mul(x, gsl_vector_complex_get(v, i)
));
    return out;
}

gsl_vector_complex *vec_dot_gslcplx(gsl_vector *v, gsl_complex x){
    gsl_vector_complex *out = gsl_vector_complex_alloc(v->size);
    for (int i=0; i< v->size; i++)
        gsl_vector_complex_set(out, i,
                           gsl_complex_mul_real(x, gsl_vector_get(v, i)));
    return out;
}

gsl_vector_complex *cvec_dot_c(gsl_vector_complex *v, complex double x){
    return cvec_dot_gslcplx(v, gsl_cplx_from_c99(x));
}

gsl_vector_complex *vec_dot_c(gsl_vector *v, complex double x){
    return vec_dot_gslcplx(v, gsl_cplx_from_c99(x));
}

complex double ddot (complex double x, complex double y){return x*y;} ❶

void gsl_vector_complex_print(gsl_vector_complex *v){
    for (int i=0; i< v->size; i++) {
        gsl_complex x = gsl_vector_complex_get(v, i);
        printf("%4g+%4gi%c",GSL_REAL(x), GSL_IMAG(x),i<v->size-1 ? '\t':'\
n');
    }
}
```

❶ C自身的复数就用一个简单的*来做乘法，就像实数。

整理工作发生在例11-5的头文件中。它用_Generic根据输入类型从例11-14中挑选对应函数。第一个参数（即“控制标志”）是不被计算的，而是简单地做一下类型检查，再根据类型来选择 一个_Generic指令的值。我们想要基于两个类型来选择一个函数，所以第一个宏会选择第

二个和第三个宏。

例11-15　利用_Generic来为一团乱麻提供一个简单的前端处理（cplx.h）

```
#include <complex.h> //nice names for C's complex types
#include <gsl/gsl_vector.h> //gsl_vector_complex

gsl_vector_complex *cvec_dot_gslcplx(gsl_vector_complex *v, gsl_complex x)
;
gsl_vector_complex *vec_dot_gslcplx(gsl_vector *v, gsl_complex x);
gsl_vector_complex *cvec_dot_c(gsl_vector_complex *v, complex double x);
gsl_vector_complex *vec_dot_c(gsl_vector *v, complex double x);
void gsl_vector_complex_print(gsl_vector_complex *v);

#define gsl_cplx_from_c99(x) (gsl_complex){.dat= {creal(x), cimag(x)}} ❶

complex double ddot (complex double x, complex double y);

#define dot(x,y) _Generic((x),                                  \      ❷
                  gsl_vector*: dot_given_vec(y),                \
                  gsl_vector_complex*: dot_given_cplx_vec(y), \
                  default: ddot)((x),(y))

#define dot_given_vec(y) _Generic((y),                          \
                        gsl_complex: vec_dot_gslcplx,                \
                        default: vec_dot_c)

#define dot_given_cplx_vec(y) _Generic((y),                     \
                        gsl_complex: cvec_dot_gslcplx,               \
                        default: cvec_dot_c)
```

❶ gsl_complex和C99双精度复数都是一个双成员的数组，包括一个双精度实数跟着一个双精度虚数[参见GSL手册与C99和C11，§6.2.5（13）]。我们所要做的一切就是构建合适的结构——复合常量是建立即时结构最好的方式。

❷ 第一个x实际上并没有被计算，仅仅检查了它的类型。这意味着

一个类似dot（x++, y）的调用将只增加x一次。

在例11-16，程序又简化了：我们可以用dot来计算一个gsl_vector和一个gsl_complex的积，或者一个gsl_vector_complex和一个C复数的积等类似的很多组合。当然，你仍然需要知道输出的类型，因为一个纯量和一个纯量相乘的返回值是一个纯量，不是一个向量，所以输出的使用依赖于输入类型。类型的爆炸式增长是一个根本问题，但是_Generic至少提供了一个基本的改善。

例11-16　回报：我们（几乎）可以使用dot而不用考虑输入类型（simple_cplx.c）

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

int main(){
    int complex a = 1+2I;                                                    ❶
    complex double b = 2+I;
    gsl_complex c = gsl_cplx_from_c99(a);

    gsl_vector *v = gsl_vector_alloc(8);
    for (int i=0; i< v->size; i++) gsl_vector_set(v, i, i/8.);

    complex double adotb = dot(a, b);                                        ❷
    printf("(1+2i) dot (2+i): %g + %gi\n", creal(adotb), cimag(adotb)); ❸

    printf("v dot 2:\n");
    double d = 2;
    gsl_vector_complex_print(dot(v, d));

    printf("v dot (1+2i):\n");
    gsl_vector_complex *vc = dot(v, a);
    gsl_vector_complex_print(vc);

    printf("v dot (1+2i) again:\n");
    gsl_vector_complex_print(dot(v, c));
}
```

❶ 声明complex与声明const有一点相像：complex int和int complex都是有效的。

❷ 最后的好处：这个函数将4次使用dot函数，每个都有不同的输入。

❸ 这里是C得到一个复数的实数和虚数部分的方法。

11.5 引用计数

本章剩余的部分演示了一些如何在new/copy/free等函数模板中添加引用计数的例子。添加引用计数并不难，我们可以利用这些例子做一些扩展，加入处理更多真实世界的考量并做一些有趣的事情。因为本章所有有趣的扩展和变化，毕竟都是基于结构外加伴随的函数实现的，这也是目前大量C库所主要实现的方式。

第一个例子展示了一个只有一个结构供使用的小型库，它的目的是把整个文件读入一个字符串。把正本Moby Dick放在内存中的一个字符串中根本不是什么难事，但是有一千个复制是非常浪费的。所以我们不是去复制非常长的数据串，而是只对不同的开始和结束点做出标记。

现在我们有几个不同的字符串视图，我们需要在字符串不再有任何视图与之关联的时候，一次性地释放字符串。归功于对象框架，这些都是非常容易实现的。

第二个例子，一个基于代理的group formation，存在一个同样的问题：只要它们有成员，这个组就应该存在，但是当最后一个成员离开的

时候就需要被释放。

11.5.1 示例：一个子字符串对象

让多个对象指向同一个字符串的技巧是在结构中加入一个引用计数成员。把四个模板成员修改如下。

- 类型定义中包含一个整数的指针，称为refs。它仅被初始化一次（通过new函数），同时所有的复制（通过copy函数获得）将共享这个字符串和这个引用计数器。
- new函数初始化refs指针，并将其设为*refs=1。
- copy函数把原始的结构复制到新的复制，并把引用计数增加。
- free函数减少引用计数器，并且如果计数器归零，就释放共享的字符串。

例11-17是这个字符串处理例子的头文件部分——fstr.h，其中引入了代表一个字符串的片段的关键数据结构，以及一个代表这些字符串片段的列表的辅助结构。

例11-17　冰山的一角（fstr.h）

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

typedef struct {          ❶
    char *data;
    size_t start, end;
    int* refs;
} fstr_s;

fstr_s *fstr_new(char const *filename);
fstr_s *fstr_copy(fstr_s const *in, size_t start, size_t len);
```

```
void fstr_show(fstr_s const *fstr);
void fstr_free(fstr_s *in);

typedef struct {          ❷
    fstr_s **strings;
    int count;
} fstr_list;

fstr_list fstr_split (fstr_s const *in, gchar const *start_pattern);
void fstr_list_free(fstr_list in);
```

❶ 我希望这些头文件已经开始叫你感到厌倦。它们就是一遍又一遍的同样的typedef/new/copy/free组合而已。

❷ 它是一个辅助的结构，并不是一个很完整的对象。请注意fstr_split函数返回这个列表，而不是指向列表的指针。

例11-18展示库的实现，fstr.c。它用GLib来在文本文件中读取，并为Perl兼容的常规表达式解析。程序中标记了本节开头强调的那些步骤，你可以学习如何使用refs成员来实现引用计数。

例11-18　一个代表子字符串的对象（fstr.c）

```
#include "fstr.h"
#include "string_utilities.h"

fstr_s *fstr_new(char const *filename){
    fstr_s *out = malloc(sizeof(fstr_s));
    *out = (fstr_s){.start=0, .refs=malloc(sizeof(int))};     ❶
    out->data = string_from_file(filename);
    out->end = out->data ? strlen(out->data): 0;
    *out->refs = 1;
    return out;
}

fstr_s *fstr_copy(fstr_s const *in, size_t start, size_t len){ ❷
    fstr_s *out = malloc(sizeof(fstr_s));
    *out=*in;
    out->start += start;
```

```
    if (in->end > out->start + len)
        out->end = out->start + len;
     (*out->refs)++;                                                    ❸
    return out;
}
void fstr_free(fstr_s *in){                                             ❹
    (*in->refs)--;
    if (!*in->refs) {
        free(in->data);
        free(in->refs);
    }
    free(in);
}

fstr_list fstr_split (fstr_s const *in, gchar const *start_pattern){
    if (!in->data) return (fstr_list){ };

    fstr_s **out=malloc(sizeof(fstr_s*));
    int outlen = 1;
    out[0] = fstr_copy(in, 0, in->end);

    GRegex *start_regex = g_regex_new (start_pattern, 0, 0, NULL);
    gint mstart=0, mend=0;
    fstr_s *remaining = fstr_copy(in, 0, in->end);
    do {                                                                ❺
        GMatchInfo *start_info;
        g_regex_match(start_regex, &remaining->data[remaining->start],
                                   0, &start_info);
        g_match_info_fetch_pos(start_info, 0, &mstart, &mend);
        g_match_info_free(start_info);
        if (mend > 0 && mend < remaining->end - remaining->start){ ❻
            out = realloc(out, ++outlen * sizeof(fstr_s*));
            out[outlen-1] = fstr_copy(remaining, mend, remaining->end-mend
);
            out[outlen-2]->end = remaining->start + mstart;
            remaining->start += mend;
        } else break;
    } while (1);

    fstr_free(remaining);
    g_regex_unref(start_regex);
    return (fstr_list){.strings=out, .count=outlen};
}

void fstr_list_free(fstr_list in){
    for (int i=0; i< in.count; i++){
        fstr_free(in.strings[i]);
```

```
    }
    free(in.strings);
}

void fstr_show(fstr_s const *fstr){
    printf("%.*s", (int)fstr->end-fstr->start, &fstr->data[fstr->start]);
}
```

❶ 对于一个新的fstr_s，引用计数被设定为1。除此之外，这个函数与new模版函数一样。

❷ copy函数复制传入的fstr_s，并设定给定子字符串的开始和结束点（要确保结束点不要超过输入的fstr_s的结束点）。

❸ 这里就是引用计数得到增加的地方。

❹ 这里就是引用计数被使用的地方，以决定基础数据是否应该被释放。

❺ 这个函数使用 Glib 中与Perl兼容的正则表达式来在给定的标记位置切分字符串，就像我们在第13章讨论过的“解析正则表达式”，正则匹配会给出输入字符串中与正则模式匹配的部分，然后我们可以使用fstr_copy来取出这个部分。然后以匹配位置的尾部作为起点，再进行匹配，取得下一个匹配的文本块。

❻ 否则没有匹配或者超出界限。

最终是一个应用。为了使其可以工作，你将需要一份*Moby Dick*的复制，或者Herman Melville写的Whale。如果你的硬盘里没有，可以尝试一下例11-19的这段指令，从Gutenberg项目下载一份。

例11-19　使用curl来获得Moby Dick的Gutenberg项目版本，然后用sed来去掉Gutenberg头部和脚注；你可能需要使用你的包管理器安装curl（find.moby）

```
if [ ! -e moby ] ; then
curl http://www.gutenberg.org/cache/epub/2701/pg2701.txt \
    / sed -e '1,/START OF THIS PROJECT GUTENBERG/d'      \
    / sed -e '/End of Project Gutenberg/,$d'             \
    > moby
fi
```

现在你有了这本书的一个复制，例11-21把它切分成各章，并用同样的切分函数来计算每章中单词whale(s)和I的数量。请注意fstr结构在这里可以被用作不透明的对象，只能使用new、copy、free、show和split函数。

这个程序需要Glib、fstr.c和本书之前的字符串工具，所以基本的makefile变成例11-20的样子。

例11-20　为cetology程序准备的makefile示例（cetology.make）

```
P=cetology
CFLAGS='pkg-config --cflags glib-2.0' -g -Wall -std=gnu99 -O3
LDLIBS='pkg-config --libs glib-2.0'
objects=fstr.o string_utilities.o
$(P): $(objects)
```

例11-21　一个例子，其中一本书被按章切分，而且每张的字数被计算（cetology.c）

```
#include "fstr.h"

int main(){
    fstr_s *fstr = fstr_new("moby");
    fstr_list chapters = fstr_split(fstr, "\nCHAPTER");
```

```
    for (int i=0; i< chapters.count; i++){
        fstr_list for_the_title=fstr_split(chapters.strings[i],"\\.");
        fstr_show(for_the_title.strings[1]);
        fstr_list me = fstr_split(chapters.strings[i], "\\WI\\W");
        fstr_list whales = fstr_split(chapters.strings[i], "whale(s|)");
        fstr_list words = fstr_split(chapters.strings[i], "\\W");
        printf("\nch %i, words: %i.\t Is: %i\twhales: %i\n", i, words.coun
t-1,
                me.count-1, whales.count-1);

        fstr_list_free(for_the_title);
        fstr_list_free(me);
        fstr_list_free(whales);
        fstr_list_free(words);
    }
    fstr_list_free(chapters);
    fstr_free(fstr);
}
```

为了给你执行这个程序的动力，我不会把结果详细打印出来。但是我会给出一些说明，从中可以看出如果Melville先生要出本书会是多么困难，或者在今天写这本书的博客也不是容易的事情。

- 每章的长度横跨几个数量级。
- Whale其实没怎么被讨论，直到第30章。
- 讲述者角色强烈。甚至在著名的海洋哺乳动物那一章，他使用了第一人称60次，如果少了这么强烈的个人形象，那么这个章节就只是个鲸鱼百科。
- GLib的正则表达式解析器比我所期望的慢了一点。

11.5.2 一个基于代理的组构造模型

这个例子是一个基于代理的group memebership模型。代理是处于一个在（-1, -1）和（1, 1）之间的正方形的二维偏好空间（因为我们将给出绘图显示）。在每一轮中，代理将伴随代理的最佳效用加入这个组。

一个代理的对一个组的最佳效用是“-（与这个组的平均位置之间的距离+*M**成员的数量）”。组的平均位置是组的各成员的平均位置（不包含查询组的代理），而*M*是一个常数，用以衡量这个代理对属于一个大的组的贡献程度相对它们对组的平均值的在贡献度：如果*M*接近0，组的尺寸基本上是无关的，代理制在意近似性；随着*M*变成无穷大，位置称为无关的，而只有组的尺寸有意义。

在一些随机条件下，代理会产生一个新的组。然而，因为每个周期中代理都可以选择一个新的组，代理可能在下一个周期放弃新近分配到的组。

引用计数的问题是相似的，而且这个例子中整个过程也大体相似。

- 类型定义包含一个叫作counter的整数。
- new函数设定counter=1。
- copy函数设定counter++。
- free函数查询if（--counter==0），如果是，则free所有的共享数据；否则，就不去触动任何事物，因为我们知道还有其他的指向这个结构的引用。

同样，通过它的接口函数来改变结构，你完全不需要在使用对象的时候思考内存分配。

仿真花费了125行代码，而且因为我用CWEB来做文档，代码文件整体上几乎是两倍长度（关于阅读和编写CWEB的讨论，请参见2.5.2“用CWEB解释代码”）。由于是解释代码的风格，这应该是非常易读的；甚至如果你有一目十行的习惯，你也可以浏览它一下。如果你手

头有CWEB，你可以生成PDF文件，并用这个格式阅读。

这个程序的输出被设计成可以重定向到Gnuplot，一个以便于自动化闻名的作图程序。这里是一个命令行脚本，其使用内嵌文档来把给定文本重定向到Gnuplot，包括一系列的数据点（用一个e来表计数据序列的结尾）。

```
cat << "------" | gnuplot --persist
set xlabel "Year"
set ylabel "U.S. Presidential elections"
set yrange [0:5]
set key off
plot '-' with boxes
2000, 1
2001, 0
2002, 0
2003, 0
2004, 1
2005, 0
e
------
```

你可能已经通过编程自动对数据产生命令来作图了，用一两个printf来完成作图的设定，然后一个for循环输出数据。进一步地，向Gnuplot发送数据以产生一个动画的序列。

下面的仿真产生了一个像这样的动画，所以你可以通过./groups | gnuplot在屏幕上显示这个动画。打印一个动画是很困难的，所以你必须自己去运行它。你将看到它的效果，尽管这样的行为并不是在仿真中编程实现的，新的组引发附近的组的变动，产生一个均匀分布的空间，以及组位置的整齐分布。政治科学家经常在政党位置中观察到类似的行为：当新的政党进入，现存的政党就会随之调节他们的位置。

现在讨论头文件。我所称的join和exit函数更可能被称为copy和free函数。group_s结构有一个size成员，代表组成员的数量——即引用计数。你可以看到我使用Apophenia和Glib。很明显，组用一个链表表示，且是group.c内私有；维护这个列表需要完整的两行代码，包括一个对g_list_append和g_list_remove的调用（见例11-22）。

例11-22　group_s的公共部分（groups.h）

```
#include <apop.h>
#include <glib.h>

typedef struct {
    gsl_vector *position;
    int id, size;
} group_s;

group_s* group_new(gsl_vector *position);
group_s* group_join(group_s *joinme, gsl_vector *position);
void group_exit(group_s *leaveme, gsl_vector *position);
group_s* group_closest(gsl_vector *position, double mb);
void print_groups();
```

下面是针对定义组对象细节的文件的内容（见例11-23）。

例11-23　group_s对象（groups.w）

```
@ Here in the introductory material, we include the header and specify
the global list of groups that the program makes use of. We'll need
new/copy/free functions for each group.
@c
#include "groups.h"

GList *group_list;
@<new group@>
@<copy group@>
@<free group@>

@ The new group method is boilerplate: we |malloc| some space,
fill the struct using designated initializers, and append the newly formed
```

group to the list.

```
@<new group@>=
group_s *group_new(gsl_vector *position){
    static int id=0;
    group_s *out = malloc(sizeof(group_s));
    *out = (group_s) {.position=apop_vector_copy(position), .id=id++, .siz
e=1};
    group_list = g_list_append(group_list, out);
    return out;
}
```

```
@ When an agent joins a group, the group is 'copied' to the agent, but
there isn't any memory being copied: the group is simply modified to
accommodate the new person. We have to increment the reference count, whic
h
is easy enough, and then modify the mean position. If the mean position
without the $n$th person is $P_{n-1}$, and the $n$th person is at position
$p$, then the new mean position with the person, $P_n$ is the weighted sum
.

$$P_n = \left( (n-1)P_{n-1}/n \right) + p/n.$$

We calculate that for each dimension.
```

```
@<copy group@>=
group_s *group_join(group_s *joinme, gsl_vector *position){
    int n = ++joinme->size; //increment the reference count
    for (int i=0; i< joinme->position->size; i++){
        joinme->position->data[i] *= (n-1.)/n;
        joinme->position->data[i] += position->data[i]/n;
    }
    return joinme;
}
```

```
@ The 'free' function really only frees the group when the reference count
is zero. When it isn't, then we need to run the data-augmenting formula
for the mean in reverse to remove a person.

@<free group@>=
void group_exit(group_s *leaveme, gsl_vector *position){
   int n = leaveme->size--; //lower the reference count
   for (int i=0; i< leaveme->position->size; i++){
        leaveme->position->data[i] -= position->data[i]/n;
        leaveme->position->data[i] *= n/(n-1.);
    }
    if (leaveme->size == 0){ //garbage collect?
        gsl_vector_free(leaveme->position);
```

```
        group_list= g_list_remove(group_list, leaveme);
        free(leaveme);
    }
}
```

@ I played around a lot with different rules **for** how exactly people evaluate the distance to the groups. In the end, I wound up using the L_3 norm. The standard distance is the L_2 norm, aka Euclidian distance, meaning that the distance between (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$. This is L_3, $\sqrt[3]{(x_1-x_2)^3+(y_1-y_2)^3}$.
This and the call to |apop_copy| above are the only calls to the Apophenia library; you could write around them **if** you don't have that library on hand.

```
@<distance@>=
apop_vector_distance(g->position, position, .metric='L', .norm=3)
```

@ By 'closest', I mean the group that provides the greatest benefit, by having the smallest distance minus weighted size. Given the utility function represented by the |dist| line, this is just a simple |**for**| loop to find the smallest distance.

```
@c
group_s *group_closest(gsl_vector *position, double mass_benefit){
    group_s *fave=NULL;
    double smallest_dist=GSL_POSINF;
    for (GList *gl=group_list; gl!= NULL; gl = gl->next){
        group_s *g = gl->data;
        double dist= @<distance@> - mass_benefit*g->size;
        if(dist < smallest_dist){
            smallest_dist = dist;
            fave = g;
        }
    }
    return fave;
}
```

@ Gnuplot is automation-friendly. Here we get an animated simulation with four lines of plotting code. The header |plot '-'| tells the system to plot the data to follow, then we print the (X, Y) positions, one to a line. The final |e| indicates the end of the data set. The main program will set some initial Gnuplot settings.

```
@c
void print_groups(){
    printf("plot '-' with points pointtype 6\n");
    for (GList *gl=group_list; gl!= NULL; gl = gl->next)
        apop_vector_print(((group_s*)gl->data)->position);
    printf("e\n");
}
```

现在我们有了一个group对象与一些用于添加、加入和离开组的接口函数，这样程序文件就可以聚焦在仿真过程中了：定义人员的数组，然后是一个重新检查成员关系和打印的主循环（见例11-24）。

例11-24　基于代理的模型，利用了group_s对象（groupabm.w）

```
@* Initializations.

@ This is the part of the agent-based model with the handlers for the
|people| structures and the procedure itself.

At this point all interface with the groups happens via the
new/join/exit/print functions from |groups.cweb.c|. Thus, there is zero
memory management code in this file--the reference counting guarantees us
that when the last member exits a group, the group will be freed.

@c
#include "groups.h"

int pop=2000,
    periods=200,
    dimension=2;

@ In |main|, we'll initialize a few constants that we can't have as static
variables because they require math.

@<set up more constants@>=
    double  new_group_odds = 1./pop,
            mass_benefit = .7/pop;
    gsl_rng *r = apop_rng_alloc(1234);

@* The |person_s| structure.

@ The people in this simulation are pretty boring: they do not die, and do
not move. So the struct that represents them is simple, with just |positio
```

n|
and a pointer to the group of which the agent is currently a member.

```
@c
typedef struct {
    gsl_vector *position;
    group_s *group;
} person_s;
```

@ The setup routine is also boring, and consists of allocating a uniform random vector in two dimensions.

```
@c
person_s person_setup(gsl_rng *r){
    gsl_vector *posn = gsl_vector_alloc(dimension);
    for (int i=0; i< dimension; i++)
        gsl_vector_set(posn, i, 2*gsl_rng_uniform(r)-1);
    return (person_s){.position=posn};
}
```

@* Group membership.

@ At the outset of this function, the person leaves its group. Then, the decision is only whether to form a new group or join an existing one.

```
@c
void check_membership(person_s *p, gsl_rng *r,
                      double mass_benefit, double new_group_odds){
    group_exit(p->group, p->position);
    p->group = (gsl_rng_uniform(r) < new_group_odds)
             ? @<form a new group@>
             : @<join the closest group@>;
}
```

```
@
@<form a new group@>=
group_new(p->position)
```

```
@
@<join the closest group@>=
group_join(group_closest(p->position, mass_benefit), p->position)
```

@* Setting up.

@ The initialization of the population. Using CWEB's macros, it is at this point

```
self-documenting.

@c
void init(person_s *people, int pop, gsl_rng *r){
    @<position everybody@>
    @<start with ten groups@>
    @<everybody joins a group@>
}

@
@<position everybody@>=
    for (int i=0; i< pop; i++)
        people[i] = person_setup(r);

@ The first ten people in our list form new groups, but because everybody'
s
position is random, this is assigning the ten groups at random.

@<start with ten groups@>=
    for (int i=0; i< 10; i++)
        people[i].group = group_new(people[i].position);

@
@<everybody joins a group@>=
    for (int i=10; i< pop; i++)
        people[i].group = group_join(people[i%10].group, people[i].positio
n);

@* Plotting with Gnuplot.

@ This is the header for Gnuplot. I arrived at it by playing around on
Gnuplot's command line, then writing down my final picks for settings here
.

@<print the Gnuplot header@>=
printf("unset key;set xrange [-1:1]\nset yrange [-1:1]\n");

@ Gnuplot animation simply consists of sending a sequence of plot statemen
ts.
@<plot one animation frame@>=
print_groups();

@* |main|.

@ The |main| routine consists of a few setup steps, and a simple loop:
calculate a new state, then plot it.
```

```
@c
int main(){
    @<set up more constants@>
    person_s people[pop];
    init(people, pop, r);

    @<print the Gnuplot header@>
    for (int t=0; t< periods; t++){
        for (int i=0; i< pop; i++)
            check_membership(&people[i], r, mass_benefit, new_group_odds);
        @<plot one animation frame@>
    }
}
```

11.5.3 结论

本节讨论了对象基本模式的几个例子：一个带有相关的new/copy/free成员的结构。这里我给出了好多的例子，这是因为最近十几年，这是一种公认的组织库代码的好的方法。

那些不带有new/copy/free的例子演示了如何扩展现有的结构，在扩展结构本身这个方向，你学习了如何使用一个匿名包含来包装一个结构。

对于函数来说，我们介绍了如何对于结构的不同实例来使用不同的函数，将一个函数包含在一个结构中，你可以生成一个分发函数。利用Vtable，分发函数可以在结构已经完成并发布的情况下继续扩展。你看到了_Generic关键字，可以根据表达式的类型来呼叫不同的函数。

你可以决定是否让你的代码更易读，还是提升你的用户界面。这些附加的表现形式确实让别人的代码更易读了。你也许有一个10年前写的一个代码库，并且需要增加一些新的功能。本章的方法这个时候就特别有用了：你可以扩展这些结构，并加入新的可能的函数。

[1] 因为没人读脚注，所以这里我做一个坦白；我喜欢M4，这是从1970年就存在的一个宏处理语言。你的系统中可能有M4，因为它是POSIX标准，而且Autoconf也使用它。除了广泛存在外，它还有两个特性，使得它很特别，也很有用。第一，它被设计成用于搜索内嵌的宏，那些内嵌的宏在一个为其他目的撰写的文件中，例如autoconf产生的脚本文件，或者HTML文件，或者C语言等。等宏出来完后，输出可以是符合标准的shell脚本、HTML，后者是C语言，其中没有M4的任何踪迹了。第二，你可以写一些宏以便产生其他的宏。C预处理不能这么做。在一个项目中，我需要产生很多的Vtable。我用M4宏产生了类型检查函数和C宏。代码没有太多的冗余，当我把m4的过滤步骤放到makefile中以后，我给别人分发了纯粹的C代码。如果你想在未过滤的源头工作，你可以使用m4，它非常流行。

第12章　多线程

过去几年卖的所有计算机，甚至于很多电话，都是多核的。如果你在一台有键盘和显示器的电脑上阅读本书，你可以通过以下的命令知道你的电脑有多少个核。

- Linux: grep cores /proc/cpuinfo。
- Mac: sysctl hw.logicalcpu。
- Cygwin: env | grep NUMBER_OF_PROCESSORS。

单线程的程序并不会充分利用硬件给你的全部的性能。幸运的是，将现有的代码转换成多线程程序并不是很困难——事实上，大部分情况下，你只需要在你的代码中加上一行额外的代码。本章我会讲述以下的内容：

- 对于现存的书写并行C代码的标准和规定做一个快速的介绍。
- 只需要一行的openMP代码就可以使得你的循环变成多线程。
- 编译OpenMP和pthreads程序的时候，你需要使用的编译器选项。
- 当使用这一魔法行的时候，你应该考虑哪些安全事项。
- 实现map-reduce，需要把一行扩展成其他的子句。
- 并行一些彼此独立的任务，如UI或者后台的基于GUI的程序。
- C的_Thread_local关键字，可以有与全局的static变量对应的线程私有复制
- 关键区域和互斥。

- OpenMP中的原子变量。
- 顺序一致性的介绍，以及你为何需要它。
- POSIX线程，它们与OpenMP有什么不同。
- 通过C原子定义原子标量变量。
- 通过C原子定义原子结构。

这一章介绍了一些标准教科书中不会介绍的内容。在我对市场的调查中，我没有发现任何一本C语言的教科书介绍了OpenMP。所以这里我会多介绍一些，也许多到你根本不需要再参考专门的关于线程理论的书了。

有一本书专门介绍并发编程，覆盖了很多本书没有介绍的知识。书的名字是：*The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications; Multicore Application Programming: for Windows, Linux, and Oracle Solaris;*。或者你也可以阅读*Introduction to Parallel Computing*（*2nd Edition*）。本书只演示了一些最安全的同步方法，通过一些细微的调整，你可以获得更好的性能。我并没有介绍cache优化，也没有介绍一些有用的OpenMP 编译器指令（pragma）（用你的搜索引擎搜索它们就可以了）。

12.1 环境

直到2011的修订版本，才把线程机制加入C标准。这有点晚了，其他的人已经提供了这种机制，所以下面有一些选择：

- POSIX线程，该线程标准定义在1995年的POSIX V1中。
  pthread_create函数把一个特定的函数分配给一个线程，这样你不得

不写出一个恰当的函数接口，和一个伴生的结构。

- Windows也有自己的线程系统，有点像pthreads。例如，CreateThread函数接受一个函数和一个指向参数的指针，这有点像pthread_create。
- OpenMP是一个使用#pragma的标准，它包含一系列的库，告诉编译器何时、如何开始多线程。这可以让你只通过一行代码就把顺序运行的循环变成一个多线程运行的循环。第一个OpenMP标准是在1998年出现的。
- C标准库规范中包含了定义多线程函数和原子变量操作的头文件。

使用哪一个取决于你的目标环境、你的目的和你的个人喜好。OpenMP使用起来非常简单，所以比起其他的多线程系统，它出错的机会比较少。它几乎被所有的主要的编译器所支持，甚至Visual Studio也支持它。但是clang对它的支持在2014年的末期还在开发中。支持标准C多线程与原子的编译器和库函数并不是统一的。如果你不能依赖OpenMP和＃pragmas，那么pthreads在任何支持POSIX的系统（即使是MinGW）下都是可用的。

这里有其他的选项，如MPI（Message Passing Interface，用于在网络节点之间交谈）或者OpenCL（对GPU处理非常有用）。在POSIX系统，你可以使用fork系统调用来高效地产生你程序的两个克隆，它们共享内存但是彼此独立运行。

配方

我们并不需要太多语法，在大部分时间里，我们只需要：

- 告诉编译器立刻开始几个线程的方法。例如（Nabokov-1962）在404行[真是巧合了]包含了这个说明，剩余的代码在两个线程之间切换。
- 标志一个点的方法，在这个点上一个线程会停止，但是主线程会继续。一些情况下，如前面早一些的例子，分界线隐式地存在于一段的末尾，但是你也可以显式地收集线程。
- 一个用来标示某段代码不应该使用多线程的方法，这是因为它们并不是线程安全的。例如，一个线程调整一个数组的尺寸为20，但是另一个线程调整这个数组的尺寸为30，这个时候会发生什么？即使调整尺寸对我们人类来说只需要几毫秒，但是如果我们把时间放慢，我们会发现即使像x＋＋这样简单的语句，也是由一系列操作组成的，而在这其中，另外一个进程会切换进来并产生冲突。使用OpenMP编译器命令，这些不可共享的部分会被标记为critical regions，在pthread影响的系统，它们会被标记为mutexes（mutual和exclusion的组合体）。
- 处理能够被多钱程同时操作的变量的方法。方法包括接受一个全局变量并生成一个线程局部的复制，和在每一次使用变量的时候，使用mini-mutex的语法。

12.2 OpenMP

作为一个例子，让我们多线程实现一个单词计数程序，我会从例11-21中借用几个字符串处理的工具。为了和多线程的程序区分开来，这些工具都在它们自己的文件中，参考例12-1。

例12-1　单词计数器，将整个的文件读入内存，并且在非单词字母

的位置进行拆分

```
#include "string_utilities.h"

int wc(char *docname){                                    ❶
    char *doc = string_from_file(docname);
    if (!doc) return 0;
    char *delimiters = " '~!@#$%^&*()_-+={[]}|\\;:\",<>./?\n";
    ok_array *words = ok_array_new(doc, delimiters);   ❷
    if (!words) return 0;
    double out= words->length;
    ok_array_free(words);
    return out;
}
```

❶ string_from_file将给定文档读入字符串，如例9-6那样。

❷ 同样借用了字符串功能库，这个函数用指定的分隔符分隔字符串，我们只是想对字符串中的单词进行计数。

例12-2在命令行指定的文件上调用单词计数函数。在程序中，main只是一个调用wc的循环，然后把单个的数字相加进行汇总。

例12-2　通过加入一行程序，我们可以在不同的线程中运行for循环（openmp_wc.c）

```
#include "stopif.h"
#include "wordcount.c"

int main(int argc, char **argv){
    argc--;
    argv++;                                              ❶
    Stopif(!argc, return 0, "Please give some file names on the command li
ne.");
    int count[argc];

    #pragma omp parallel for                             ❷
    for (int i=0; i< argc; i++){
        count[i] = wc(argv[i]);
```

```
        printf("%s:\t%i\n", argv[i], count[i]);
    }

    long int sum=0;
    for (int i=0; i< argc; i++) sum+=count[i];
    printf("Σ:\t%li\n", sum);
}
```

❶ argv［0］是程序的名字，我们应该跳过它。命令行中剩下的参数就是我们要进行单词计数的文件名字了。

❷ 把这一行加入后，for循环就运行在并行的线程中了。

OpenMP利用下面这一行来使得单线程程序变成了多线程：

```
#pragma omp parallel for
```

它指定了后面跟随的for 循环要分解成不同的片段，并且根据系统目前的状况，来确定用多少个线程来运行这些片段。本例中，我兑现了我的诺言，用一行程序把一个普通程序变成了一个并行的程序。

OpenMP会计算出你的系统可以运行多少个线程，并把工作分拆到这些线程中去。有的时候，如果你要手工设置要运行*N*个线程，你可以在运行程序之前设置一个环境变量：

```
export OMP_NUM_THREADS=N
```

或者在你的程序中使用一个C函数：

```
#include <omp.h>
omp_set_num_threads(N);
```

这些方法可能是在你把线程设置为1的时候最有用了。如果你想把

线程数设置成你的计算机的处理器数，你可以使用：

```
#include <omp.h>
omp_set_num_threads(omp_get_num_procs());
```

提示

利用＃define定义的宏不能扩展成＃pragma，如果你想将一个宏并行化，你必须使用_Pragma操作符［C99和C11 § 6.10.9］，操作符的输入被（在语言的官方标准中）非字符串化并用于pragma命令。例如：

```
#include <stdio.h>

#define pfor(...) _Pragma("omp parallel for") \
    for(__VA_ARGS__)
```

```
int main(){
    pfor(int i=0; i< 1000; i++){
        printf("%i\n", i);
    }
}
```

你只能在_Pragma的括号中使用一个单个的字符串，当你需要使用多个字符串的时候，你需要用一个子宏把所有的输入当成一个字符串。下面是一个预处理块，它使用这种方式定义了OMP_critical宏。如果_OPENMP定义了，它会利用一个给定的标签扩展成OpenMP 的critcial块，否者它会被空白代替。

```
#ifdef _OPENMP
   #define PRAGMA(x) _Pragma(#x)
   #define OMP_critical(tag) PRAGMA(omp critical(tag))
#else
   #define OMP_critical(tag)
#endif
```

12.2.1 编译OpenMP、pthreads和C原子（atom）

对于gcc和clang（clang对OpenMp的支持在有些平台上还在开发中）编译这个程序需要加入-fopenmp编译开关。如果你需要一个单独的链接步骤，把 -fopenmp也加入到链接步骤（编译器会知道需要哪些库并完成哪些需要的步骤）。对于pthreads，你需要加入-pthread开关，C的原子性支持（在gcc中，当本书还在撰写中）需要链接原子库，所以如果你同时使用以上三个，你需要把这些行加入你的makefile文件。

```
CFLAGS=-g -Wall -O3 -fopenmp -pthread
LDLIBS=-fopenmp -latomic
```

如果你在你的OpenMP项目中使用autoconf，你需要把下面的这行加入configur.ac脚本：

```
AC_OPENMP
```

这会产生一个$OPENMP_CFLAGS变量，然后你需要在makefile.am中加入下面的开关。例如：

```
AM_CFLAGS = $(OPENMP_CFLAGS) -g -Wall -O3 …
AM_LDFLAGS = $(OPENMP_CFLAGS) $(SQLITE_LDFLAGS) $(MYSQL_LDFLAGS)
```

利用这三行代码，autoconf会在任何已知的支持OpenMP的平台正确

地编译你的代码。

OpenMP标准要求如果一个编译器接受OpenMP 编译器命令，那么_OPENMP变量必须被定义。你可以按照要求把#ifdef _OPENMP命令放到你的代码中。

一旦你的程序被编译成了thread_wc，就使用./thread_wc ' find ～ -type f ' 对你的目录中的每个文件运行单词计数功能。你可以使用top命令来查看有几个wc命令被调用了。

12.2.2 冲突

现在我们有了需要的语法来让程序变成多线程程序，我们能保证它一定工作吗？对于简单的应用，我们可以验证循环中的每一次迭代和其他的迭代之间没有什么交互。但是对于其他的应用，我们需要小心了。

为了验证一组线程是否会工作，我们需要知道对于每个变量会发生的所有的副作用。

- 如果一个变量对一个线程是私有的，我们可以保证它就像在一个单线程的程序中，不会发生任何的冲突。循环中的循环变量，上面例子中的i，对于每一个线程来说都是私有的，在循环内部声明的变量也是私有的。
- 如果一个变量被一些线程读取，但是不会被这些线程的任何一个线程改写，这个时候你也是安全的。这里并不是量子物理的世界，所以读一个变量并不会改变它的状态（目前我还没有介绍C原子性标志，如果你不设置它，那么你就不能读取它）。

- 如果一个变量被一个线程改写，但是不会被任何线程读，这里也没有竞争，这个线程其实也是私有的。
- 如果一个变量在多个线程之间共享，也就是说，它被一个线程写，同时又被其他的线程读或者写，现在你有了真正的问题，本章余下的部分都是在讨论这种情况。

第一个结论就是，如果有可能，我们应该避免写入共享的变量。回头看看我们的程序是如何做到这一点的。所有的线程都使用count数组，但是i次迭代只触及数组中的第i个元素，所以数组中的每个元素其实是线程私有的。另外，就像argv一样，count数组并没有在循环的时候改变尺寸、释放等。后面我们还会介绍进一步的方法，那就是如何避免使用count数组。

我们并不知道printf函数用了哪些内部变量，但是我们可以查看C标准，C标准告诉我们标准库中所有作用于输入输出流的操作（几乎是stdio.h中的所有操作）都是线程安全的。所以我们可以调用printf函数而不用担心互相发生干扰。

当我写下这个例子的时候，需要加点小心，以便所有的条件都被满足。但是有一些建议，例如避免全局变量等，就算是我们写单线程程序的时候，这些建议也是对的。后C99声明变量的风格在这里是非常有用，因为如果你把变量声明在一个线程块的时候，这个变量毫无疑问对这个线程是私有的。

另外，OpenMP的omp parallel for 编译命令只理解简单的循环：迭代子是整数类型，每次迭代根据固定的步长来增加或者减少，最后的终止条件是迭代子和一个循环无关的变量比较大小。对一个固定数组的每

一个元素应用相同的操作都符合这种模式。

12.2.3 映射缩减

单词计数程序有一个通用的模式：每一个线程做一个独立的工作，把输入映射到输出，但是我们真正感兴趣的是把所有独立的输出缩减或归并（reduce）到一个单独的聚合体。OpenMP支持这种映射缩减模式。例12-3把count数组替换为一个单个的变量total_wc，并且在OpenMP编译器命令上加入了reduction（+:total_wc）。这里，编译器高效地完成了以下的工作，它把每个线程单独的输出组合到total_wc这一单个的变量。

例12-3　在一个for循环上应用映射缩减要求扩展#programa omp parallel子句（mapreduce_wc.c）

```
#include "stopif.h"
#include "wordcount.c"

int main(int argc, char **argv){
    argc--;
    argv++;
    Stopif(!argc, return 0, "Please give some file names on the command li
ne.");
    long int total_wc = 0;

    #pragma omp parallel for \
     reduction(+:total_wc)                                    ❶
    for (int i=0; i< argc; i++){
        long int this_count = wc(argv[i]);
        total_wc += this_count;
        printf("%s:\t%li\n", argv[i], this_count);
    }

    printf("Σ:\t%li\n", total_wc);
}
```

❶ 在omp parallel for加入缩减子句，告诉编译器这个变量保存所有线程的汇总。

另外，这里有个限制：在缩减reduction（+:variable）语句里面的加号，只能被以下几个基本的运算符号所取代，如代数运算（+,*,-）,位运算（&,|, ^）, 逻辑运算（&&,||）。否则你必须回退到像上面那样使用count数组，然后在循环结束后书写你自己的缩减代码（例12-5演示了如何求取最大值）。另外，不要忘记了在循环开始之前初始化缩减变量。

12.2.4 多任务

上面介绍了我们有一个数组，并在每一个数组元素上执行相同的操作，与此不同的是，我们可能有两个完全不同的操作，它们彼此是独立的，并且可以并行地运算。例如，有图形用户界面的程序通常把界面放到一个线程中，而把后台处理部分放到另外一个线程中。所以后台处理线程变慢了，并不会使得图形界面变得没有反应。自然，对这一部分的编译器指令是parallel sections：

```
#pragma omp parallel sections
{
    #pragma omp section
    {
    //Everything in this block happens only in one thread
    UI_starting_fn();
    }

    #pragma omp section
    {
    //Everything in this block happens only in one other thread
    backend_starting_fn();
    }
}
```

这里有一些OpenMP的一些特性我没有介绍，不过你也许会感兴趣。

Simd

单个指令，多数据。一些处理器有能力在一个向量的每一个元素上应用相同的操作。这并不是在所有的处理器上都适用，并且与运行在多核上的多线程不同。具体可以查看#pragma omp simd或者你的编译器手册，因为有些编译器会尽可能地自动使得某些操作simd化。

#pragma omp task

当任务的数量在开始的时候是未知的，你可以使用#pragma omp task来开始一个新的线程。例如，你可以用一个线程遍历树结构，当你到达一个叶节点的时候，开始另外一个线程来处理这个叶节点。

#pragma omp cancel

你可以使用多线程来搜索某个东西，当一个线程发现了目标以后，其他的线程就没必要再继续下去了。使用#pragma omp cancel（在 pthread 中使用 pthread_ cancel）来取消其他的线程。

另外，我必须加上一个警告，否则一些读者会把#pragma放到每一个单独的for循环前面。产生线程是有额外的代价的，下面的代码：

```
int x = 0;
#pragma omp parallel for reduction(+:x)
for (int i=0; i< 10; i++){
     x++;
}
```

会花费更多的时间去产生线程，而不是去递增变量x，如果不用多线程，我保证程序会运行得更快。你可以自由地使用多线程，但是你最好能确保多线程确实是提高了程序的性能。

每个线程的建立和取消都是有代价的，这个事实提醒我们也许少一点的线程比更多的线程更有效。例如，如果你有一个嵌套在另外一个循环内部的循环，并行化外部循环要比并行化内部循环更有效。

如果你能确保你的线程片段不会写入任何的共享变量，并且所有的函数都是线程安全的，那么你不需要在继续读下去了，只要在正确的位置插入#pragma omp parallel for 或者 parallel sections，然后速度就提高了。下面的章节，或者说其实大部分的多线程编程都在集中讨论如何修改共享资源的策略。

12.3 线程本地

静态变量——即使声明在#pragma omp parallel区域内的那些变量，也默认都是所有线程共享的。你可以在编译器命令中使用threadprivate子句来生成一个线程的私有复制。

```
static int state;
#pragma omp parallel for threadprivate(state)
for (int i=0; i< 100; i++)
     …
```

基于一般的共识，系统会保持住每一个线程私有变量的值。如果static_x在线程4结束的时候是2.7，那么在线程4下一次开始的时候，这个值还应该是2.7。总有一个主线程在运行；在并行区域外面，主线程

保持它的静态变量的复制。

C的_Thread_local关键字利用同样的方法来分割静态变量。在C语言中，一个线程本地的静态变量“生命期就是整个线程的执行过程，保存的值在线程开始的时候被初始化”在一个并行区域内，如果我们从线程4读入，那么这个值和从另外一个并行区域内从线程4读入是一致的。这个行为和OpenMP是一致的；如果它们在不同的线程被读入，那么C标准指定线程本地存储在每一个并行区域被重新初始化。

在每一个并行区域外面都有一个主线程［并没有显式地开始，但是C11 §5.1.2.4（1）暗示这一点］，所以在主线程中线程私有的静态变量和传统的与程序有共同生命期的静态变量很相似。

Gcc和clang提供__thread关键字，它是在标准加入_Thread_local关键字之前的gcc的一个扩展。你可以在一个函数中这样使用：

```
static __thread int i; //GCC/clang-specific; works today.
// or
static _Thread_local int i; //C11, when your compiler implements it.
```

函数外面，static关键字是可选的，因为它是默认的。标准要求threads.h头文件中定义一个_Thread_local的别名 thread_local，就像stdbool.h头文件中定义一个_Bool的一个别名bool一样。

你可以通过一块预处理检查块来确定你要使用哪一个，像下面这样，把字符串threadlocal在给定的条件下设置成正确的内容。

```
#undef threadlocal
#if __STDC_VERSION__ > 201100L
    #define threadlocal _Thread_local
#elif defined(__APPLE__)
```

```
    #define threadlocal //as of this writing, not yet implemented.
#elif (defined(__GNUC__) || defined(__clang__)) && !defined(threadlocal)
    #define threadlocal __thread
#else
    #define threadlocal
#endif
```

非静态变量本地化

如果一个变量分割成为所有线程中的私有复制，我们就必须使得变量在每一个线程中被初始化，并且要指定在线程结束后我们要采取的操作。OpenMP中引入的thread private()子句用一个初始化值初始化静态变量。并且在离开多线程区域的时候保存这个静态变量，以便重新进入多线程区域的时候再次使用。

你已经见过这样一个子句了：reduction（+:var）子句告诉OpenMP在每个线程内把复制的变量初始化为0，如果是乘法就是1。然后让每一个线程自己做加法或者减法，然后在线程退出的时候，把每一个私有的复制累加到var这个变量中。

在并行区域外声明的非静态变量默认上是所有线程共享的。你可以在#pragma omp parallel行加入firstprivate（localvar）子句来为每一个线程加入localvar的私有复制。一个私有复制加入到每一个线程，并且它的值初始化为线程开始时的值。线程结束后，这些私有复制的变量被销毁。原有的值不会被触及。加入lastprivate（localvar）复制最后一个线程的最后一个值（在for循环中哪个最大的i的值，或者是sections列表中的最后一个元素）到外面的变量中去。在firstprivate（localvar）和lastprivate（localvar）子句中使用相同的变量，这种用法很常见。

12.4 共享资源

目前，我们强调了使用私有变量的价值，并且演示了如何将一个静态变量变成每一个线程中的私有变量。但是有的时候，有些资源必须要共享，critical region是一种最简单的保护方法。它把一段代码标志为每次只允许一个线程运行它。就像很多OpenMp的结构一样，它作用于后续的程序块：

```
#pragma omp critical (a_private_block)
{
    //interesting code here.
}
```

我们保证这段代码每次只有一个线程能够运行它。如果一个线程正在运行它，而另外一个线程试图运行它的时候，它必须在程序块的开始处等待，直到正在运行这个程序块的线程结束运行，并离开这个程序块为止。

这叫作堵塞（blocking），被堵塞的线程在某段时间是不活动的。这并不是很有效率。但是没效率的代码比错误的代码要强。

（a_private_block）带括号的形式，是允许你链接关键区域的名字，例如保护在代码中不同位置使用的单一资源。如果你不想一个线程在读一个结构的时候，另外一个线程再写入这个结构，你应该使用这种形式：

```
#pragma omp critical (delicate_struct_region)
{
    delicate_struct_update(ds);
}
```

```
[other code here]

#pragma omp critical (delicate_struct_region)
{
    delicate_struct_read(ds);
}
```

我们保证在任何时候，在总体的两个代码段落上，只有一个线程在运行，这样就不会出现在调用delicate_struct_update的同时调用delicate_struct_read。中间的代码运行在通常的线程中。

警告

名字从技术上来说是可以忽略的，但是未命名的关键区域被当成属于同一个组。对一个小型的程序，这是可行的（就像你在internet上找到的那些示例代码）。对于不太简单的程序，这就不太合理了。给每一个关键区域命名，你可以避免把两个不相关的关键区域连接起来。

考虑下面一个问题，对于某一个数字（质数或者非质数），找到它有多少个因子。例如，数字18可以被六个正整数整除：1，2，3，6，9，18。数字13有两个因子，1和13，这代表13是一个质数。

发现质数是很容易的，从1到10000000的范围内，有664579的质数。但是在这个范围内，只有446个数只有3个因子，有6个数有7个因子。只有一个数有17个因子。其他的模式也非常容易发现：有2228418个数恰好有8个因子。

例12-4是发现因子数量的程序，我们利用OpenMP使用多线程，主要使用了两个数组。第一个是包含一千万个数的数组——factor_ct。对任何一个大于2的数，初始化每个元素的值为2，这是因为每一个数都可以被1和自身整除。然后，如果它能被2整除，那么我们就将元素的值加1。如果它能被3整除，元素的值再加1。与此类推，直到它能被五百万整除（只有一千万这一个数最大可以被五百万整除）。在过程的结束，我们知道了每个有多少个因子。你可以使用一个for 循环把这些因子的数量通过printf打印出来。

然后，建立另外一个数组来计数多少个数有1、2、3……因子。在做这个之前，我们必须发现最大的因子数量，以便知道我们要建立的数组的大小；然后我们可以遍历factor_ct数组以便开始计数。

每一步都可以通过#pragma omp parallel for来并行处理，但是冲突也会发生。用于计算乘以5的那个线程和用于乘以7的那个线程很容易同时修改factor_ct[35]这个元素。为了防止写入冲突，我们把写入的代码标记为关键区域：

```
#pragma omp critical (factor)
factor_ct[i]++;
```

提示

这些编译器命令作用于紧随其后的程序块。块用一对花括号来表示，如果没有花括号，一行代码会形成一个块。

当一个线程增加factor_ct[30]的时候，它没有必要阻止其他的线程修改factor_ct[33]，关键区域是关于特定的某块代码的。如果某块代码与某个资源相关联，那么使用关键区域是有意义的，但是目前我们需要保护的是一千万个独立的资源，这个时候我们需要使用互斥和原子变量。

互斥（mutex）是mutual exclusion的缩写。就像上面使用过的关键区域一样，它也被用于堵塞线程。但是互斥是一个结构，所以我们可以有一个数组，其中包含一千万个互斥。在写元素i之前，我们锁上互斥i，当写入操作结束后我们释放互斥i，有点像我们在使用第i个关键区域。代码如下：

```
omp_lock_t locks[1e7];
for (long int i=0; i< lock_ct; i++)
    omp_init_lock(&locks[i]);

#pragma omp parallel for
for (long int scale=2; scale\*i < max; scale++) {
        omp_set_lock(&locks[scale*i]);
        factor_ct[scale*i]++;
        omp_unset_lock(&locks[scale*i]);
     }
```

omp_set_lock函数其实是一个写入-设置函数：如果互斥没被锁住，则锁住它并继续；如果互斥已经被锁住了，堵塞这个线程并等待，直到锁住那个互斥的线程达到omp_unset_lock并且清除锁住这个状态。

正如要求的那样，我们有效地产生了一千万个独立的关键区域。唯一的问题就是互斥结构自己需要占据空间，并且分配一千万个空间比基本的数学计算本身需要付出更多的工作。在代码中，我给出的解决方案

是我只使用128个互斥，并且锁住i%128个互斥。这意味着两个作用于两个不同数字的独立的线程有1/128的可能不需要互相堵塞。这并不可怕，在我的实验中，比起分配一千万个互斥的方法有很大的速度提高。

编译器理解那些编译器命令，但是互斥确实是一个一般的C语言结构，所以这个例子需要包含<omp.h>这个头文件，以得到它的定义。

例12-4给出了代码；发现最大因子数的代码单独在下面的代码中给出。

例12-4　产生一个因子计数数组，发现这个数组中的最大元素，然后对因子数分别为1，2……的数字分别计数

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h> //malloc
#include <string.h> //memset

#include "openmp_getmax.c"                                    ❶

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    int lock_ct = 128;
    omp_lock_t locks[lock_ct];
    for (long int i=0; i< lock_ct; i++)
        omp_init_lock(&locks[i]);

    factor_ct[0] = 0;                                         ❷
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;

    #pragma omp parallel for
    for (long int i=2; i<= max/2; i++)
        for (long int scale=2; scale*i < max; scale++) {
                omp_set_lock(&locks[scale*i % lock_ct]);      ❸
                factor_ct[scale*i]++;
```

```
                omp_unset_lock(&locks[scale*i % lock_ct]); ❹
            }
    int max_factors = get_max(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        int factors = factor_ct[i];
        omp_set_lock(&locks[factors % lock_ct]);          ❺
        tally[factors]++;
        omp_unset_lock(&locks[factors % lock_ct]);
    }
    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

❶ 看下一个程序列表。

❷ 初始化，定义0和 1是非质数。

❸ 在读操作或者要修改一个变量的写操作之前，锁住互斥。

❹ 在读操作或者要修改一个变量的写操作之后，解锁互斥。

❺ 我重新使用了这些互斥以便节省一个初始化操作，但是这些互斥和上面用到的是彼此独立的。

例12-5发现factor_ct数组中的最大的值，因为OpenMP并不提供max缩减，我们必须维护一个maxes数组并且在其中查找最大的值，数组的长度是omp_get_ max_hread()。一个线程能使用omp_get_thread_num()来发现自己的索引。

例12-5　并行寻找数组中的最大元素（openmp_getmax.c）

```
int get_max(int *array, long int max){
```

```
    int thread_ct = omp_get_max_threads();
    int maxes[thread_ct];
    memset(maxes, 0, sizeof(int)*thread_ct);

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        int this_thread = omp_get_thread_num();
        if (array[i] > maxes[this_thread])
            maxes[this_thread] = array[i];
    }

    int global_max=0;
    for (int i=0; i< thread_ct; i++)
        if (maxes[i] > global_max)
            global_max = maxes[i];
    return global_max;
}
```

在这个例子中，每一个互斥包装一个单一的代码块，就像利用上面的一对关键区域那样，现在你可以使用一个互斥来保护代码中不同地点的同一资源。

```
omp_set_lock(&delicate_lock);
delicate_struct_update(ds);
omp_unset_lock(&delicate_lock);

[other code here]

omp_set_lock(&delicate_lock);
delicate_struct_read(ds);
omp_unset_lock(&delicate_lock);
```

原子

一个原子是小的，是不可再分的元素。原子操作通常借助处理器的特性来工作，OpenMP把它们限制在只能在标量上使用：也就是说用在整数、浮点数，或者有的时候可以用在指针上（内存地址）。C会提供一个原子结构，但是你需要使用互斥来保护这个结构。

但是作用于标量的简单操作非常普遍，在这种情况下，我们可以不使用互斥，而只是使用原子操作，这就像在每次使用变量的时候，有一个隐含的互斥在保护这个变量一样。

你可以用编译器命令来告诉OpenMP你想要使用原子了。

```
#pragma omp atomic read
out = atom;

#pragma omp atomic write seq_cst
atom = out;

#pragma omp atomic update seq_cst
atom ++; //or atom--

#pragma omp atomic update
//or any binary operation: atom *= x, atom /=x, ...
atom -= x;

#pragma omp atomic capture seq_cst
//an update-then-read
out = atom *= 2;
```

seq_cst是可选的，但是这里推荐使用（如果你的编译器支持它）；我一会儿再介绍它。

现在，是编译器的责任来建立正确的指令来确保代码中从一个原子读入并不会被写入原子的代码所影响。

在因子计数这个例子中，所有被互斥保护的资源都是标量，所以我们并不需要使用互斥，原子使得例12-6比起互斥版本的例12-4更短，也更易读。

例12-6　用原子代替互斥（atomic_factors.c）

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h> //malloc
#include <string.h> //memset

#include "openmp_getmax.c"

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    factor_ct[0] = 0;
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;

    #pragma omp parallel for
    for (long int i=2; i<= max/2; i++)
        for (long int scale=2; scale*i < max; scale++) {
                #pragma omp atomic update
                factor_ct[scale*i]++;
            }
    int max_factors = get_max_factors(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        #pragma omp atomic update
        tally[factor_ct[i]]++;
    }

    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

连续一致性

一个好的编译器会以一种数学上与你代码一致的方式记录一系列的操作，但是会更快地运行它。如果一个变量在第10行初始化，但是直到第20行才使用它，也许在第20行一起初始化并使用它比起分别在两行处理它要快。这里是两个线程的例子，取自C11§7.17.3（15），我们用更易读的伪代码表示。

```
x = y = 0;


// 线程 1:

r1 = load(y);

store(x, r1);


//线程2:

r2 = load(x);

store(y, 42);
```

读完以后，好像r2不会是42，因为42被保存在y中，而y在r2赋值语句的后面。如果线程1完全在线程2之前执行，或者在线程2两行代码之间，或者完全在线程2以后，r2都不可能等于42。但是编译器可以颠倒线程2中的两行代码，因为一行是关于r2和x，另外一个是关于y，它们之间没有关联性，所以那一行先运行没有什么关系，一则合理的顺序是：

```
x = y = 0;

store(y, 42); //线程2

r1 = load(y); //线程1

store(x, r1); //线程1

r2 = load(x); //线程2
```

现在，所有的x、y、r1、r2都是42了。

C标准给出了另外一个例子，并且指出“这并不是一个有用的行为，实现上是不允许的”。

这就是seq_cst 子句的由来了：它告诉编译器在给定线程中原子操作的顺序应该和代码文件中的顺序一致。它利用了C原子的顺序一致性，并在OpenMP4.0中被加入。也许你的编译器还不支持它。要注意那些细微之处，编译器会把线程内部那些不相关的代码的顺序打乱。

12.5 pthread

现在让我们用pthread的方法来改写上面的例子。我们有相同的元素：一个分发并汇集线程的方法，并且使用互斥。pthread并没有提供原子变量，但是C语言提供了，参考下面的代码。

代码中主要的不同是用于开始一个新线程的pthread_create 函数通过一个函数指针接受一个函数，因为这个函数接受一个void类型的指针，我们必须要写一个函数相关的结构来接受数据。函数同时返回另外一个结构，即使你定义了一个只针对于函数的typedef，在输入的结构中包含输出的元素要比分别定义输入结构和输出结构简单。

在给出例子的全部内容之前，让我们先提取出关键的部分（这也意味着有些变量目前还没有定义）：

```
tally_s thread_info[thread_ct];
for (int i=0; i< thread_ct; i++){
    thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                               .tally=tally, .max=max, .factor_ct=factor_ct,
                               .mutexes=mutexes, .mutex_ct=mutex_ct};
    pthread_create(&threads[i], NULL, add_tally, &thread_info[i]);
}
for (int t=0; t< thread_ct; t++)
    pthread_join(threads[t], NULL);
```

第一个for循环产生一定数量的线程（针对不同的情况让pthreads动态地决定产生线程的数量是有一定困难的）。它首先建立需要的结构，然后通过呼叫pthread_create函数来调用 add_tally函数，传入与之对应的结构。循环结束后，就有thread_ct个线程在工作了。

下一个循环是汇总步骤，pthread_join函数被堵塞，直到所有其他的线程都完成了工作。这样如果其他的线程还没有结束，我们就不能越过这个for循环。for循环结束后，程序退回到单个主线程的程序。

OpenMP互斥和pthreads的互斥看起来很像。在有些例子中，转换到pthreads互斥只是涉及换一下名字这么简单。

例12-7演示了用pthreads 重新写的程序。把每一个子过程分解成不同的线程，定义一个函数相关的结构，分发和汇总步骤加入了很多代码（我这里重复使用了OpenMp get_max函数）

例12-7　利用pthreads 的因子程序

```
#include <omp.h>    //get_max is still OpenMP
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h> //malloc
#include <string.h> //memset

#include "openmp_getmax.c"

typedef struct {
    long int *tally;
    int *factor_ct;
    int max, thread_ct, this_thread, mutex_ct;
    pthread_mutex_t *mutexes;
} tally_s;

void *add_tally(void *vin){
    tally_s *in = vin;                                                    ❶
    for (long int i=in->this_thread; i < in->max; i += in->thread_ct){
        int factors = in->factor_ct[i];
        pthread_mutex_lock(&in->mutexes[factors % in->mutex_ct]);   ❷
        in->tally[factors]++;
        pthread_mutex_unlock(&in->mutexes[factors % in->mutex_ct]);
    }
    return NULL;
}

typedef struct {
    long int i, max, mutex_ct;
    int *factor_ct;
    pthread_mutex_t *mutexes ;
} one_factor_s;

void *mark_factors(void *vin){
```

```
    one_factor_s *in = vin;
    long int si = 2*in->i;
    for (long int scale=2; si < in->max; scale++, si=scale*in->i) {
        pthread_mutex_lock(&in->mutexes[si % in->mutex_ct]);
        in->factor_ct[si]++;
        pthread_mutex_unlock(&in->mutexes[si % in->mutex_ct]);
    }
    return NULL;
}

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    int thread_ct = 4, mutex_ct = 128;
    pthread_t threads[thread_ct];
    pthread_mutex_t mutexes[mutex_ct];
    for (long int i=0; i< mutex_ct; i++)
        pthread_mutex_init(&mutexes[i], NULL);

    factor_ct[0] = 0;
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;

    one_factor_s x[thread_ct];
    for (long int i=2; i<= max/2; i+=thread_ct){
        for (int t=0;t <thread_ct && t+i <= max/2; t++){//extra threads do
 no harm.
            x[t] = (one_factor_s){.i=i+t, .max=max,
                            .factor_ct=factor_ct, .mutexes=mutexes, .mutex
_ct=mutex_ct};
            pthread_create(&threads[t], NULL, mark_factors, &x[t]); ❸
        }
        for (int t=0; t< thread_ct; t++)
            pthread_join(threads[t], NULL);                          ❹
    }
    FILE *o=fopen("xpt", "w");
    for (long int i=0; i < max; i ++){
        int factors = factor_ct[i];
        fprintf(o, "%i %li\n", factors, i);
    }
    fclose(o);

    int max_factors = get_max(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));
```

```
    tally_s thread_info[thread_ct];
    for (int i=0; i< thread_ct; i++){
        thread_info[i]=(tally_s){.this_thread=i, .thread_ct=thread_ct,
                                 .tally=tally, .max=max, .factor_ct=factor_c
t,
                                 .mutexes=mutexes, .mutex_ct=mutex_ct};
        pthread_create(&threads[i], NULL, add_tally, &thread_info[i]);
    }
    for (int t=0; t< thread_ct; t++)
        pthread_join(threads[t], NULL);

    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

❶ 除了pthread_create结构需要以外，这个临时的typedef tally_s也增加了安全性。我必须在书写输入和输出结构的时候非常小心，但是在main和这里的包装函数，内部的结构还是会得到类型检查的。下周的时间，我把tally改成一个整形数的数组，如果我没有正确地改变其他的部分，那么编译器会发出警告。

❷ pthread 互斥和OpenMp互斥看起来很像。

❸ 这里是线程生成步骤。一个线程信息指针数组在循环之前被声明。然后循环填充这些线程信息指针，利用pthread_create生成新的线程，接着把线程信息指针传入要运行的线程中去。第二个参数控制一些线程的属性，本书这里没有介绍。

❹ 第二个循环产生输出。pthread_join的第二个参数是一个地址，我们线程中的函数mark_factors可以把结果写入到这个地址。

警告

for循环结尾的花括号标志着一个生存范围的结束，所以任何本地声明的变量都被清空。正常来说，直到函数返回的时候，我们才会达到生存范围的结束，但是pthread_create的进入点是线程运行的时候，主线程还在继续，所以这个代码会失败。

```
for (int i=0; i< 10; i++){

    tally_s thread_info = {...};

    pthread_create(&threads[i],

        NULL, add_tally, &thread_info);

}
```

因为&thread_info的内容在add_tally使用的时候被清空了。如果把声明放到循环的外面：

```
tally_s thread_info;

for (int i=0; i< 10; i++){

    thread_info = (tally_s) {...};

    pthread_create(&threads[i],

        NULL, add_tally, &thread_info);

}
```

这也是不会工作的，因为保存在thread_info的内容在循环的第二次迭代的时候被改变了，即使第一次的时候，也只是查看这个位置。这样例子建立了一个函数输入的数组，保证每一个线程的信息都会被保留，而且当下一个线程建立的时候不会被改变。

pthreads能帮助我们做更多的事情吗？这里有更多的候选。例如，pthread_rwlock_t是一个互斥，如果任何一个线程正在写入这个线程，它能堵塞读写操作，但是它并不堵塞并发读操作。pthread_count_t是一个信号，用来立刻堵塞任何未堵塞的并发线程，并且可以用来实现通常的互斥来实现读写锁住。如果你有更大的能力，你会更可能把事情搞砸。在今年的计算机上，你可以使用pthread写出比OpenMp更精致的多线程代码，但是过了一年后，当你在更新的电脑上运行这些代码的时候，这些代码就都是错的了。

OpenMP并没有提及pthread，POSIX标准也没有提及OpenMP，目前还没有一个貌似合法的文档来要求OpenMp 中使用的线程和POSIX中使用的线程必须完全匹配。但是你的编译器的开发者必须要发明一些方法来实现OpenMP，POSIX或者是Windows，或者是C的线程库，如果它们为不同的标准开发不同的线程处理流程，那么他们就需要更辛苦的工作。另外，你的计算机的处理器并没有pthread核或者是分离的OpenMp核：它只有一系列的机器指令用来控制线程，这是编译器的责任把标准或规范中的语法转换成这些一系列的指令。这样来说，如果我们进行混杂或者匹配这些不同的标准，也是合理的。例如我们可以使用OpenMP＃pragma这种简单的语法来开始线程，然后使用pthread的互斥或者C的原子来保护资源，或者使用OpenMp当中，其中的一个部分我们使用pthread。

12.6 C原子

C标准包含两个头文件：stdatomic.h和threads.h，其中指定了用于原子变量和线程的函数和类型。这里我会给出一个例子，利用pthreads使

用多线程并利用C原子来保护变量。

这里有两个原因让我没有使用C线程。第一个原因，我只把经过验证的代码放到本书中，在我写这本书的时候，我还找不到一个编译器/标准库的组合来实现threads.h头文件。这是可以理解的，第二个原因：C线程是基于C++线程的，C++线程是基于Windows和POSIX线程之间的最小交集的，所以C线程的最大特点就是它并没有太多的特性，但是C原子却是一个非常靓丽的例外。

给出类型 my_type，它是一个结构，我们把它声明为原子：

```
_Atomic(my_type) x
```

也许下一次编译器发布的时候，我们可以这样写：

```
_Atomic my_type x
```

这么写可以清楚地表明_Atomic是一个类型修饰符，就像const。对于标准中定义的integer，你可以使用automic_int x, 或者 atomic_bool x 等。

将变量声明为原子性赋予了这些变量一些特性：对于x++、--x、x *=y和其他的简单的二元操作/赋值都将是线程安全的了［C11 §6.5.2.4（2）和§6.5.16.2（3）］。这些操作以及下面介绍的所有的线程操作，都是seq_cst，就像我们在OpenMP“顺序一致性”章节中讨论的原子性一样（事实上，OpenMP V4.0&2.12.6指出OpenMP的原子和C11的原子有相似的行为）。但是其他的操作必须通过原子指定函数来操作。

- 利用atomic_init（&your_var, starting_val） 来初始化，指定开始的

值“同时也初始化任何附加的状态，这些状态在实现原子对象的时候需要保持”。这并不是线程安全的，所以在你分发线程之前，把它们包装进一个互斥或者关键区域内。我们也可以在声明行使用ATOMIC_VAR_INIT宏来达到同样的目的，所以你可以使用：

```
_Atomic int i = ATOMIC_VAR_INIT(12);
//或者
_Atomic int x;
atomic_init(&x, 12);
```

- 使用atomic_store（&your_var,x）把x以线程安全的方式赋值给your_var。
- 使用x=atomic_load（&your_var）以线程安全的方式从your_var读入值并赋值给x。
- 使用x=atomic_exchange（&your_var,y）以线程安全的方式把y写入到 your_var，并且把上一个值复制到x。
- 使用x=atomic_fetch_add（&your_var,7）以线程安全的方式把7加入到your_var，并且把x设置成相加以前的值；atomic_fetch_sub是减法（但是没有 atomic_fetch_mul和 atomic_fetch_div）。

还有很多原子变量，主要是因为C委员会希望未来线程库的实现会使用这些原子变量来产生互斥或者C标准里面的其他的结构。因为我假定你不会设计自己的互斥，我不会介绍这些内容（就像atomic_comapre_exchange_weak和_strong函数，用于比较和交换操作）。

例12-8演示了用原子变量重写的例子。我使用pthreads多线程，所以还有一些解释说明，但是关于互斥的解释被去掉了。

例12-8　通过C语言的原子实现因子程序（c_factors.c）

```
#include <pthread.h>
#include <stdatomic.h>
#include <stdlib.h> //malloc
#include <string.h> //memset
#include <stdio.h>

int get_max_factors(_Atomic(int) *factor_ct, long int max){
    //single-threading to save verbiage.
    int global_max=0;
    for (long int i=0; i< max; i++){
        if (factor_ct[i] > global_max)
            global_max = factor_ct[i];
    }
    return global_max;
}

typedef struct {
    _Atomic(long int) *tally;
    _Atomic(int) *factor_ct;
    int max, thread_ct, this_thread;
} tally_s;

void *add_tally(void *vin){
    tally_s *in = vin;
    for (long int i=in->this_thread; i < in->max; i += in->thread_ct){
        int factors = in->factor_ct[i];
        in->tally[factors]++;                                    ❶
    }
    return NULL;
}

typedef struct {
    long int i, max;
    _Atomic(int) *factor_ct;
} one_factor_s;

void *mark_factors(void *vin){
    one_factor_s *in = vin;
    long int si = 2*in->i;
    for (long int scale=2; si < in->max; scale++, si=scale*in->i) {
        in->factor_ct[si]++;
    }
    return NULL;
}
```

```
int main(){
    long int max = 1e7;
    _Atomic(int) *factor_ct = malloc(sizeof(_Atomic(int))*max);  ❷

    int thread_ct = 4;
    pthread_t threads[thread_ct];

    atomic_init(factor_ct, 0);
    atomic_init(factor_ct+1, 1);
    for (long int i=2; i< max; i++)
        atomic_init(factor_ct+i, 2);

    one_factor_s x[thread_ct];
    for (long int i=2; i<= max/2; i+=thread_ct){
        for (int t=0; t < thread_ct && t+i <= max/2; t++){
            x[t] = (one_factor_s){.i=i+t, .max=max,
                                  .factor_ct=factor_ct};
            pthread_create(&threads[t], NULL, mark_factors, x+t);
        }
        for (int t=0; t< thread_ct && t+i <=max/2; t++)
            pthread_join(threads[t], NULL);
    }

    int max_factors = get_max_factors(factor_ct, max);
    _Atomic(long int) tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    tally_s thread_info[thread_ct];
    for (int i=0; i< thread_ct; i++){
        thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                                   .tally=tally, .max=max,
                                   .factor_ct=factor_ct};
        pthread_create(&threads[i], NULL, add_tally, thread_info+i);
    }
    for (int t=0; t< thread_ct; t++)
        pthread_join(threads[t], NULL);

    for (int i=0; i<max_factors+1; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

❶ 之前，我们用互斥或者#pragma omp atomic来保护这一行。因为tally数组中的元素被声明为原子，我们保证简单的数学运算，如增加等

本身就是线程安全的。

❷ _Atomic关键字是一个类型修饰符，就像const一样，一个原子int的尺寸和一个一般的int的尺寸并不需要一致。

原子结构

结构可以是原子的。但是“存取一个原子结构或联合的成员会导致未定义的行为” [C11 §6.5.2.3（5）]，这指出了当我们使用它们的时候要遵守一定的流程：

（1）把共享的原子结构复制到同样类型的私有的非原子结构。

（2）工作在私有的复制的结构上。

（3）把修改后的私有结构复制回原子结构。

如果有两个线程可以修改同一个结构，你不能保证你的结构在步骤（1）的读取和步骤（3）的写入之间不会被修改。所以你可能还需要每次只有一个线程可以写入，或者通过设计实现，或者通过互斥实现。但是你不需要在读取的时候使用互斥了。

这里是一个精致的质数发现器。在本例子使用的淘汰方法（一个Sieve of Eratosthenes的变化版本）已经证明在发现质数的过程中比其他方法要快，但是本例子主要用于演示原子结构。

我要检查一个候选不能被除了它自己以外的任何数整除。但是如果一个候选数不会被3整除，也不会被5整除，那么我们知道它一定不会被15整除，所以我们只是检查是否一个数字能被更小的质数整除。另外我

们没必要检查候选数的后一半，因为最大的因子满足2*因子＝候选数。这样，伪代码如下：

```
for (candidate in 2 to a million){
    is_prime = true
    for (test in (the primes less than candidate/2))
        if ((candidates/test) has no remainder)
            is_prime = false
}
```

目前唯一的问题就是保持小于candidate/2的质数的那个列表。我们需要一个能够修改尺寸的列表，这意味着我们必须使用realloc。我将使用一个原始的数组，但是没有结尾标记，所以我们也需要保持长度。这是使用原子结构的一个好机会，因为数组和长度必须保持同步。

在例12-9中，prime_list是一个所有线程共享的结构。你可以看见它的地址作为函数的参数被传递了好多次，其他的时候我们在atomic_int、atomic_store和 atomic_load中使用它。add_a_prime函数是唯一的修改它的函数，我们使用上面给出的流程，先把它复制到一个私有的结构，然后再复制回来。它被一个互斥包装，因为同时的reallocs会是一个灾难。

Test_a_number 函数有一个值得关注的细节：它一直等待，直到prime_list已经有了candidate/2范围内的质数，否则有些质数会丢失。这是一个方便的特性；你可以查看我们并没有进入deadlock，其中每个线程都在等待其他的线程。这以后，算法就如上面的伪代码所示了。注意这段代码中没有任何地方使用互斥。因为它只是使用atomic_load读入了结构。

例12-9　使用原子结构发现质数（c_primes.c）

```
#include <stdio.h>
#include <stdatomic.h>
#include <stdlib.h> //malloc
#include <stdbool.h>
#include <pthread.h>

typedef struct {                                    ❶
    long int *plist;
    long int length;
    long int max;
} prime_s;

int add_a_prime(_Atomic (prime_s) *pin, long int new_prime){
    prime_s p = atomic_load(pin);                   ❷
    p.length++;
    p.plist = realloc(p.plist, sizeof(long int) * p.length);
    if (!p.plist) return 1;
    p.plist[p.length-1] = new_prime;
    if (new_prime > p.max) p.max = new_prime;
    atomic_store(pin, p);
    return 0;
}

typedef struct{
    long int i;
    _Atomic (prime_s) *prime_list;
    pthread_mutex_t *mutex;
} test_s;

void* test_a_number(void *vin){
    test_s *in = vin;
    long int i = in->i;
    prime_s pview;
    do {
        pview = atomic_load(in->prime_list);
    } while (pview.max*2 < i);

    bool is_prime = true;
    for (int j=0; j < pview.length; j++)
        if (!(i % pview.plist[j])){
            is_prime = false;
            break;
        }
```

```
    if (is_prime){
        pthread_mutex_lock(in->mutex);            ❸
        int retval = add_a_prime(in->prime_list, i);
        if (retval) {printf("Too many primes.\n"); exit(0);}
        pthread_mutex_unlock(in->mutex);
    }
    return NULL;
}

int main(){
    prime_s inits = {.plist=NULL, .length=0, .max=0};
    _Atomic (prime_s) prime_list = ATOMIC_VAR_INIT(inits);
    pthread_mutex_t m;
    pthread_mutex_init(&m, NULL);

    int thread_ct = 3;
    test_s ts[thread_ct];
    pthread_t threads[thread_ct];

    add_a_prime(&prime_list, 2);
    long int max = 1e6;
    for (long int i=3; i< max; i+=thread_ct){
        for (int t=0; t < thread_ct && t+i < max; t++){
            ts[t] = (test_s) {.i = i+t, .prime_list=&prime_list, .mutex=&m
};
            pthread_create(threads+t, NULL, test_a_number, ts+t);
        }
        for (int t=0; t< thread_ct && t+i <max; t++)
            pthread_join(threads[t], NULL);
    }

    prime_s pview = atomic_load(&prime_list);
    for (int j=0; j < pview.length; j++)
        printf("%li\n", pview.plist[j]);
}
```

❶ 列表本身和列表的长度在重新分配的时候必须保持一致，所以我们把它放到一个结构里，并且只声明结构的原子实例。

❷ 这个函数使用流程把原子结构装入一个非原子结构的一个复制，修改这个复制，然后使用atomic_store把这个结构复制回原子结构的版本。这并不是线程安全的，所以每次必须确保只有一个线程调用它。

❸ 因为add_a_prime不是线程安全的，我们把它包装进一个互斥。

这一章覆盖了如何并行运行代码的很多种候选方法。利用OpenMP，建立代码用来分发和汇总线程非常简单，困难之处在于如何跟踪所有的变量：每一个线程涉及的变量必须被区分处理。

最简单的一类变量就是只读变量，还有那些在线程内部生成并销毁，而不和其他线程打交道的变量。我们应该只写那些没有任何副作用以及不改变输入的函数（指针都是用const修饰符）。我们完全可以安全地并行运行这些程序。某种意义上说，这些函数完全没有时间或者环境的概念：sum函数只是作它名字代表的操作，sum（2,2）总是返回4，不管我们何时运行它，也不管我们多频繁地调用它。事实上，有一种纯函数语言，试图只使用这种类型的函数。

状态变量是那些随着函数运行改变值的变量。一旦包含状态变量，函数不再是时间无关了。一个用于查询银行账号余额的函数，今天运行也许会返回一个很大的数，但是当你明天运行的时候，也许只会返回一个很小的数了。那些遵守纯粹函数哲学的开发人员认为我们应该避免使用状态变量。但是它们不可避免，因为我们周围的世界就是充满状态的，而我们的程序需要描述这个世界。当你使用纯函数的时候，你不知道你能有多长时间内不涉及状态。例如Harold Abelson力图不使用状态函数，但是最后却放弃了。不得不承认，这个世界确实是充满了状态的，如银行账号、伪随机数生成，以及电子电路等。

本章主要的内容就是如何在并行环境下处理状态变量，当时间分叉后，你有几种工具使得时间再合并在一处。这些工具包括原子操作、互斥和关键区域。它们都强迫状态以一种有序的方式进行更行。虽然它们

可以完成工作，并且你可以验证并调试你的工作，但是最简单的方法就是在书写处理时间和环境的函数的时候，不使用状态变量。

第13章　函数库

本章将讲述几个库，它们可以让你的日常工作变得更轻松。

在我的印象里，经历了这么多年，C库已经变得不那么墨守成规了。10年前，常用库提供你工作所必需的最小的工具组合，而且期望你从那些基本的工具之上建立便利的和开发人员友好的版本。常用库将需要你来执行所有的内存分配，因为函数库不可以不经允许就抢一块内存。然后，本章展示的所有函数都利用一个“方便”的接口，类似用于cURL的curl_easy…函数，或者SqLite的单一函数以运行所有那些粗暴的数据库操作步骤，或者我们需要通过GLib来配置一个mutex。如果它们需要中间的工作环境以把工作做好，它们就会这么做。所以它们用起来真的很方便。

我将从相对标准和较常用的库开始，然后讲述一些我因为编写一些特殊用途程序而偏好的库,包括SQLite、GNU科学计算库、libxml2和libcURL。我没办法猜测你用C来做什么，但是这些对于很多应用目的来说都是友好的、可靠的系统。

13.1　GLib

既然标准库留下了太多的空缺需要填补，最终出现一个库来填补这个沟堑就是自然而然的事情了。Glib满足了足够的基本计算需要，也就是早期的CompSci能给你的。它已经被实现了普遍的移植（甚至是并不

符合POSIX标准的Windows），且就目前来说，也已经是足够稳定并且是可以依赖的。

我将不再给你一些GLib的示范代码，因为我已经给了你几个了。

- 例2-2中对链接列表的简短介绍。
- “单元测试”的测试框架。
- “Unicode”中的Unicode工具。
- “通用结构”中的Hash表。
- “引用计数”中的将一个文本文件读入内存，并用Perl兼容的常规表达式解析。

在下面的几页里，我会在13.2.2“为巨大的数据集合使用mmap”中提及GLib的贡献， 它为POSIX和Windows包装了mmap。

进一步地，如果你在写一个鼠标-窗口程序，那么将需要一个事件循环来捕捉和派发鼠标和键盘事件；GLib提供了这个功能。它还为POSIX和非POSIX（比如Windows）系统提供了功能完善的文件工具；提供了配置文件的一个简单的解析器；以及一个为较为复杂的过程提供的轻量级的词汇扫描等。

13.2 POSIX

POSIX标准在标准C库上添加了几个有用的函数。考虑到POSIX标准的流行，是值得对它们了解一下的。这里我主要介绍特别有用的两部

分：正则表达式解析和将文件映射到内存。

13.2.1 解析正则表达式

正则表达式是一种用文本表示模式的方法，如一个数字后面跟着一个或多个字母,或者数字-逗号-空格-数字，这一行上没有其他的东西，用基本的正则表达式可以表示为［0-9］+[[:alpha:]]+和^[0-9]+,[0-9]+\$。POSIX标准指定了一系列的C函数来解析那些固定语法的正则表达式，这些函数已经被包装成了上百个工具。我想我几乎每天都在使用它们，或者通过命令行，或者通过POSIX标准工具，如sed、awk或grep，或者通过在C代码中处理文本解析的任务。也许我需要在一个文件中找到一个人的名字，或者有人在字符串中给出了一个日期的范围，如04Apr2009- 12Jun2010。我想把这个字符串解析成六个有用的部分，或者从一个有关鲸鱼类的文献中发现某一个章节等。

提示

如果你想把某一个字符串根据某个单字分隔符分割成不同子字符串，strtok可以完成这个任务。看9.1.4“strtok的赞歌”部分。

但是，我决定不在本书中包含正则表达式的教程了，如果你在Internet搜索“正则表达式教程”，你会得到许多个链接。在Linux系统，你可以使用man 7 regex来发现很多内容。如果你安装了Perl语言，你可以使用man perlre，它主要介绍了Perl语言兼容的正则表达式。[Friedl，

2002]这部分的内容多到可以写一本书。这里我主要介绍在POSIX的C库中如何使用正则表达式。

有三种基本类型的正则表达式：

- 基本的正则表达式（BREs）是第一个草稿本，只有有限的几个字符有特殊的意义，就像＊代表0个或多个前面的原子。如[0-9]*可以代表一个可能的整数。如果我们想表达它是一个特殊的符号，那么我们需要在它前面加上一个反斜杠，前面有正号的一位或多位整数可以表示为：+[0-9]+。
- 扩展正则表达式（EREs）是第二个草稿本，在这个版本中，特殊字符前面不需要反斜杠了，如果我们想表示它是一个一般的字符，我们需要加上反斜杠，现在前面有正号的一位或多位整数可以表示为：+[0-9]+。
- Perl把正则表达式当成自己的核心，它的作者在正则表达式中加入了很多重要的改进，包括向前看／向后看特性，非贪婪修饰符用来匹配最小的匹配，以及注释等。

前两个类型的正则表达式通过定义在POSIX标准中的一小部分函数来实现。它们也许是你标准库的一部分。PCREs通过libpcre库来实现，你可以通过包管理器下载得到。你可以通过man pcreapi来查看这些函数的细节。Glib提供一个方便的、更高层次的libpcre的包装，如例子11-18所示。

正则表达式是POSIX中的基础的部分，本部分给出的例子13-2可以在Linux和Mac上正确地编译，不需要使用任何额外的编译器开关。

```
CFLAGS="-g -Wall -O3 --std=gnu11" make regex
```

POSIX和PCRE需要运行以下四个步骤：

（1）通过regcomp或者pcre_compile编译regex。

（2）编译过的regex通过regexec或者pcre_exec运行字符串。

（3）如果你想抽取在正则表达式中被标记的子字符串，利用regexec或者pcre_exec函数以及偏移量从基本字符串中复制出内容。

（4）释放编译过的regex使用的内存。

最开始的两个步骤和最后的一个步骤分别可以通过一行代码来完成，所以如果你的问题是是否一个字符串匹配一个正则表达式，这个任务很简单。我不会过多介绍regcomp、regexec和regfree的开关以及其他的一些细节，因为关于它们的POSIX标准可以通过Linux和BSD的manpages来查看，你也可以通过很多网站来查看这些手册。

如果你要提取出子串，事情就变得有点复杂了。Regex中的括号代表解析器会根据括号中的子模式获得匹配的内容（哪怕它只匹配空串）。这样，ERE模式“（.）*o*”匹配“*hello*”，作为一个副作用，它把最大可能匹配.的hell存储起来。输入Regexec的第三个参数是在模式中被括号括起来的子表达式的数目；在下面的例子中我把它叫作matchcount，第四个参数是有match+1个 regmatch_t元素的数组。Regmatch_t有两个成员；rm_so, 标记着匹配的开始；rm_eo, 标记着结尾。数组中的第0个元素包含匹配整个regex（假设括号括起来整个模式）的开始和结束位置。下面的元素代表着扩起来的那些子表达式匹配的开始和结束位置。

数组中的顺序与这些括号在模式中的顺序一致。

例13-1演示了一个头文件，这个头文件描述了两个功能函数，它们的示例代码在本节的后面。Regex_match函数＋宏＋结构允许我们命名可选的参数，就像以前10.11.2“可选参数和命名参数”一节介绍的那样，它接受一个字符串和一个regex，产生一个包含子串的数组。

例13-1　一些正则表达式工具的头文件（regex_fns.h）

```
typedef struct {
    const char *string;
    const char *regex;
    char ***substrings;
    _Bool use_case;
} regex_fn_s;

#define regex_match(...) regex_match_base((regex_fn_s){__VA_ARGS__})

int regex_match_base(regex_fn_s in);
char * search_and_replace(char const *base, char const*search, char const
*replace);
```

我们需要一个单独的搜索-替换函数，因为POSIX并没有提供这样的一个函数。如果替换的内容和被替换的内容的长度不一致，那么我们需要把原始的字符串重新分配内存。我们已经有了把字符串分解成子串的工具了，所以search_and_replace利用括号子串来分解字符串，然后在正确的地方插入替换的内容，重新建立新串。

无论如何，它都返回一个NULL，所以你可以通过下面的语句完成全局搜索和替换。

```
char *s2;
while((s2 = search_and_replace(long_string, pattern))){
    char *tmp = long_string;
```

```
    long_string = s2;
    free(tmp);
}
```

这里没有考虑效率：regex_match函数每次都重新编译字符串，如果我们意识到result[1].rm_eo位置的所有内容都是不需要重新搜索的，那么全局的查找和替换会更有效率。这种情况下，我们可以把C语言当成C的原型定义语言：写一个简单的版本，如果效率评估发现程序效率很差，那么我们用更高效的代码来代替它。

例13-2提供了代码，对于上面讨论过的核心内容对应的代码，我都给出了标记，并且在代码结束的时候进行了讨论。结尾的Test函数演示了一些简单的用法。

例13-2　几个用于正则表达式解析的工具（regex.c）

```
#define _GNU_SOURCE //cause stdio.h to include asprintf
#include "stopif.h"
#include <regex.h>
#include "regex_fns.h"
#include <string.h> //strlen
#include <stdlib.h> //malloc, memcpy

static int count_parens(const char *string){                          ❶
    int out = 0;
    int last_was_backslash = 0;
    for(const char *step=string; *step !='\0'; step++){
        if (*step == '\\' && !last_was_backslash){
            last_was_backslash = 1;
            continue;
        }
        if (*step == ')' && !last_was_backslash)
            out++;
        last_was_backslash = 0;
    }
    return out;
}
int regex_match_base(regex_fn_s in){
```

```
    Stopif(!in.string, return -1, "NULL string input");
    Stopif(!in.regex, return -2, "NULL regex input");

    regex_t re;
    int matchcount = 0;
    if (in.substrings) matchcount = count_parens(in.regex);
    regmatch_t result[matchcount+1];
    int compiled_ok = !regcomp(&re,in.regex, REG_EXTENDED          ❷
                                        +(in.use_case ? 0 : REG_ICASE)
                                        +(in.substrings ? 0 : REG_NOSUB))
;
    Stopif(!compiled_ok, return -3, "This regular expression didn't "
                                    "compile: \"%s\"", in.regex);

    int found = !regexec(&re, in.string, matchcount+1, result, 0);  ❸
    if (!found) return 0;
    if (in.substrings){
        *in.substrings = malloc(sizeof(char*) * matchcount);
        char **substrings = *in.substrings;
        //match zero is the whole string; ignore it.
        for (int i=0; i< matchcount; i++){
            if (result[i+1].rm_eo > 0){//GNU peculiarity: match-to-empty m
arked with -1.
               int length_of_match = result[i+1].rm_eo - result[i+1].rm_so
;
               substrings[i] = malloc(strlen(in.string)+1);
               memcpy(substrings[i], in.string + result[i+1].rm_so,
                         length_of_match);
                substrings[i][length_of_match] = '\0';
            } else { //empty match
                substrings[i] = malloc(1);
                substrings[i][0] = '\0';
            }
        }
        in.string += result[0].rm_eo; //end of whole match;
    }
    regfree(&re);                                                  ❹
    return matchcount;
}

char * search_and_replace(char const *base, char const*search, char const
*replace){
    char *regex, *out;
    asprintf(&regex, "(.*)(%s)(.*)", search);                      ❺
    char **substrings;
    int match_ct = regex_match(base, regex, &substrings);
    if(match_ct < 3) return NULL;
```

```
    asprintf(&out, "%s%s%s", substrings[0], replace, substrings[2]);
    for (int i=0; i< match_ct; i++)
        free(substrings[i]);
    free(substrings);
    return out;
}

#ifdef test_regexes
int main(){
    char **substrings;

    int match_ct = regex_match("Hedonism by the alps, savory foods at ever
y meal.",
           "([He]*)do.*a(.*)s, (.*)or.* ([em]*)al", &substrings);
    printf("%i matches:\n", match_ct);
    for (int i=0; i< match_ct; i++){
        printf("[%s] ", substrings[i]);
        free(substrings[i]);
    }
    free(substrings);
    printf("\n\n");

    match_ct = regex_match("", "([[:alpha:]]+) ([[:alpha:]]+)", &substring
s);
    Stopif(match_ct != 0, return 1, "Error: matched a blank");

    printf("Without the L, Plants are: %s",
             search_and_replace("Plants\n", "l", ""));
}
#endif
```

❶ 你需要把一个分配好的数组传入regexec，以保存匹配的子字符串，所以你需要知道有多少个子串。这个函数接受一个ERE，并且对那些没有使用反斜杠来转义的左括号进行计数。

❷ 这里我们把regex编译成regex_t，如果重复使用这个函数就没有效率了，因为每次regex都被重新编译。你可以缓存编译后的正则表达式，这作为一个练习留给读者。

❸ 这是regexec的用法。如果你就是想知道是否有匹配，你可以传

入NULL和0作为匹配的内容和长度。

❹ 不要忘记释放regex_t使用的内部的内存。

❺ 搜索-替换功能是通过字符串分解成（匹配前的所有内容）（匹配的内容）（匹配后的所有内容）来完成的。这里用regex代表。

13.2.2 为巨大的数据集合使用mmap

前面我们提到三种类型的内存（静态、手工和自动），而这里还有第四种：基于磁盘的。利用这种类型，我们可以选择一个在硬盘上的文件，用mmap把它映射到内存的某个地址。

共享库一般是这样工作的：系统找到libwhatever.so，指派内存地址给文件中需要的函数，然后将可以将函数装入内存了。

或者，我们可以实现跨进程共享数据，只要让他们都mmap到同样的文件。

或者，我们可以用这个来把数据结构保存到内存。把一个文件mmap到内存，用memcpy来把你在内存中的数据结构复制到映射的内存，然后这些数据就被存储下来为下次使用。当你的数据结构中有指向其他结构的指针的时候，就会遇到麻烦；把一系列指向结构的指针转化为可存储的形式是序列化（serialization）问题，这里不再赘述。

当然，当数据太大无法装入内存的时候是需要一些处理手法的。一个mmap的阵列的大小是受到你的硬盘的限制的，而不是内存。

例13-3给出了示例代码，主要的工作由load_mmap函数来完成。如

果把它当作一个malloc来使用，那么它需要创建这个文件并把它扩展到一个正确的尺寸；如果你打开了一个已经被打开的文件，它必须先被打开，然后再mmap。

例13-3　一个磁盘文件可以被透明地mmap到内存（mmap.c）

```
#include <stdio.h>
#include <unistd.h> //lseek, write, close
#include <stdlib.h> //exit
#include <fcntl.h> //open
#include <sys/mman.h>
#include "stopif.h"

#define Mapmalloc(number, type, filename, fd) \                    ❶
            load_mmap((filename), &(fd), (number)*sizeof(type), 'y')
#define Mapload(number, type, filename, fd) \
            load_mmap((filename), &(fd), (number)*sizeof(type), 'n')
#define Mapfree(number, type, fd, pointer) \
            releasemmap((pointer), (number)*sizeof(type), (fd))

void *load_mmap(char const *filename, int *fd,size_t size, char make_room)
{  ❷
    *fd=open(filename,
            make_room=='y' ? O_RDWR | O_CREAT | O_TRUNC : O_RDWR,
            (mode_t)0600);
    Stopif(*fd==-1, return NULL, "Error opening file");

    if (make_room=='y'){//Stretch the file size to the size of the (mmappe
d) array
       int result=lseek(*fd, size-1, SEEK_SET);
       Stopif(result==-1, close(*fd); return NULL,
           "Error stretching file with lseek");

       result=write(*fd, "", 1);
       Stopif(result!=1, close(*fd); return NULL,
           "Error writing last byte of the file");
    }
    void *map=mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, *fd, 0);
    Stopif(map==MAP_FAILED, return NULL, "Error mmapping the file");
    return map;
}

int releasemmap(void *map, size_t size, int fd){                   ❸
```

```
    Stopif(munmap(map, size) == -1, return -1, "Error un-mmapping the file
");
    close(fd);
    return 0;
}

int main(int argc, char *argv[]) {
    int fd;
    long int N=1e5+6;
    int *map = Mapmalloc(N, int, "mmapped.bin", fd);

    for (long int i = 0; i <N; ++i) map[i] = i;                    ❹

    Mapfree(N, int, fd, map);

    //Now reopen and do some counting.
    int *readme = Mapload(N, int, "mmapped.bin", fd);

    long long int oddsum=0;
    for (long int i = 0; i <N; ++i) if (readme[i]%2) oddsum += i;
    printf("The sum of odd numbers up to %li: %lli\n", N, oddsum);

    Mapfree(N, int, fd, readme);
}
```

❶ 这里用宏来包装函数，这样你就不用每次都输入sizeof了，而且当为装载而分配内存的时候，我也不想逼自己记住如何调用load_mmap的不同方式。

❷ 宏隐藏了这个函数的两种不同的调用方式。如果只是重新打开现存的数据，文件会被打开，mmap会被调用，结果会被检查，也就是这样了。如果作为一个分配函数调用，我们需要把文件扩展到正确的长度。

❸ 用munmap来释放映射的文件，这类似于malloc的配套函数free；之后再关掉这个文件的句柄。数据会被保存到硬盘上，所以当你第二天回来的时候你可以重新打开它并从你离开的点开始继续。如果你想完整

地移除这个文件，可以用unlink（“filename”）。

❹ 代价：你无法识别map是在磁盘上而不是在通常的内存里。

最后一点：mmap函数是POSIX标准的，所以除了Windows系统和一些嵌入式设备，几乎可以在任何地方找到它。在Windows中，你可以做同样的事情，但是要用不同的函数名和配置；参见CreateFileMapping和MapViewOfFile。GLib把mmap和对应的Windows函数用if POSIX… else if Windows…的结构包装了一下，并被命名为g_mapped_file_new。

13.3 GNU科学计算库

如果听到某人问一个以“我正在试图用C实现某种数值方法……”开头的问题[Press，1992]，正确的反应几乎永远是下载GNU科学计算库（GSL）,因为它已经为你实现了[Gough，2003]。

有些积分函数要比其他函数好，就像7.6“被摒弃的float”一节中所提及的，有些敏感的数值算法可能在多数情况下都无法给你一个精确的结果。所以在计算领域，使用现存的库总是会得到好的报酬。

至少，GSL提供了一个可靠的随机数产生器（C标准的随机数产生器可能会在不同的机器上表现不同，这一点不利于产生可重复的查询），以及方便地拆分子集和处理向量和矩阵。标准线性代数函数、函数最小值计算、统计学基础函数（平均值和方差），以及排列组合结构等，即便你并不是整天和数字打交道的，这些也是对你有用的。

并且，如果你知道什么是特征向量（Eigenvector）、贝叶斯函数，

或者快速傅里叶变换，那么这里就是你可以找到它们的地方。

你已经看过例11-14了，它使用了GSL 的向量和复数。在例13-4中我给出了另外一个使用GSL的例子，虽然你将注意到字符串gsl_仅在这个例子中出现了一两次。GSL是一个历史悠久但提供了常用基本工具的、并期望你以它为基础完成其他工作的绝佳例子。例如，GSL的手册提供了你所需的，让你能够用优化函数来产生丰富成果的示范代码。我认为库应该再为我们提供一些功能，所以我为GSL写了一组包装函数，即Apophenia，一个用于数据建模的库。例如，apop_data数据结构和原始的GSL的矩阵与向量和行/列名字以及一个文本数据的数据绑定起来，这样就使得基本的数据处理结构更加贴近现实世界中的数据看起来的样子。这个库的调用模式看起来也更像第10章中介绍的现代形式。

我们建立了一个优化器，就像我们在10.12“void指针以及它指向的结构”一节介绍的那样，这个程序接受任何函数，并把接受的函数当成一个黑箱。优化器给指定的函数一个输入，然后利用输出值来提高对下一个输入的猜测，以便下一个输入能够产生更大的输出；利用一个足够智慧的搜索算法，一系列的猜测将会收敛到函数最大的输入。利用优化程序其实就是把要优化的函数写成正确的方式，并把它传递给优化器。

例如，我们给定了一列空间（本例中是R^2）中的数据点$x_1, x_2, x_3, ...$，我们想让一点y和所有这些点的距离之和最小。也就是说，给定距离函数D， 我们像让$D(y, x_1)+ D(y, x_2) + D(y, x_3) + ...$ 最小。

优化器需要一个函数能够接受这些数据点和一个候选点，然后计算对每一个 x_i计算$D(y, x_i)$. 这听起来有点像我们在12.2.3“映射缩减”中讨论过的操作一样， apop_map_sum 实现这一点（它甚至使用OpenMP实

现了并行化）。apop_data结构提供了一致的方法，用来提供用于优化操作的*x*点的集合。另外，物理学家和GSL用户通常更喜欢最小值，而经济学家和那些试图从随机事件中找寻规律的人偏好最大值，这个不同可以很轻松地通过加一个符号来解决。最小化距离，就是最大化负距离。

一个优化过程是复杂的（优化器搜索空间的维度是多少？优化器在哪里可以找到参考数据集合？优化器使用哪些搜索策略？）所以，apop_estimate函数接受apop_model结构，结构中有一个函数钩子（hook）以及其他一些信息。把这个距离最小器称为模型有点怪异，但是我们把很多事情都认为是一些统计模型（线性回归，支持向量机，模拟等），都是这么工作的。它们输入数据，利用目标函数找寻最优解，然后把最优解当成给定数据的模型的参数集合。

例13-4实现了distance函数的整个过程，把所有相关的元数据包装进apop_model，所有实质性的动作位于apop_estimate这一行。它返回模型结构的参数值，这个模型的参数值就是与所有点距离最小的那个点。

例13-4　找到与一组输入点距离之和最小的点（gsl_distance.c）

```
#include <apop.h>

double one_dist(gsl_vector *v1, void *v2){
    return apop_vector_distance(v1, v2);
}

long double distance(apop_data *data, apop_model *model){
    gsl_vector *target = model->parameters->vector;
    return -apop_map_sum(data, .fn_vp=one_dist, .param=target);     ❶
}

apop_model *min_distance= &(apop_model){
  .name="Minimum distance to a set of input points.",.p=distance, .vsize=-
1};❷
```

```
int main(){
    apop_data *locations = apop_data_falloc((5, 2),                    ❸
                            1.1, 2.2,
                            4.8, 7.4,
                            2.9, 8.6,
                            -1.3, 3.7,
                            2.9, 1.1);
    Apop_model_add_group(min_distance, apop_mle, .method="NM simplex",   ❹
                                                    .tolerance=1e-5);
    apop_model *est=apop_estimate(locations, min_distance);            ❺
    apop_model_show(est);
}
```

❶ 在输入数据集合的每一行上应用输入函数one_dist。我们需要最小距离，想要一个求最大值的函数求取最小值的常用招数就是取目标函数负值，这就是为什么这里有一个负号。

❷ .vsize元素显示出apop_estimate暗暗做了很多工作。它将分配模型的parameters元素，并且把这个元素设定为-1，表示参数的计数应该等于数据集合中列的数量。

❸ 传递给apop_data_falloc的第一个参数是一个维度列表；然后把这个网格中给定的维度用5个二维的点填充。参考10.4“多列表”一节。

❹ 这一行给模型添加了一组设定，以便设定优化如何进行：使用Nelder-Mead单纯性算法，并且持续尝试直到算法的测量误差小于1e-5。添加.verbose=y以得到一些关于优化搜索的每步迭代的信息。

❺ 现在每个事情都就位了，在最后一行运行优化代码：给定locations数据，找寻能使函数min_distance值最小的那个点。

13.4 SQLite

结构化查询语言（Structured Query Language, SQL）是一个大体上人类可阅读的与数据库交互的语言。因为数据库一般是在磁盘上，它可以是任意大的容量。一个SQL数据库有两个对付大型数据集合的特长：抽取一个数据集合的子集，以及合并数据集合。

本节不会去讲述SQL的太多细节，因为已经有很多可用的教材了。如果我可以引用我自己[Klemens，2008]有关于SQL和从C中使用它的一章，或者就在你惯用的搜索引擎中键入sql tutorial。入门是非常简单的。这里，我主要关注在让你学习如何使用SQLite库本身上。

SQLite仅通过一个C文件和一个头文件就提供了一个数据库。那个文件包括SQL的解析器、与一个磁盘文件交流的各种内部数据结构和函数，以及几十个我们可用于与数据库交互的接口函数。下载文件，把它在你的项目目录中展开，添加sqlite3.o到你的makefile的objects行，你手中就得到了一个完整的SQL数据库引擎。

与数据库互动，你只需要使用几个函数即可，比如打开数据库、关闭数据库、发送一个查询，以及从数据库中得到一行数据。

下面是一些可用的数据库打开和关闭函数：

```
sqlite3 *db=NULL; //The global database handle.

int db_open(char *filename){
    if (filename) sqlite3_open(filename, &db);
    else          sqlite3_open(":memory:", &db);
    if (!db) {printf("The database didn't open.\n"); return 1;}
    return 0;
}

//The database closing function is easy:
sqlite3_close(db);
```

我喜欢只用一个全局的数据库句柄（handle）。如果我需要打开多个数据库，那么我用SQL attach命令来打开另一个数据库。使用一个捆绑（attach）的数据库内的一个表的SQL语句大体为：

```
attach "diskdata.db" as diskdb;
create index diskdb.index1 on diskdb.tab1(col1);
select * from diskdb.tab1 where col1=27;
```

如果第一个数据库句柄在内存中，并且所有的磁盘数据库都被捆绑了，那么你将需要明确地指定哪些新的表或者索引会被写到磁盘上；任何你没有明确指定的东西都被放入一个位于更快的、随用随弃的内存中的临时表中。如果你将表格写到内存中，你可以随后用create table diskdb.saved_table as select * from table_in_memory来写入磁盘。

查询

这里有一个宏，用于向数据库引擎发送没有返回值的SQL。比如attach和create index查询告诉数据库采取某个行动，但是不返回数据。

```
#define ERRCHECK {if (err!=NULL) {printf("%s\n",err); return 0;}}

#define query(...){char *query; asprintf(&query, __VA_ARGS__); \
                    char *err=NULL;                            \
                    sqlite3_exec(db, query, NULL,NULL, &err);  \
                    ERRCHECK                                   \
                    free(query); free(err);}
```

ERRCHECK宏是标准的（参见SQLite手册）。我把对sqlite3_exec的调用包装在一个宏中，这样你可以写类似下面的代码：

```
for (int i=0; i< col_ct; i++)
     query("create index idx%i on data(col%i)", i, i);
```

以类似printf的风格来构建查询其实是从C中调用SQL的惯例，而且你将看到你的查询更多是在运行中被构建出来的，而不是从源代码中逐字构建的。这种形式有一个陷阱：SQL的like分句和printf对%符号的有冲突，所以query（"select *from data where col1 like 'p%%nts' "*）会失败，因为*printf*认为*%%*是对它有意义的。而*query*（*"%s"*，*"select* from data where col1 like 'p%%nts' "）会正常工作。无论如何，在执行中建立查询是如此常用，所以用一个额外的%来修正这个查询所带来的麻烦是值得的。

从SQLite取回数据需要一个回调函数，参见10.12.1“具有通用输入的函数”。下面是一个向屏幕打印的例子。

```
int the_callback(void *ignore_this, int argc, char **argv, char **column){
   for (int i=0; i< argc; i++)
        printf("%s,\t", argv[i]);
    printf("\n");
    return 0;
}

#define query_to_screen(...){                              \
    char *query; asprintf(&query, __VA_ARGS__);            \
    char *err=NULL;                                        \
    sqlite3_exec(db, query, the_callback, NULL, &err); \
    ERRCHECK                                               \
    free(query); free(err);}
```

回调函数的输入看起来和main函数类似：你有一个argv，也就是以一个长度为argc的文本元素的列表。列的名字（也是一个长度为argc的文本列表）则在column中。向屏幕打印意味着直接将传入的字符串输出，这么做很简单，将资料放入数组的做法类似：

```
typedef {
    double *data;
    int rows, cols;
```

```
} array_w_size;

int the_callback(void *array_in, int argc, char **argv, char **column){
    array_w_size *array = array_in;
    *array = realloc(&array->data, sizeof(double)*(++(array->rows))*argc);
    array->cols=argc;
    for (int i=0; i< argc; i++)
        array->data[(array->rows-1)*argc + i] = atof(argv[i]);
}

#define query_to_array(a, ...){\
    char *query; asprintf(&query, __VA_ARGS__);     \
    char *err=NULL;                                 \
    sqlite3_exec(db, query, the_callback, a, &err); \
    ERRCHECK                                        \
    free(query); free(err);}

//sample usage:
array_w_size untrustworthy;
query_to_array(&untrustworthy, "select * from people where age > %i", 30);
```

当我们试图将数值和字符串混合使用的时候，麻烦就来了。在我之前提及的Apophenia库中，我花了一到两页来实现一个处理数值和文本混合的情况。

不过，我们可以高兴地看到，这段代码，配合2个SQLite文件和对makefile中objects行的调整，已经足以为你的程序提供完整的SQL数据库功能了。

13.5 libxml和cURL

cURL库是一个处理各种Internet协议的C库，包括HTTP、HTTPS、POP3、Telnet、SCP和不可缺少的Gopher。如果需要与服务器交互，你也许可以用LibcURL来完成这个任务。就像你将在下面的例子中看到的那样，这个库有一个简易的接口，只需要你提供几个变量，就能够建立

连接。

在我们上网时，类似XML和HTML的标记语言非常流行，所以有必要再介绍一下libxml2。

扩展标记语言（Extensible Markup Language，XML）是用来描述普通文本文件的格式，但是它实际上是一个树状结构的定义。图13-1的前半部分是一个典型的差不多可以阅读的XML数据；后半部分展示了这段文本产生的树状结构。处理一个标记良好的树是一个相对简单的事情：我们可以从根节点开始（通过xmlDocGetRootElement），然后做一个迭代遍历来检查所有的元素，或者我们可以得到所有带par标记的元素，或者我们可以得到所有带title标记且是第二章的子节点的元素等。在下面的示例代码中，//item/title指代树中任何地方的、所有的父节点是一个item的title元素。

libxml2通过对象代表的文档、节点和节点列表来理解标记树的语言。

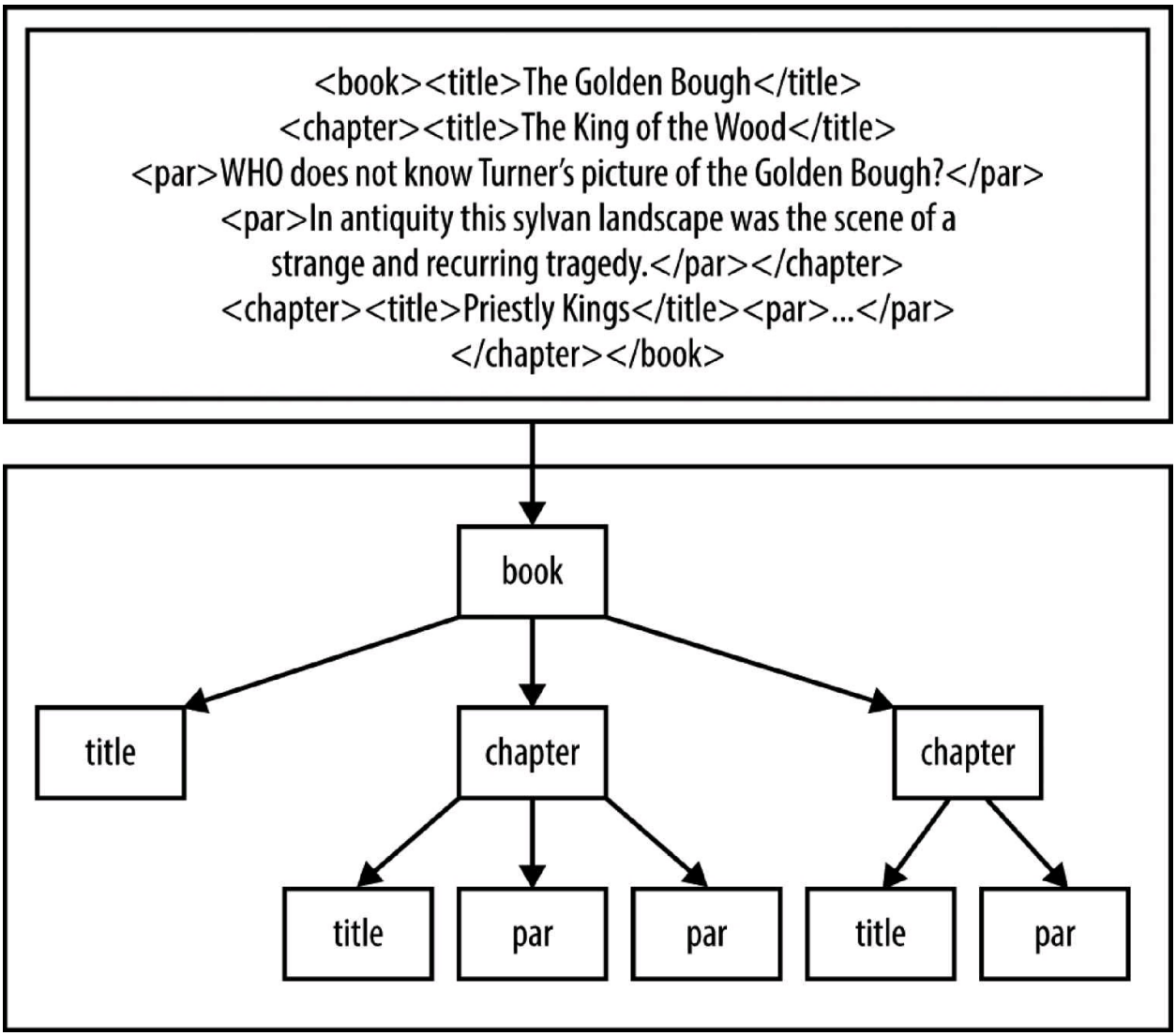


图13-1　一个XML文件及其中内置的树状结构

例13-5代表一个完整的例子。我用Doxygen来做它的文档（参见2.5“编制文档”），因此它看起来会这么长。而且，如果你有浏览长代码块的习惯，那么就试一下，看看是否理解它。如果你手头有Doxygen，你可以试着产生文档并在你的浏览器中查看。

例13-5　将纽约时报头条解析为一个简单格式 （nyt_feed.c）

```
/** \file
```

```
  A program to read in the NYT's headline feed and produce a simple
  HTML page from the headlines. */
#include <stdio.h>
#include <curl/curl.h>
#include <libxml2/libxml/xpath.h>
#include "stopif.h"
/** \mainpage
The front page of the Grey Lady's web site is as gaudy as can be, including

several headlines and sections trying to get your attention, various forma
tting
schemes, and even photographs--in <em>color</em>.

This program reads in the NYT Headlines RSS feed, and writes a simple list
 in
plain HTML. You can then click through to the headline that modestly piques

your attention.

For notes on compilation, see the \ref compilation page.
*/

/** \page compilation Compiling the program

Save the following code to \c makefile.

Notice that cURL has a program, \c curl-config, that behaves like \c pkg-c
onfig,
but is cURL-specific.

\code
CFLAGS =-g -Wall -O3 'curl-config --cflags' -I/usr/include/libxml2
LDLIBS='curl-config --libs ' -lxml2 -lpthread
CC=c99

nyt_feed:
\endcode

Having saved your makefile, use <tt>make nyt_feed</tt> to compile.

Of course, you have to have the development packages for libcurl and libxm
l2
installed for this to work.
*/

//These have in-line Doxygen documentation. The < points to the prior text
//being documented.
```

```
char *rss_url = "http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml"
;
                                        /**< The URL for an NYT RSS feed. */
char *rssfile = "nytimes_feeds.rss"; /**< A local file to write the RSS to
.*/
char *outfile = "now.html"; /**< The output file to open in your browser.*
/

/** Print a list of headlines in HTML format to the outfile, which is over
written.

\param urls The list of urls. This should have been tested for non-NULLness

\param titles The list of titles, also pre-tested to be non-NULL. If the l
ength
     of the \c urls list or the \c titles list is \c NULL, this will crash.

*/
void print_to_html(xmlXPathObjectPtr urls, xmlXPathObjectPtr titles){
    FILE *f = fopen(outfile, "w");
    for (int i=0; i< titles->nodesetval->nodeNr; i++)
        fprintf(f, "<a href=\"%s\">%s</a><br>\n"
                 , xmlNodeGetContent(urls->nodesetval->nodeTab[i])
                 , xmlNodeGetContent(titles->nodesetval->nodeTab[i]));
    fclose(f);
}

/** Parse an RSS feed on the hard drive. This will parse the XML, then find

all nodes matching the XPath for the title elements and all nodes matching
the XPath for the links. Then, it will write those to the outfile.

  \param infile The RSS file in.
*/
int parse(char const *infile){
    const xmlChar *titlepath= (xmlChar*)"//item/title";
    const xmlChar *linkpath= (xmlChar*)"//item/link";

    xmlDocPtr doc = xmlParseFile(infile);
    Stopif(!doc, return -1, "Error: unable to parse file \"%s\"\n", infile
);

    xmlXPathContextPtr context = xmlXPathNewContext(doc);
    Stopif(!context, return -2, "Error: unable to create new XPath context
\n");

    xmlXPathObjectPtr titles = xmlXPathEvalExpression(titlepath, context);
    xmlXPathObjectPtr urls = xmlXPathEvalExpression(linkpath, context);
```

```
    Stopif(!titles || !urls, return -3, "either the Xpath '//item/title' "
                             "or '//item/link' failed.");

    print_to_html(urls, titles);

    xmlXPathFreeObject(titles);
    xmlXPathFreeObject(urls);
    xmlXPathFreeContext(context);
    xmlFreeDoc(doc);
    return 0;
}

/** Use cURL's easy interface to download the current RSS feed.

\param url The URL of the NY Times RSS feed. Any of the ones listed at
           \url http://www.nytimes.com/services/xml/rss/nyt/ should work.

\param outfile The headline file to write to your hard drive. First save
the RSS feed to this location, then overwrite it with the short list of li
nks.

 \return 1==OK, 0==failure.
 */
int get_rss(char const *url, char const *outfile){
    FILE *feedfile = fopen(outfile, "w");
    if (!feedfile) return -1;

    CURL *curl = curl_easy_init();
    if(!curl) return -1;
    curl_easy_setopt(curl, CURLOPT_URL, url);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, feedfile);
    CURLcode res = curl_easy_perform(curl);
    if (res) return -1;

    curl_easy_cleanup(curl);
    fclose(feedfile);
    return 0;
}

int main(void) {
    Stopif(get_rss(rss_url,rssfile), return 1,"failed to download %s to %s.
\n",
                                                    rss_url,rssfile);
    parse(rssfile);
    printf("Wrote headlines to %s. Have a look at it in your browser.\n",
outfile);
}
```

附录A　C101

本附录介绍了一些语言的基本知识，并不是每个人都需要阅读它。

- 如果你有使用一般脚本语言如Python、Ruby 或者Visual Basic编程的经历，那么这个附录可能适合你。我并不需要向你解释什么是变量、函数、循环或者其他基本的程序建造模块，这个附录的主要内容就是介绍C语言和典型的脚本语言之间的区别。
- 如果你很久以前学过C语言，现在有点淡忘了。快速地浏览一遍这个教程就会知道C语言自己的一些特点了。
- 如果你经常使用C语言，你可以不需要读这个教程。你也可以越过，或者快速浏览第二部分的前面的一些内容，它们主要介绍了C语言的一些常见的错误和误解。

别指望着读完这个教程了就成为了C语言的专家——语言中实践的知识才是最宝贵的。但是现在你可以从第二部分开始，开始了解这门语言的一些常见的使用技巧了。

结构

就像1978年Kernighan & Ritchie那本经典的C语言书那样，我用一个输出hello world的程序来开始我的教程。

```
//tutorial/hello.c
#include <stdio.h>
```

```
int main(){
    printf("Hello, world.\n");
}
```

开头的两个斜杠代表这是一个编译器会忽略的注释。

这段代码给出了C语言中几个关键的部分，从结构上来说，C语言中的几乎所有部分就是：

- 一个预处理指令，如＃include<stdio.h>。
- 一个变量或类型的声明（虽然这个程序中没有）。
- 函数块，如main，包含一些要计算的表达式（如printf）。

在讲解预处理指令、声明、块、表达式等内容的定义和使用之前，我们需要明白如何让这个程序运行起来，以便它能和我们打个招呼。

C要求编译的步骤，编译的步骤需要运行一行命令

脚本语言需要另外一个程序来解析你脚本的文本；C语言需要一个编译器来接受你程序的文本，并产生在操作系统能够运行的一个程序。使用编译器有时候比较麻烦，所以你可以用其他的程序间接调用编译器。你的集成开发环境（IDE）一般都有一个编译运行按钮，或者在POSIX系统中，make命令也会帮你运行编译器。

如果你没有编译器和make，利用前面介绍过的1.1“使用包管理器”就可以学习如何获得它们。最简单的方式是，利用包管理器安装gcc、clang或者make。

当编译器和make文件被安装后，如果你把上面的程序保存成了

hello.c以后，你可以使用make来通过以下的命令行来运行编译器。

```
make hello
```

这会产生一个hello的程序，然后你可以通过在命令行中输入或者在你的文件管理器中单击而运行这个程序，接着查看它是否输出了期望的内容。

示例代码中包含了一个makefile文件，它告诉make命令把一些编译开关输入编译器。Make和makefile文件的内容在1.4“使用makefile”部分我们进行了详细的讨论。这里我只是提到一个开关：-Wall。这个开关告诉编译器列出所有技术上来说正确但也许并不是你真实的意图的那些部分。这个部分叫作静态分析。C语言的编译器非常擅长做这个工作。你不应该认为编译过程是一个没有什么用处的过场，相反，你应该把它想象成你把你的代码在运行之前提交给了世界上一流的C语言专家来审核。

如果你使用Mac并且不喜欢-Wall选项，那么参见在1.3.1“一些我喜欢的选项”一部分给出的警告。

很多博客的用户把编译部分当成很重要的一部分，在命令行上，如果你每次通过./yourprogram运行你的程序前，都需要键入make yourprogram这个命令，这多少有点麻烦，你可以写一个别名或者是shell脚本来做这个工作。在POSIX shell， 你可以定义：

```
function crun { make $1 && ./$1; }
```

然后使用：

```
crun hello
```

来编译，如果编译成功，就会直接运行了。

有标准库，并且它是你操作系统的一部分

目前的程序都不是完全独立的，而是链接到一些其他程序也使用的库中的一些公用函数上。库的路径是你的编译器会在硬盘上查找的一个目录；1.3.2“路径”部分介绍了更多的细节。这些库中一个关键的部分是C标准库，IOS C标准定义了这个标准库，标准力争让你的代码有更好的移植性。我们的printf函数就是在C标准库中定义的。

有预处理器

库是以二进制的方式存在的，计算机可以执行它，但是人没有办法阅读它，除非你有读懂二进制的超能力，否则你不能通过查看编译后的库来确保你正确地使用了printf函数。所以库一般都有陪伴它的一些文件——头文件，这些头文件是库中功能的纯文本模式的声明，定义了每一个函数的输入和输出。如果你在代码中包含了正确的头文件，那么编译器就可以验证你是否正确地使用了这个函数，你输入的变量或者类型和库中二进制代码中的函数所期望的是否一致。

预处理的的一个主要活动就是把预处理指令（以#开头）中的文本替换成其他的文本。还有其他的用途（查看预处理器部分），但是这个附录中我只介绍一个用法，那就是包含文件。当预处理看到这个语句：

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

它会在这个语句的位置包含所有stdio.h文件的所有内容。角括号代表文件在包含路径上，包含路径与库路径是不同的（在1.3.2“路径”部分进行了讨论）。如果一个头文件在工作目录，可以使用“myfile.h”。

.h后缀代表文件是一个头文件，头文件是纯文本文件，编译器并不区分头文件和代码文件，但是按照惯例，我们只是把声明放到头文件中。

当预处理完成了它的工作，这个文件中的所有内容或者是一个变量或者类型的声明，或者是一个函数的定义。

有两种类型的注释

```
/* Multiline comments run between a slash-star
   and a star-slash. */

//Single-line comments run from a double-slash to the end of the line.
```

没有**print**关键字

标准库中的printf函数把文本输出到屏幕。它有自己的子语法来精确描述变量是如何输出的。这里我不介绍细节了，因为你可以通过在命令行中输入man 3 printf来查看具体使用时的所有的语法细节，而且你可以在在本书和本附录中看见这些用法的实例。子语法包括一串纯文本，纯文本中包括插入变量标记和一些代表不可见字符的代码，这些不可见字符包括回车和tab等。下面是本附录中你会遇到的printf函数中使用的六个实例。

```
\n A newline
\t A tab
```

```
%i Insert an integer value here
%g Insert a real number in general format here
%s Insert a string of text here
%% Insert a plain percent sign here
```

变量声明

声明在C语言和其他一些脚本语言中有很大的区别，这些脚本语言从存在的变量或者第一次被使用的变量推导出这个变量的类型。上面我已经介绍过，编译步骤是检查和核实你的代码与你的真实意图是否一致的。给每一个变量声明一个类型会让编译器有更多的机会来检查这种一致性。函数声明和类型声明的含义也在于此。

变量必须声明

hello程序没有什么变量，下面是一个声明了基本变量并演示了printf基本用法的程序。注意printf的第一个参数（格式限定符）有三个插入变量标记，所以后面需要跟随三个变量。

```
//tutorial/ten_pi.c
#include <stdio.h>

int main(){
    double pi= 3.14159265;//POSIX defines the constant M_PI in math.h, by the way.
    int count= 10;
    printf("%g times %i = %g.\n", pi, count, pi*count);
}
```

这个程序输出：

```
3.14159 times 10 = 31.4159.
```

本书一直都在使用三个基本的数据类型：int、double和char。它们分别是：整型，双精度浮点型和字符型。

有些人就算去死，也不愿意去声明变量。不过从上面的例子你也看到了，声明变量就是第一次使用变量的时候，把类型名字放到前面而已。当你阅读不熟悉的代码时，每个变量在首先使用的时候都有类型名字，这对你理解代码有很大的帮助。

如果你有同样类型的多个变量，你可以把它们声明在同一行，例如下面的声明：

```
int count=10, count2, count3=30; //count2 is uninitialized.
```

函数也需要声明或者定义

函数的定义描述了一个函数是如何工作的，就像这个简单的函数：

```
int add_two_ints(int a, int b){
    return a+b;
}
```

这个函数接受两个整型数，在函数内把它们称为a和b，并返回一个整数，这个整数是a和b的和。我们可以把声明语句单独写成一个语句，这个语句指定了函数的名字，输入类型（括号内）和输出类型（函数前面）：

```
int add_two_ints(int a, int b);
```

这个声明并没有告诉我们add_two_ints具体如何实现的，但是这些信息足够编译器对所有这个函数的使用来做一致性检查了，编译器确保

输入的类型是整数，并且函数的返回值也被当成一个整数来使用。这个声明可以直接写到代码中，或者通过#include"mydeclarations.h"头文件包含来实现。

一个块被当成一个单元的代码，用一对花括号来扩起来。这样一个函数就是一个声明紧跟着一个代码块，用来实现这个函数。

如果函数的定义在代码中出现在你使用这个函数之前，编译器有了进行一致性检查的所有的信息，你就不需要单独的声明了。因为这样，很多代码被用一种从底向上的方式来书写，main 函数在文件的最下面，上面是main 函数中要使用的函数的定义。在上面是函数中要使用的其他的函数的定义。最上面是头文件，包含所有函数需要使用的函数和类型的定义。

另外，你的函数可以是void类型，这代表着它们什么也不返回。有些函数并不产生输出或者改变变量，但是它们有其他的作用。例如下面的程序是一个把错误信息按照固定格式写入到文件（在你硬盘上生成的）的函数，使用FILE类型，相关的函数被声明在stdio.h中，你可以看到char*类型指定了一个文本字符串。

```
//tutorial/error_print.c
#include <stdio.h>

void error_print(FILE *ef, int error_code, char *msg){
    fprintf(ef, "Error #%i occurred: %s.\n", error_code, msg);
}

int main(){
    FILE *error_file = fopen("example_error_file", "w"); //open for writing

    error_print(error_file, 37, "Out of karma");
}
```

基本的数据类型可以包含在数据和结构中

只有三个基本数据类型，这明显是不够的。我们可以把它们组合成同构类型的数组，或者构成异构类型的结构。

一个数组是一系列相同类型的元素。下面是一个程序，其中声明了一个包含10个整数的数组和20个字符的字符串。

```
//tutorial/item_seven.c
#include <stdio.h>
int intlist[10];

int main(){
    int len=20;
    char string[len];

    intlist[7] = 7;
    snprintf(string, len, "Item seven is %i.", intlist[7]);
    printf("string says: <<%s>>\n", string);
}
```

Snprintf函数根据你给定的最大长度来输出字符串到屏幕，使用如printf 一样的语法。你可以了解如何处理一串字符，另外要明白为什么intlist可以声明在函数的外边，而string必须声明在函数的里面。

一个索引就是从第一个元素开始的偏移量。第一个元素是数组中从0位置开始的，所以它是intlist[0]，10个元素的数组的最后一个元素是intlist[9]。这引起过很多痛苦并引起了很多争论，但是它有存在的意义。

你可以发现在不同的组合语义下都有类似的0开始惯例（Bruckner, Schnittke）。但是很多情况下，我们使用的计数词，第一个， 第二个，第七个等于偏移量是有冲突的：数组中第七个元素是intlist[6]，我这里

称它为“数组中的元素6”。

到后面你就会明白，数组也可以写成一种星号的方式：

```
int *intlist;
```

上面的例子中，一系列字符用char* msg来声明。

可以定义新的结构类型

异构的类型可以组合成一个结构，这个结构可以被当成一个单元。下面的例子声明并使用了ratio_s类型，描述了一个分数的分子、分母以及它最后的小数的值。类型的定义其实就是花括号内部的一系列声明。

当我们使用定义的结构时，你会发现有很多的点操作符。给定结构r， r.numerator是结构的分子成员。表达式（double）den是一个类型转换，把整型的den转换成了double（原因在下面给出）。声明语句以外建立新结构的方法就像是一个类型转换，类型的名字在括号中，跟随着花括号中的点元素。有其他的方法用来初始化一个结构。

```
//tutorial/ratio_s.c
#include <stdio.h>
typedef struct {
    int numerator, denominator;
    double value;
} ratio_s;

ratio_s new_ratio(int num, int den){
    return (ratio_s){.numerator=num, .denominator=den, .value=num/(double)
den};
}

void print_ratio(ratio_s r){
    printf("%i/%i = %g\n", r.numerator, r.denominator, r.value);
}
```

```
ratio_s ratio_add(ratio_s left, ratio_s right){
    return (ratio_s){
        .numerator=left.numerator*right.denominator
                    + right.numerator*left.denominator,
        .denominator=left.denominator * right.denominator,
        .value=left.value + right.value
        };
}

int main(){
    ratio_s twothirds= new_ratio(2, 3);
    ratio_s aquarter= new_ratio(1, 4);
    print_ratio(twothirds);
    print_ratio(aquarter);
    print_ratio(ratio_add(twothirds, aquarter));
}
```

你可以发现一个类型占用多大的空间

sizeof操作符接受一个类型名字，它会告诉你这个类型的一个实例需要占用多少内存。有的时候这很有用。

这个短程序比较了ratio_s结构中int和double的尺寸。Printf中的%zu格式仅仅用于sizeof操作符的返回值。

```
//tutorial/sizeof.c
#include <stdio.h>

typedef struct {
    int numerator, denominator;
    double value;
} ratio_s;

int main(){
    printf("size of two ints: %zu\n", 2*sizeof(int));
    printf("size of two ints: %zu\n", sizeof(int[2]));
    printf("size of a double: %zu\n", sizeof(double));
    printf("size of a ratio_s struct: %zu\n", sizeof(ratio_s));
}
```

没有特殊的字符类型

整数5100和整数51都占用sizeof（int）内存。但是“Hi”和“Hello”是有不同字符的字符串，脚本语言通常有自己的字符串类型，在内部帮助你管理字符序列。C中的字符串就是一个字符数组，很简单、很纯粹。

字符串的末尾有一个NUL字符‘\0’作为标记，这个字符不能输出，而且我们一般也不用太关心它（注意单个的字符我们用单引号，如'x'，而字符串我们用双引号，如"xx"）。函数strlen（mystring）会计算字符串中有几个字符（但是并不包含NUL字符）。为字符串分配多少空间是另外一个话题：你可以简单地声明char pants[1000]= "trousers"，虽然你浪费了NUL后面的991个字符空间。

由于我们用数组来表示字符串，有些事情变得很简单。给定：

```
char* str="Hello";
```

你可以通过下面的方法把“Hello” 变成“Hell”。

```
str[4]='\0';
```

多数情况下，对于字符串，你都会调用库函数来帮助你完成必要的功能，如下面常用的几个：

```
#include <string.h>
char *str1 = "hello", str2[100];
strlen(str1);              //get the length up to but excluding the ’\0’
strncpy(str2, 100, str1); //copy at most 100 bytes from str1 to str2
strncat(str2, 100, str1); //append at most 100 bytes from str1 onto str2
strcmp(str1, str2); //are str1 and str2 different? zero=no, nonzero=yes
snprintf(str2, 100, "str1 says: %s", str1); //write to a string, as above.
```

在第9章，我讨论了可以减轻你负担的一些其他字符串函数，因为有了这些智能的函数，字符串处理变得愉快了很多。

表达式

一个不做任何事，只是声明了一些变量、函数和类型的程序只是一系列名词，下面让我们讨论一下使用这些名词的动词。C语言通过计算表达式来完成某种动作。表达式被组织成函数的形式。

C语言的作用域规则很简单

变量的作用域是这个变量能在这个程序中被使用的范围。

如果一个变量在一个函数的外面被声明，从它声明的位置开始，直到这个文件的结束，它可以被这个范围内的任何表达式所使用。这个范围内的函数也可以使用这个变量。变量在程序开始的时候被初始化，并且在整个程序运行期间都存在。它们被称为静态变量，也许是因为在整个程序运行期间，它们位于内存中不变的位置。

如果一个变量在一个块（包括函数块）中被声明，那么这个变量在声明的时候被建立，并且在块的结尾花括号的地方被销毁。

可在6.2“持久性”查看静态变量的更多说明，它给出了如何在函数内部声明一个长期生存变量的方法。

主函数是一个特殊的函数

当一个程序运行的时候，第一件发生的事就是建立上面介绍的文件

全局变量。还没有什么计算发生，所以它们的值或者是给定的常数值，或者被初始化为零（如果声明如int gv;）。

脚本语言通常允许函数内部有一些指令，并且一些其他的指令在脚本文件的其他地方。任何需要执行的C语言表达式必须出现在函数内，并且从main函数开始执行。在上面的snprintf例子中，长度为len的数组必须在main中，因为获得len的值需要一些计算，而这些计算不能发生在程序的开始阶段。

因为main函数被操作系统调用，它的声明方式必须是操作系统能够认识的两种方式中的一种。

```
int main(void);
//which can be written as
int main();
```

或

```
int main(int, char**)
//where the two inputs are customarily named:
int main(int argc, char** argv)
```

你已经看见过第一个版本的例子了，没有什么输入，有一个整数作为输出。这个整数通常代表错误代码，如果它的值不是零，可以根据不同的语义被翻译成代表不同的错误。如果是0，代表程序执行得很顺利。返回0代表程序执行顺利是C语言中的一个传统。在main的结尾（看7.1“不需要明确地从main函数返回”），例8-6讨论了一种简单模式的例子。

C程序的大部分工作就是计算表达式

所以全局变量已经被建立起来，操作系统也为main函数准备了输入，程序开始执行main函数中的代码了。

从这里开始，就是局部变量的声明、流程控制（if-else分支语句和while循环语句）和计算表达式了。

借用前一个例子，考虑计算这个序列的时候，系统做了什么？

```
int a1, two_times;
a1 = (2+3)*7;
two_times = add_two_ints(a1, a1);
```

声明以后，a1=（2+3）*7要求计算第一个表达式（2+3），这个可以被5替换。然后计算5*7，替换为35。就像我们人类面对这一个表达式一样，但是C语言把这一个计算-替换的原则应用得更广泛。

在a1=35这一表达式中，发生了两件事情。第一件是把这个表达式替换成了35。第二件是状态改变的副作用：把a1的值改变为35。有些语言只关注于计算，但是C语言允许计算的同时去改变状态。这种情况你在上面已看见很多次了：在计算printf（"Hello\n"）;时，如果成功运行了，这一表达式的值就是0。计算出这个值的同时，屏幕上的内容被改变了。

当这些替换发生后，这一行上只剩下35。系统继续执行下一行。

函数计算的时候输入是被复制的

上面代码片段中，two_times=add_two_ints（a1,a1）首先计算a1两次，然后利用35作为输入计算add_two_ints。所以是a1的复制被送入函

数而不是a1本身。这意味着函数没有办法修改a1的值。如果在函数内部你貌似修改了输入的值，其实你修改的只是输入的一个复制。如果真的想修改输入，后面我会介绍。

表达式被分号分割

是的，C语言用分号来分割表达式。这是一种风格的选择，你也可以使用换行符、多余的空格、tab等来提高可读性。

有一些增加或者放大变量的简单方法

C有一些很好的简写方式用来执行一些修改变量的数学运算。我们可以把x=x+3简写为x+=3；或者把x=x/3写成x/=3。把变量加1是一个很常用的操作，我们有两种方法来完成。++x和x++都可以把x增加1。但是x++把没有增加前的值用于它周围的表达式，而++x把增加后的值用于它周围的表达式。

```
x++; //increment x. Evaluates to x.
++x; //increment x. Evaluates to x+1.

x--; //decrement x. Evaluates to x.
--x; //decrement x. Evaluates to x-1.

x+=3; //add three to x.
x-=7; //subtract seven from x.
x*=2; //multiply x by two.
x/=2; //divide x by two.
x%=2; //replace x with modulo
```

C有扩展的逻辑定义

我们有的时候需要知道一个表达式是真还是假，以便在if-else 结构

中决定由哪一个分支来执行。在C语言中没有true和false关键字，代替它的是，如果一个表达式是0（或者NUL字符‘\0’，或者是一个空指针），那么表达式被认为是假，任何其他的数值都被认为是真。

所有这些表达式的值都被最终转换成0或者1。

```
!x              //not x
x==y            //x equals y
x != y          //x is not equal to y
x < y           //x is less than y
x <= y          //x is less than or equal to y
x || y          //x or y
x && y          //x and y
x > y || y >= z //x is greater than y or y is greater than or equal to z
```

例如，如果x不是非0值，那么!x计算成0，!!x计算成1。

&&和||是懒惰的，它只计算能够确定整个表达式是真还是假的必要部分，而不是计算全部。例如，考虑表达式（a<0 || sqrt（a）<10），对整型或者浮点型-1求平方根是错误的。（看11.4.1“_Generic”部分讨论了C对虚数的支持。）如果a＝-1，就算我们不去计算sqrt（a）<10，仅仅靠前半部分，（a<0 || sqrt（a）<10）最后的值也一定是真。因此，表达式是不会去计算后半部分的，这样就避免了对负数求平方根的错误。

两个整形数相除，最后的结果也是一个整数

很多的开发者避免使用浮点数，它们认为整数计算得更快，并且可以避免一些舍入的错误。C语言有三个不同的操作符：实数除法，整数除法和求模操作。前两个看起来一致。

```
//tutorial/divisions.c
#include <stdio.h>
```

```
int main(){
    printf("13./5=%g\n", 13./5);
    printf("13/5=%i\n", 13/5);
    printf("13%%5=%i\n", 13%5);
}
```

下面是输出：

```
13. /5=2.6
13/5=2
13%5=3
```

表达式3.是一个浮点的实数。如果除法中的除数和被除数中有一个浮点数，那么我们就会用浮点除法，产生一个浮点数结果。如果除数和被除数都是整数，那么最后的结果就是一个整数，就像你用浮点数除法，但是结果中的浮点数的小数点部分被丢弃而最后变成一个整数。求摸运算符%给出余数。

由于浮点除法和整数除法的不同，我们在上面的例子中num/（double）den必须使用类型转换。更多的讨论可以参考7.3“减少类型转换”。

C有三元条件操作符

表达式：

```
x ? a : b
```

如果x是真，表达式为a， 否则是b。

我过去认为这样的可读性并不好，很少有脚本语言有这个操作符，

但是现在我认为它很有用。它是一个表达式，所以我们可以把它用在很多地方。

```
//tutorial/sqrt.c
#include <math.h> //The square root function is declared here.
#include <stdio.h>

int main(){
    double x = 49;
    printf("The truncated square root of x is %g.\n",
                                    x > 0 ? sqrt(x) : 0);
}
```

三元条件操作符就像&&和||一样，有短路的行为，上面的例子中，如果x<0，那么sqrt（x）不会被执行。

分支和循环语句和其他的语言没有太大的区别

C语言中的if-else语句唯一的与其他语言不同的地方在于它没有then关键字。需要计算的条件用括号括起来，条件为真时需要执行的是一个表达式或者是一个程序块。如下面的几个例子：

```
//tutorial/if_else.c
#include <stdio.h>

int main(){
    if (6 == 9)
        printf("Six is nine.\n");

    int x=3;
    if (x==1)
        printf("I found x; it is one.\n");
    else if (x==2)
        printf("x is definitely two.\n");
    else
        printf("x is neither one nor two.\n");
}
```

while循环重复执行程序块直到给定的条件为假，例如，这个程序向用户打10次招呼。

```
//tutorial/while.c
#include <stdio.h>

int main(){
    int i=0;
    while (i < 10){
        printf("Hello #%i\n", i);
        i++;
    }
}
```

如果在while语句后面括号中的控制语句在第一次判断的时候就是假，那么while循环体不会被运行。但是do-while循环至少会保证运行一次。

```
//tutorial/do_while.c
#include <stdio.h>

void loops(int max){
    int i=0;
    do {
        printf("Hello #%i\n", i);
        i++;
    } while (i < max); //Note the semicolon.
}

int main(){
    loops(3); //prints three greetings
    loops(0); //prints one greeting
}
```

for循环只是一个紧凑的while循环

while循环通常包含三个部分：

- The initializer (int i=0)。
- The test condition (i < 10)。
- The stepper (i++)。

for循环把这三个部分封装到一个地方。下面的for循环与上面的while循环等价：

```
//tutorial/for_loop.c
#include <stdio.h>

int main(){
    for (int i=0; i < 10; i++){
        printf("Hello #%i\n", i);
    }
}
```

因为这个块是一行程序，所以花括号也是可选的，我们可以这样写：

```
//<em>tutorial/for_loop2.c</em>
#include &lt;stdio.h>

<strong>int </strong>main(){
<strong>for</strong> (<strong>int</strong> i=0; i &lt; 10; i++) printf("He
llo #%i<strong>\n</strong>", i);
}
```

人们经常担心边界错误，本来我想执行10次，但是程序真正执行了9次或者11次。上面的形式（i=0开始， 验证i<10）正确地执行了10次。这也是正确地遍历一个数组的方法：

```
int len=10;
double array[len];
for (int i=0; i< len; i++) array[i] = 1./(i+1);
```

对于遍历一个序列或者在数组中的每个元素上施加某种操作，C语言不提供特殊的语法来完成这些操作（这些操作可以通过宏或者是函数来实现）。这就意味着你在源代码中会经常看见（int I = 0;i<len;i++）这种方式。

另外一方面，这种形式可以很方便地修改以适用不同的场景。如果你需要每次递增2，你就需要写成（int I = 0;i<len;i+＝2）。如果你需要一直循环直到遇到一个零元素，就写成（int I = 0;array[i]!=0;i+＝2）。你也可以把任何一个部分留空，所以如果你并不初始化一个新的变量，你可以写成下面这样for（ ;array[i]!=0;i+＝2）。

指针

指针有的时候可以被叫作别名变量，引用或者是标签（虽然C语言中有一个并不相关的东西，也叫标签。标签在C语言中很少使用，我在“标签，goto， switch 和break”一节讨论过了）。

一个指向double的指针或者是别名本身并不包含double的数据，只是指向包含double数据的某个位置。现在对同样一个事物，你有两个名字了。如果这个事物改变了，这两个版本都会看到变化。这与复制一个东西是完全不同的，复制的时候，对原始数据的改变不会影响已经复制的数据。

你可以直接要求一块内存

malloc 函数在程序中为用户分配一块内存。例如，我们可以分配一块容纳3000个整数的数组。

```
malloc(3000*sizeof(int));
```

这是本教程中第一次提到内存分配，前面提到的声明如int list[100]会自动分配内存。当声明的变量离开它的作用域的时候，自动分配的内存被自动销毁。与此相反，手工分配的内存必须用手工的方式来释放（或者等到程序结束的时候）。这种持续性有的时候是必需的。同样，自动分配的数组不能改变尺寸，手工分配的可以。两者之间的不同在6.1“自动，静态和手工内存”一节有更多的讨论。

现在我们分配了空间，如何引用它？这个时候需要用到指针了，因为我们给malloc分配的内存一个别名。

```
int *intspace = malloc(3000*sizeof(int));
```

声明中的星号（int *）代表我们声明了一个指向某个位置的指针。

内存是有限的资源，所以如果随意地使用会造成内存不足的错误，这种错误有的时候会发生。通过free函数释放内存并返回给系统。或者一直等到程序结束，操作系统会自动回收为我们的程序分配的内存。

数组就是一块内存，一块内存也可以被用作数组

在第6章，我讨论了数组和指针是不一样的，但是它们也有很多的相似点。

内存中，一个数组是某种数据类型的一片连续的区域。如果你想获得int list[100]这个数组中的第 6 个元素，系统会首先找到这个数组开始的地方，然后偏移6*sizeof（int）字节。

所以数组存取的方括号索引操作符list[6]就是一个进行地址偏移的符号，我们在使用数组的时候都要进行这种操作。如果我们有指针指向了一片连续的内存区域，通过偏移来发现正确位置这种操作完全可以用指针来完成。

下面是手工分配数组并把它输出到文件的一个例子。这个例子可以很轻松地用一个自动分配内存的数组来改写，但是出于演示目的，我们使用手工分配的内存。

```
//tutorial/manual_memory.c
#include <stdlib.h> //malloc and free
#include <stdio.h>

int main(){
    int *intspace = malloc(3000*sizeof(int));
    for (int i=0; i < 3000; i++)
        intspace[i] = i;

    FILE *cf = fopen("counter_file", "w");
    for (int i=0; i < 3000; i++)
        fprintf(cf, "%i\n", intspace[i]);
    free(intspace);
    fclose(cf);
}
```

通过malloc分配的内存可以被程序可靠地使用，但是由于没有被初始化，所以可能是一些随机的垃圾数据。我们可以用下面的语句分配内存并同时初始化为0。

```
int *intspace = calloc(3000, sizeof(int));
```

注意这个函数输入两个参数，而malloc只输入一个参数。

一个指向标量的指针就是包含一个元素的数组

假设我们有一个叫作i的指针，它指向一个单独的整数。它是长度为1的数组，如果你要求i[0]，它会找到整数的位置，并且偏移0字节，整个过程和我们使用长数组时发生的事情完全相同。

但是我们通常不把一个单个的变量当成长度为1的数组，所以对于单个变量的场景，我们用一种更简单的符号用来存储指针所指向的内容：如果用在声明语句之外，i[0]和*i是相同的。这多少有点混淆，因为当我们声明的时候，星号代表不同的含义。这么做是有其意义的（看6.3.3“错误来源于星号”），现在你只需要记住声明行中的星号代表这是一个指针，其他行中的星号代表这个指针指向的变量。

下面的代码把list数组的第一个元素设置为7，最后一行检查这个值，如果有错误发生，程序会用一个错误的信息来终止。

```
//tutorial/assert.c
#include <assert.h>

int main(){
    int list[100];
    int *list2 = list; //Declares list2 as a pointer-to-int,
                       //pointing to the same block of memory list points to
.

    *list2 = 7;        //list2 is a pointer-to-int, so *list2 is an int.

    assert(list[0] == 7);
}
```

有特殊的符号来表示用指针获得结构的成员

给出下面的声明：

```
ratio_s *pr;
```

我们知道pr是一个指向ratio_s的指针，并不是ratio_s本身。pr的尺寸只能够容纳一个单独的指针，并不需要容纳结构ratio_s的所有的内容。

我们可以通过（*pr）.numerator来获得结构内部的numerator成员，因为（*pr）是一个普通的ratio_s，点符号用来获得它的子元素。我们可以用一个尖头的符号来代替括号加星号的组合。例如：

```
ratio_s *pr = malloc(sizeof(ratio_s));
pr->numerator = 3;
```

pr->numerator 和（*pr）.numerator这两种形式是一致的，但是前一种方式可读性更好。

指针能让你修改函数的输入

现在我们知道，函数的所有输入变量都是变量的复制，而不是变量本身。当函数退出后，复制被销毁，原始的变量没有发生任何改变。

现在我们把指针输入到函数。复制的指针也指向同样的位置。下面简单的程序用这个策略来修改原始的变量。

```
//tutorial/pointer_in.c
#include <stdlib.h>
#include <stdio.h>

void double_in(int *in){
    *in *= 2;
}

int main(){
    int x[1]; // declare a one-item array, for demonstration purposes
    *x= 10;
    double_in(x);
    printf("x now points to %i.\n", *x);
}
```

Double_in函数并没有改变in，但是改变了in指针指向的变量。这样，变量x被函数double_in 扩大一倍。

这种方法用得很普遍，所以你会发现很多的函数使用指针，而不是普通的值。但是有的时候，你想让你的函数作用于普通的变量。这个时候你可以用取地址操作符（&）来获得变量的地址。如果x是一个变量，那么&x就是一个指向这个变量的指针。这可以简化我们上面的程序。

```
//tutorial/address_in.c
#include <stdlib.h>
#include <stdio.h>

void double_in(int *in){
    *in *= 2;
}

int main(){
    int x= 10;
    double_in(&x);
    printf("x is now %i.\n", x);
}
```

任何东西都有地址，所以任何东西都可以指向。

你不可以把一个函数传递给另外一个函数，你也不能用一个数组来容纳函数。但是你可以把一个指向函数的指针传递给函数，你可以有一个数组来容纳函数指针。这里我不介绍语法，6.3.5“typedef作为一种教学工具”一节有更多的讨论。

函数并不关心数据在哪里，只是处理指向数据的指针，这非常普遍。例如，建立链表的函数并不关心链表中的那些数据，只是关心它们在哪里。给出另外一个例子，我们可以传递一个函数指针，这样我们就

有一个函数，它的唯一目的就是运行另外一个函数。在这种情况下，C语言提供了一个特殊的输出类型——void指针。给出声明：

```
void *x;
```

x可以指向任何函数、结构、整数或者其他的任何东西。10.12“void指针以及它所指向的结构”更多地讨论了这种类型的指针如何用于多种应用场合。

后记

停止另一场比赛，重新开始—

——选择Bob Dylan在1965年新港民谣音乐节巡演闭幕式的歌曲

“It’s All Over Now Baby Blue（它到处都是，淡蓝色）”

你可能会大叫：等等！作者你可说过，我可以用库来使自己的工作更轻松，我在自己的领域已经是一个专家，我四处搜罗，但是我还是没有发现一个适合我用的库！

如果这个人是你，那么现在是时候揭秘我写这本书的秘密企图了：作为一个C用户，我希望有更多的人来写出优秀的库为我所用。如果你已经读完了本书，你就知道如何基于其他库来写出现代意义上的C代码，如何知道写出一套围绕几个简单的对象的函数，如何使得接口用户友好，如何撰写代码文档方便地测试，有哪些工具可用，如何用一个Git版本库，以便这样别人也能提交工作，以及如何用Autotools为一般大众打包工作。C是现代计算的基础，所以当你用C解决了某个问题的时候，这个解决方案就可以被世界各地的各种平台所利用。

Punk Rock是一种自力更生的艺术形式。一个共识是，音乐由我们自己创造，你并不需要某个公司的审查委员会的许可就能写点什么新的东西并向世界发行。事实上，我们已经具备了所有的工具，剩下的只是让梦想成真。

术语表

Alignment（对齐）

一种对数据的元素必须在一定边界处开始的要求。例如，给定一个8位的对齐要求，一个结构如果有一个1位的char变量，后面跟着一个8位的int，则需要在char之后有7位的填充，这样int就是从一个8位的边界上开始了。

ASCII

美国信息交换标准代码（American Standard Code for Information Interchange）。一个把英语本身的字母对应0～127的数字的映射。提示：在很多系统中，man ascii将打印出代码表。

Automatic allocation（自动分配）

对于一个自动分配变量，它在内存中的空间是在这个变量被声明的时候由系统分配的，并且在它的范围之外被移除。

Autotools

一系列来自GNU的、用于简化在任何系统中的编译过程的程序，包括Autoconf、Automake和Libtool。

Benford’s law

在一个大数据集的首数字倾向于一个类log的分布：1出现的概率为30%，2大约是17.5%……，9大约是4.5%。

Boolean（布尔值）

即真/假。以Geoge Boole—一位生活在19世纪早期到中期的英国数学家的名字命名。

BSD

伯克利软件发行版（Berkeley Software Distribution）。一个POSIX实现。

callback function（回调函数）

一个函数（A）被作为输入发送给另一个函数（B），这样函数B可以通过它自己的操作来调用函数A。例如，泛化的排序函数通常是用一个输入函数来比较2个元素。

call graph

一个用来显示函数调用与被调用关系的方块-箭头图。

Cetology

对鲸类（海洋哺乳类）的研究学科。

compiler（编译器）

正式地说，就是把一个程序从（人类可阅读的）文本转化为（人类难以辨认的）机器指令的程序。一般被用来指代预处理器+编译器+连接器。

debugger（调试器）

一个需要和编译好的程序互动地执行的程序，使得用户可以暂停程序，检查和修改变量值，等。一般对理解bug有用。

deep copy（深度复制）

一个含有指针的结构的复制，其跟踪所有的指针并对这些指针所指向的数据做出复制。

encoding（编码）

将人类语言字符转换为计算机可处理的数字代码的方法。参见ASCII、 multibyte encoding和wide- character encoding。

environment variable

一个代表某个程序的环境的变量，由其父程序（一般是shell）来设定。

external pointer

参见Opaque Pointer。

floating point（浮点数）

一种类似于科学记数法的表述数字的方法，类似2.3×10^4，具有一个指数部分（本例中是4）和一个尾数部分（本例中是2.3）。在写下尾数之后，把指数部分想象为允许小数点漂浮到正确的位置上。

frame（帧）

堆栈中的空间，其中存放函数的信息（例如输入参数和自动变

量）。

gdb

GNU调试器。

global（全局）

当一个变量的作用域是整个程序，它就是全局的。C实际上并不存在全局作用域，但是如果一个变量处于可以被这个程序的所有代码文件所包含的头文件中，它实际上就相当于一个全局变量。

glyph

一种用于书面交流的符号。

GNU

代表Gnu’s Not UNIX。

GSL

代表GNU科学计算库。

heap（堆）

用于手动分配内存的内存空间。可与stack（堆栈）对比。

IDE

即集成开发环境（Integrated Development Environment）。通常是基于一个图形界面的文本编辑器，并带有编译、调试和其他服务于程序员的特性的工具。

integration test（集成测试）

一项运行一系列的步骤以测试一个代码库中的多个部分的测试（每个部分应该有自己的单元测试）。

library（库）

基本上，一个没有main函数的程序，因此它是为其他程序所准备的一系列的函数、typedef和变量。

linker（连接器）

一个程序，用于把一个程序的各分散的部分（比如独立的目标文件

和库）连接起来，并调整对外部目标文件函数和变量的引用。

Linux

技术上讲，是一种操作系统内核，但是一般用来指代一个与完整的BSD/ GNU/ Internet Systens Consortium/ Linux/ Xorg/等捆绑的所形成的统一的包。

macro（宏）

一个（通常是）较短的文字段落，用于替代（通常是）较长的文字。

manual allocation（手工分配）

应程序的请求，用malloc或者calloc在heap（堆）中分配一个变量，并应用户的要求用free释放。

multibyte encoding

一种用可变数量的字符来代表单字符人类语言的编码方法。可与wide-character encoding对比。

mutex（互斥锁）

mutual exclusion的缩写，一个用来确保在同一时刻只有一个线程可以使用资源的数据结构。

NaN

即Not-a-Number。IEEE 754（浮点数）标准定义为数学上不可能的计算的输出，比如0/0或者log(-1)。经常被用于作为缺少数据或坏数据的标志。

object（对象）

一个关联了对其操作的函数的数据结构。理论上讲，对象封装了一个概念，为其他代码与该对象互动提供了有限的入口点。

object file（目标文件）

一个包含机器可阅读指令的文件。通常是对源代码文件运行编译器的结果。

opaque pointer（不透明指针）

一个指向数据的指针，处理该指针的函数不能读到指针本身，但是被传递到其他的函数中后可以读到数据。一个脚本语言的函数可以调用一个返回指向C侧数据的不透明指针的C函数，在脚本语言中后来的函数可以用那个指针来操作C侧的数据。

POSIX

指代The Portable Operating System Interface。一个类UNIX操作系统遵从的IEEE标准，描述了一系列的C函数、shell和一些基本工具。

preprocessor（预处理器）

概念上说，一个程序在编译器紧前运行的程序，执行类似#include和#define这样的指令。在实践中，通常是编译器的一部分。

process（过程）

一个运行中的程序。

profiler（优化器）

一个用来报告你程序中的运行时间花费在何处的程序，这样你在提速优化的时候就知道应该聚焦在哪里。

pthread

POSIX线程。一个用在POSIX标准中定义的C语言线程接口产生的线程。

RNG

随机数产生器，其中“随机”基本上意味着你可以合理地期望一个序列的随机数不与任何另外的序列有系统性的关联。

RTFM

即“请读手册（Read the manual）”的缩写。

Sapir-Whorf假说

关于我们所说的语言决定着我们具有的思维能力的假说。简单地说，我们经常用词汇思考，这是显然的；复杂地说，我们不能想象也无法由语言描述的事物，这显然不对。

scope（作用域）

一个变量被声明和可使用的代码范围。好的代码风格都会尽量缩小

变量的作用域。

script（脚本）

一个用可解释的语言，比如shell，编写的程序。

segfault

段错误（segmentation fault）。

segmentation fault

你正在接触为你的程序分配的内存之外的内存。

SHA

安全哈希算法（Secure Hash Algorithm，SHA）。

shell

一个支持用户与操作系统互动的程序，可以是命令行或者脚本的形式。

SQL

结构化查询语言（Structured Query Language）。一种与数据库交互的标准方法。

stack（堆栈）

函数执行发生的内存空间。特别是，自动变量被放在这里。每个函数都得到一个帧，而且每次一个子函数被调用，它的帧在概念上讲是被堆放在调用它的函数的帧的上方。

static allocation（静态分配）

使得变量或者文件范围的方法，其中变量在函数中由static关键词分配。分配发生在程序执行之前，而且直到程序退出，变量一直存在。

test harness（测试工具）

用来运行一系列的单元测试和整合测试的环境。提供简单的辅助结构的配置/释放，允许检查可能（正确地）引发主程序的崩溃。

thread（线程）

计算机独立于其他线程运行的一系列的指令。

token（标识）

被当作语法单元的一组字符，比如一个变量名、一个关键字，或者一个类似*或+的操作符。解析文本的第一步就是把它打散为token；strtok_r和strtok_n就是为此设计的。

type punning（类型转换）

把一个变量从一种类型转化为另一种类型，从此强迫编译器将这个变量按照第二种类型处理。例如，假设struct {int a; char *b;} astruct，那么(int) astruct就是一个整数（但是为了安全地转换，参见11.1.2“C，更少的缝隙”）。经常是无法移植的；通常是不好的形式。

type qualifier（类型修饰符）

一个对于编译器应该如何处理一个变量的描述符。与变量的类型（int、float等）无关。C中唯一的类型修饰符是const、restrict、volatile和_Atomic。

union（联合）

可以被理解为多种类型的一块内存。

unit test（单元测试）

一段用来测试代码库中某段代码的代码。对比参见integration test。

UI

即用户界面。对于C语言的库来说，需要提供用户使用这个库时与之兼容的类型定义、宏定义和函数声明。

UTF

即Unicode Transformation Format。

variadic function

输入参数的数量可变的函数（例如，printf）。

wide-character encoding

一种文本编码方式，其中每个人类语言字符都由一个固定长度的char类型表示。例如，UTF-32就明确每个Unicode均用4字节表示。对每

个人类语言字符用多个字节表示，但是这个定义又与multibyte encoding不同。

XML

即扩展标记语言（Extensible Makeup Language）。

作者简介

自从于加州理工学院获得社会科学博士后，Ben Klemens就一直从事统计分析和人口的计算机辅助建模工作。他的观点是，写代码一定应该是趣味横生的，并先后非常愉快地为布鲁金斯学会、世界银行、美国国家精神健康中心等机构写过分析和建模代码（主要是C代码）。他作为布鲁金斯学会的非常驻研究员，与自由软件基金会一道，做了很多工作来确保有创意的程序员拥有保留其作品使用权的权利。他目前为美国联邦政府工作。

封面介绍

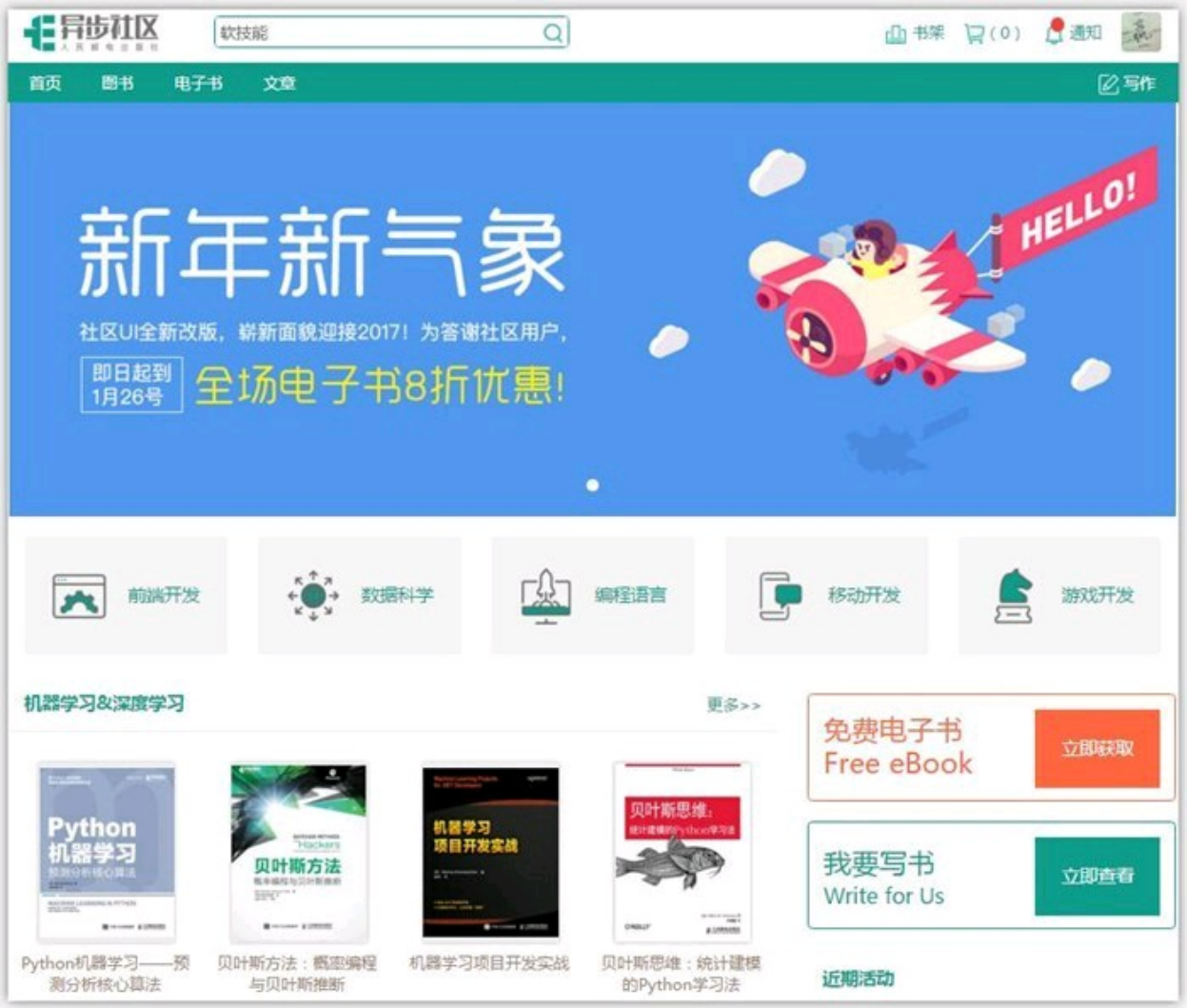
本书封面上的动物是斑袋貂（Spilocuscus maculatus），一种生活在澳大利亚、新几内亚和附近小岛上热带雨林和红树林中的有袋目哺乳动物。它的头部呈圆形，耳朵小而不明显，皮毛厚实，有一条有助于攀援的可卷曲的尾巴。卷曲的尾巴是斑袋貂的典型特征，尾巴上部贴近身体的部分覆盖着绒毛，而下半部分的内侧覆盖着粗糙的鳞甲，适合缠住树枝。它的眼睛可以看到黄色、橙色及红色，视野像蛇一样狭窄。 斑袋貂通常非常胆小，所以很少被人发现。它是夜间活动的，在夜间捕猎和进食，白天在树枝上自己搭建的巢穴中睡眠。它行动缓慢，有时很懒——因此经常被误认为是树懒、负鼠，甚至是猴子。 典型的斑袋貂是独居动物，独自筑巢和进食，与其他个体特别是具有竞争性的雄性之间的互动常是有侵略性和冲突性的。雄性斑袋貂以气味标识它们的领地以警告其他雄性，从它们的身体和气味排泄腺散发出具有穿透性的麝香气味。它们在树枝上散布唾液以警示出现在它们领地上的其他个体。如果它们在自己的领地上遇到其他的雄性个体，它们会发出吠叫、咆哮和嘶叫的声音，并且直立起来保护它们的领地。 斑袋貂具有非特异性的牙齿排列方式，使得它们可以食用多种植物。已知它们可以吃花朵、小动物，偶尔也吃蛋。斑袋貂的天敌包括蟒蛇和一些掠食性鸟类。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。 使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一

次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区

微信订阅号

微信服务号

官方微博

QQ群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn