C和C++实务精选

C专家编程

Peter Van Der Linden 著
徐波 译

人民邮电出版社

图书在版编目（CIP）数据

C 专家编程/（美）林登（Linden，R.）著；徐波译. —北京：人民邮电出版社，2002.12
ISBN 7-115-10627-4

I. C… II. ①林…②徐… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2002）第 088792 号

版 权 声 明

C 和 C++实务精选

C 专家编程

◆ 著 Peter Van Der Linden
译 徐 波
责任编辑 陈冀康

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www. ptpress.com.cn
读者热线 010-67132705
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000 1/16
印张：19. 5
字数：430 千字 2002 年 12 月第 1 版
印数：1-4 000 册 2002 年 12 月北京第 1 次印刷

著作权合同登记 图字：01 - 2002 - 2765 号
ISBN 7-115-10627-4/TP • 3084

定价：40.00 元

内容提要

《C 专家编程》展示了最优秀的 C 程序员所使用的编码技巧，并专门开辟了一章对 C++的基础知识进行了介绍。

书中 C 的历史、语言特性、声明、数组、指针、链接、运行时、内存以及如何进一步学习 C++等问题进行了细致的讲解和深入的分析。全书撷取几十个实例进行讲解，对 C 程序员具有非常高的实用价值。

本书可以帮助有一定经验的 C 程序员成为 C 编程方面的专家，对于具备相当的 C 语言基础的程序员，本书可以帮助他们站在 C 的高度了解和学习 C++。

序

最近，我逛了一家书店，当我看到大量枯燥乏味的 C 和 C++书籍时，心情格外沮丧。我发现极少有作者想向读者传达这样一个信念：任何人都可以享受编程。在冗长而乏味的阅读过程中，所有的奇妙和乐趣都烟消云散了。如果你硬着头皮把它啃完，或许会有长进。但编程本来不该是这个样子的呀！

编程应该是一项精妙绝伦、充满生机、富有挑战的活动，而讲述编程的书籍也应时时迸射出激情的火花。本书也是一本教学性质的书籍，但它希望重新把快乐融入编程之中。如果本书不合你的口味，请把它放回到书架上，但务必放到更显眼的位置上，这里先谢过了。

好，听了这个开场白，你不免有所疑问：关于 C 语言编程的书可以说是不胜枚举，那么这本书又有什么独到之处呢？

《C 专家编程》应该是每位程序员的第二本学习 C 语言的书。这里所提到的绝大多数教程、提示和技巧都是无法在其他书上找到的，即使有的话，它们通常也是作为心得体会手工记录在手册的书页空白处或旧打印纸的背面。作者以及 Sun 公司编译器和操作系统小组的同事们在多年 C 语言编程实践中，积累了大量的知识和经验。书中讲述了许多有趣的 C 语言故事和轶闻，诸如连接到因特网上的自动售货机、太空软件中存在的问题，以及一个 C 语言的缺陷怎样使整个 AT&T 长途电话网络瘫痪等。本书的最后一章是 C++语言的轻松教程，帮助你精通这门日益流行的从 C 语言演化而来的语言。

本书讲述的是应用于 PC 和 UNIX 系统上的 ANSI 标准 C 语言。对 C 语言中与 UNIX 平台复杂的硬件结构（如虚拟内存等）相关的特性作了详细描述。对于 PC 的内存模型和 Intel 8086 系列对 C 语言产生影响的部分也作了全面介绍。C 语言基础相当扎实的人很快就会发现书中充满了很多程序员可能需要多年实践才能领会的技巧、提示和捷径。它覆盖了许多令 C 程序员困惑的主题：

- typedef struct bar{ int bar; }bar 的真正意思是什么？
- 我怎样把一些大小不同的多维数组传递到同一个函数中？
- 为什么 extern char *p;同另一个文件的 char p[100];不能够匹配？
- 什么是总线错误（bus error）？什么是段违规 (segmentation violation)？
- char *foo[]和 char(*foo)[]有何不同？

如果你对这些问题不是很有把握，很想知道 C 语言专家是如何处理它们的，那么请继续

阅读！即使你对这些问题已经了如指掌，对 C 语言的其他细节也是耳熟能详，那么也请阅读本书，继续充实你的知识。如果觉得不好意思，就告诉书店职员“我是为朋友买书。”

Peter Van Der Linden 于加州硅谷

前言

C 代码。C 代码运行。运行码运行...请!

——*Barbara Ling*

所有的 C 程序都做同一件事，观察一个字符，然后啥也不干。

——*Peter Weinberger*

你是否注意到市面上存有大量的 C 语言编程书籍，它们的书名具有一定的启示性，如：*C Traps and Pitfalls*（本书中文版《C 陷阱与缺陷》已由人民邮电出版社出版），*The C Puzzle Book*, *Obfuscated C and Other Mysteries*，而其他的编程语言好像没有这类书。这里有一个很充分的理由!

C 语言编程是一项技艺，需要多年历练才能达到较为完善的境界。一个头脑敏捷的人很快就能学会 C 语言中基础的东西。但要品味出 C 语言的细微之处，并通过大量编写各种不同程序成为 C 语言专家，则耗时甚巨。打个比方说，这是在巴黎点一杯咖啡与在地铁里告诉土生土长的巴黎人该在哪里下车之间的差别。本书是一本关于 ANSI C 编程语言的高级读本。它适用于已经编写过 C 程序的人，以及那些想迅速获取一些专家观点和技巧的人。

编程专家在多年的实践中建立了自己的技术工具箱，里面是形形色色的习惯用法、代码片段和灵活掌握的技巧。他们站在其他更有经验的同事的肩膀上，或是直接领悟他们的代码，或是在维护其他人的代码时聆听他们的教诲，随着时间的推移，逐步形成了这些东西。另外一种成为 C 编程高手的途径是自省，在认识错误的过程中进步。几乎每个 C 语言编程新手都曾犯过下面这样的书写错误：

```
if(i = 3
```

正确的应该是：

```
if(i == 3
```

一旦有过这样的经历，这种痛苦的错误（需要进行比较时误用了赋值符号）一般不会再犯。有些程序员甚至养成了一种习惯，在比较式中先写常数，如：if(3 == i)。这样，如果不小心误用了赋值符号，编译器就会发出“attempted assighnment to literal(试图向常数赋值)”的错误信息。虽然当你比较两个变量时，这种技巧起不了作用。但是，积少成多，如果你一直留心这些小技巧，迟早会对你有所帮助的。

价值 2000 万美元的 Bug

1993 年春天，在 SunSoft 的操作系统开发小组里，我们接到了一个“一级优先”的 Bug 报告，是一个关于异步 I/O 库的问题。如果这个 Bug 不解决，将会使一桩价值 2000 万美元的硬件产品生意告吹，因为对方需要使用这个库的功能。所以，我们顶着重压寻找这个 Bug。经过几次紧张的调试，问题被圈定在下面这条语句上：

```
x == 2;
```

这是个打字错误，它的原意是一条赋值语句。程序员的手指放在“=”键上，不小心多按了一下。这条语句成了将 x 与 2 进行比较，比较结果是 true 或者 false，然后丢弃这个比较结果。

C 语言的表达能力也实在是强，编译器对于“求一个表达式的值，但不使用该值”这样的语句竟然也能接受，并且不发出任何警告，只是简单地把返回结果丢弃。我们不知道是应该为及时找到这个问题的好运气而庆幸，还是应该为这样一个常见的打字错误可能付出高昂的代价而痛心疾首。有些版本的长整数程序已经能够检测到这类问题，但人们很容易忽视这些有用的工具。

本书收集了其他许多有益的故事。它记录了许多经验丰富的程序员的智慧，避免读者再走弯路。当你来到一个看上去很熟的地方，却发现许多角落依然陌生，本书就像是一个细心的向导，帮助你探索这些角落。本书对一些主要话题如声明、数组/指针等作了深入的讨论，同时提供了许多提示和记忆方法。本书从头到尾都采用了 ANSI C 的术语，在必要时我会用日常用语来诠释。

编程挑战

小 启 发

样例框

我们设置了“编程挑战”这个小栏目，像这样以框的形式出现。

框中会列出一些对你所编写的程序的建议。

另外，我们还设置了“小启发”这个栏目，也是以框的形式出现的。

“小启发”里出现的是在实际工作中所产生一些想法、经验和指导方针。你可以在编程中应用它们。当然，如果你觉得你已经有了更好的指导原则，也完全可以不理会它们。

约定

我们所采用的一个约定是用蔬菜和水果的名字来代表变量的名字（当然只适用于小型程序片段，现实中的程序不可如此）：

```
char pear[40];
double peach;
int mango = 13;
long melon = 2001;
```

这样就很容易区分哪些是关键字，哪些是程序员所提供的变量名。有些人或许会说，你不能拿苹果和桔子作比较。但为什么不行呢？它们都是在树上生长、拳头大小、圆圆的可食之物。一旦你习惯了这种用法，你就会发现它很有用。另外还有一个约定，有时我们会重复某个要点，以示强调。

和精美食谱一样，《C 专家编程》准备了许多可口的东西，以实例的样式奉献给读者。每一章都被分成几个彼此相关而又独立的小节。无论是从头到尾认真阅读，还是随意翻开一章选一个单独的主题细细品味，都是相当容易的。许多技术细节都蕴藏于 C 语言在实际编程中的一些真实故事里。幽默对于学习新东西是相当重要的，所以我在每一章都以一个“轻松一下”的小栏目结尾。这个栏目包含了一个有趣的 C 语言故事，或是一段软件轶闻，让读者在学习的过程中轻松一下。

读者可以把本书当作 C 语言编程的思路集锦，或是 C 语言提示和习惯用法的集合，也可以从经验丰富的编译器作者那里汲取营养，更轻松地学习 ANSI C。总之，它把所有的信息、提示和指导方针都放在一个地方，让你慢慢品味。所以，请赶紧翻开书，拿出笔，舒舒服服在坐在电脑前，开始快乐之旅吧！

轻松一下——优化文件系统

偶尔，在 C 和 UNIX 中，有些方面是令人感觉相当轻松的。只要出发点合理，什么样的奇思妙想都不为过。IBM/Motorola/Apple PowerPC 架构具有一种 E.I.E.I.O 指令[1]，代表“Enforce

[1] 可能是由一个名叫 McDonald 的老农设计的。

In-Order Execution of I/O”（在 I/O 中实行按顺序执行的方针）。与这种思想相类似，在 UNIX 中也有一条称作 tunefs 的命令，高级系统管理员用它修改文件系统的动态参数，并优化磁盘中文件块的布局。

和其他的 Berkeley[1]命令一样，在早期的 tunefs 在线手册上，也是以一个标题为“Bugs”的小节来结尾。内容如下：

Bugs:

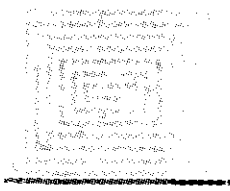
这个程序本来应该在安装好的（mounted）和活动的文件系统上运行，但事实上并非如此。因为超级块（superblock）并不是保持在高速缓冲区中，所以该程序只有当它运行在未安装好的（dismounted）文件系统中时才有效。如果运行于根文件系统，系统必须重新启动。

你可以优化一个文件系统，但不能优化一条鱼。

更有甚者，在文字处理器的源文件中有一条关于它的注释，警告任何人不得忽视上面这段话！内容如下：

如果忽视这段话，你就等着烦吧。一个 UNIX 里的怪物会不断地纠缠你，直到你受不了为止。

当 SUN 和其他一些公司转到 SVr4 UNIX 平台时，我们就看不到这条警训了。在 SVr4 的手册中没有了“Bugs”这一节，而是改名为“注意”（会不会误导大家？）。“优化一条鱼”这样的妙语也不见了。作出这个修改的人现在一定在受 UNIX 里面怪物的纠缠，自作自受！



编程挑战

计算机日期

关于 time_t，什么时候它会到达尽头，重新回到开始呢？

写一个程序，找出答案。

1. 查看一下 time_t 的定义，它位于文件/user/include/time.h 中。

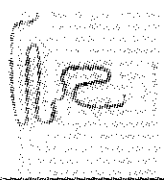
2. 编写代码，在一个类型为 time_t 的变量中存放 time_t 的最大值，然后把它传递给 ctime()函数，转换成 ASCII 字符串并打印出来。注意 ctime()函数同 C 语言并没有任何关系，它只表示“转换时间”。

如果程序设计者去掉了程序的注释，那么多少年以后，他不得不担心该程序会在 UNIX 平台上溢出。请修改程序，找出答案。

[1] 加州大学伯克利分校，UNIX 系统的许多版本都是在那里设计的。 ——译者注

1. 调用 time()获得当前的时间。
2. 调用 difftime()获得当前时间和 time_t 所能表示的最大时间值之间的差值(以秒计算)。
3. 把这个值格式化为年、月、周、日、小时、分钟的形式，并打印出来。

它是不是比一般人的寿命还要长?



解决方案

计算机日期

这个练习的结果在不同的 PC 和 UNIX 系统上有所差异，而且它依赖于 time_t 的存储形式。在 Sun 的系统中，time_t 是 long 的 typedef 形式。我们所尝试的第一个解决方案如下:

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

int main() {
  time_t biggest= 0x7FFFFFFF;

  printf("biggest = %s \n", ctime(&biggest));
  return 0;
}
```

这是一个输出结果:

```
  biggest = Mon Jan 18 19:14:07 2038
```

显然，这不是正确的结果。ctime()函数把参数转换为当地时间，它跟世界统一时间 UTC（格林尼治时间）并不一致，取决于你所在的时区。本书写作地是加利福尼亚，比伦敦晚 8 个小时，而且现在的年份跟最大时间值的年份相差甚远。

事实上，我们应该采用 gmtime()函数来取得最大的 UTC 时间值。这个函数并不返回一个可打印的字符串，所以不得不用 asctime()函数来获取一个这样的字符串。权衡各方面情况后，修订过的程序如下:

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

int main() {
  time_t  biggest = 0x7FFFFFFF;
  printf("biggest = %s \n", asctime(gmtime(&biggest)));
  return 0;
}
```

它给出了如下的结果:

```
biggest = Tue Jan 19 03:14:07 2038
```

看！这样就挤出了 8 个小时。

但是，我们并未大功告成。如果你采用的是新西兰的时区，你就会又多出 13 个小时，前提是它在 2038 年仍然采用夏令时。他们在 1 月份时采用的是夏令时，因为新西兰位于南半球。但是，由于新西兰的最东端位于日界线的东面，在那里它应该比格林尼治时间晚 10 小时而不是早 14 小时。这样，新西兰由于其独特的地理位置，不幸成为该程序的第一个 Bug 的受害者。

即使像这样简单的问题也可能在软件中潜伏令人吃惊的隐患。如果有人觉得对日期进行编程是小菜一碟，一次动手便可轻松搞定，那么他肯定没有深入研究问题，程序的质量也可想而知。

目录

第1章

C：穿越时空的迷雾

C 诡异离奇，缺陷重重，却获得了巨大的成功。

——*Dennis Ritchie*

1.1 C 语言的史前阶段

听上去有些荒谬，C 语言的产生竟然源于一个失败的项目。1969 年，通用电气、麻省理工学院和贝尔实验室联合创立了一个庞大的项目——Multics 工程。该项目的目的是创建一个操作系统，但显然遇到了麻烦：它不但无法交付原先所承诺的快速而便捷的在线系统，甚至连一点有用的东西都没有弄出来。虽然开发小组最终勉强让 Multics 开动起来，但他们还是陷入了泥淖，就像 IBM 在 OS/360 上面一样。他们试图建立一个非常巨大的操作系统，能够应用于规模很小的硬件系统中。Multics 成了总结工程教训的宝库，但它同时也为 C 语言体现“小即是美”铺平了道路。

当心灰意冷的贝尔实验室的专家们撤离 Multics 工程后，他们又去寻找其他任务。其中一位名叫 Ken Thompson 的研究人员对另一个操作系统很感兴趣，他为此好几次向贝尔管理层提议，但均遭否决。在等待官方批准时，Thompson 和他的同事 Dennis Ritchie 自娱自乐，把 Thompson 的“太空旅行”软件移植到不太常用的 PDP-7 系统上。太空旅行软件模拟太阳系的主要星体，把它们显示在图形屏幕上，并创建了一架航天飞机，它能够飞行并降落到各个行星上。与此同时，Thompson 加紧工作，为 PDP-7 编写了一个简易的新型操作系统。它比 Multics 简单得多，也轻便得多。整个系统都是用汇编语言编写的。Brian Kernighan 在 1970 年给它取名为 UNIX，自嘲地总结了从 Multics 中获得的那些不应该做的教训。图 1-1 描述了早期 C、UNIX 和相关硬件系统的关系。

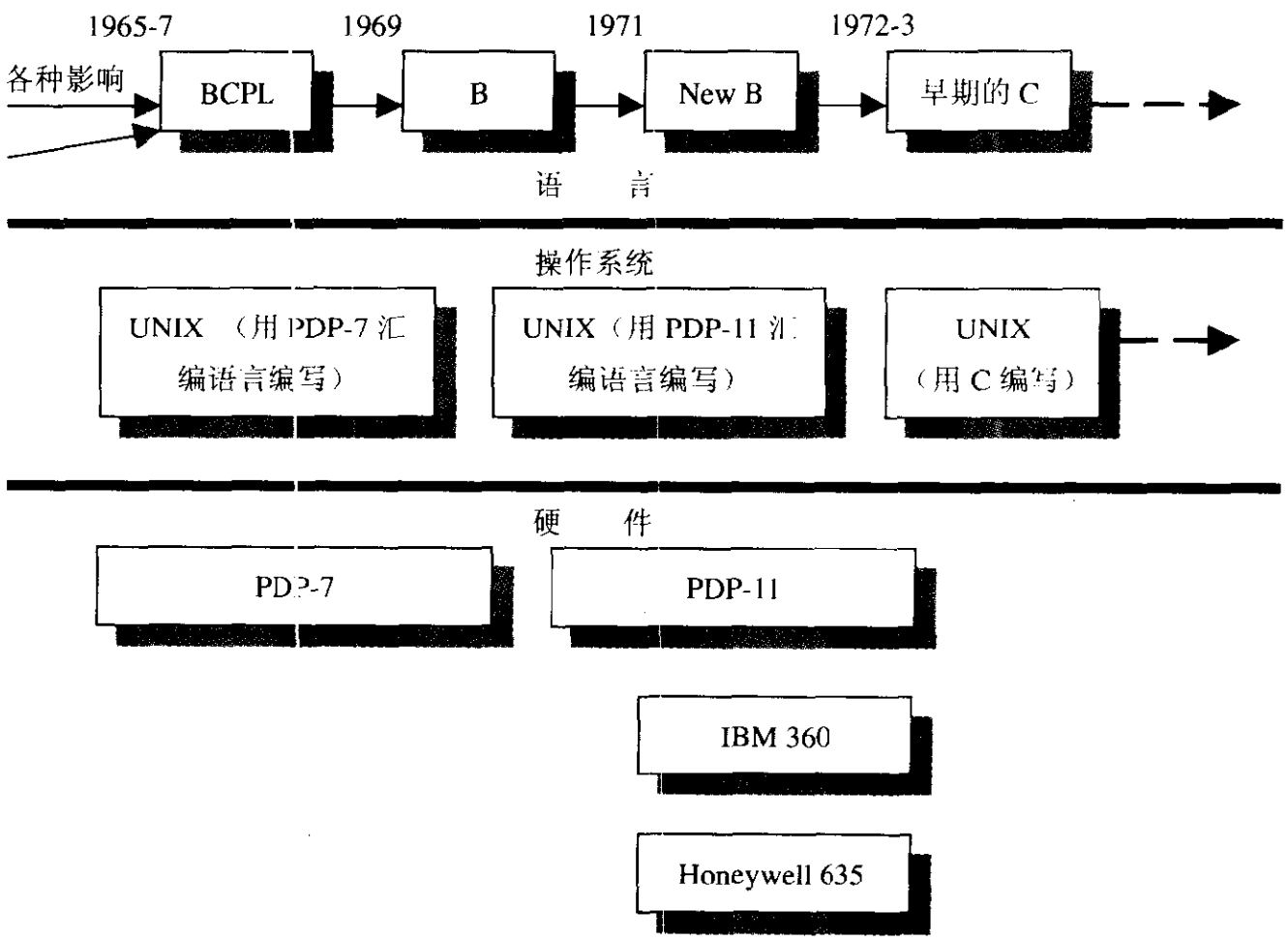


图 1-1　早期 C、UNIX 和相关的硬件系统

是先有 C 语言还是先有 UNIX 呢？说起这个问题，人们很容易陷入先有鸡还是先有蛋的套套中。确切地说，UNIX 比 C 语言出现得早（这也是为什么 UNIX 的系统时间是从 1970 年 1 月 1 日起按秒计算的，它就是那时候产生的啊）。然而，我们这里讨论的不是家禽趣闻，而是编程故事。用汇编语言编写 UNIX 显得很笨拙，在编制数据结构时浪费了大量的时间，而且系统难以调试，理解起来也很困难。Thompson 想利用高级语言的一些优点，但又不想像 PL/I[1]那样效率低下，也不想碰见在 Multics 中曾遇到过的复杂问题。在用 Fortran 进行了一番简短而又不成功的尝试之后，Thompson 创建了 B 语言，他把用于研究的语言 BCPL[2]作了简化，使 B 的解释器能常驻于 PDP-7 只有 8KB 大小的内存中。B 语言从来不曾真正成功过，因为硬件系统的内存限制，它只允许放置解释器，而不是编译器，由此产生的低效阻碍了使用 B 语言进行 UNIX 自身的系统编程。

---

[1] 学习、使用和实现 PL/I 的困难使一位程序员写了这样一首打油诗：“IBM 有个 PL/I，语法比 JOSS 还糟糕，到处都见它踪影，实实在在是垃圾。JOSS 是个老古董，它可不是因简单而闻名。”

[2] “BCPL：A Tool for Compiler Writing and System Programming（BCPL，编译器编写和系统编程的工具），” Martin Richards, Proc. *AFIPS Spring Joint Computer Conference*, 34(1969), pp.557-566。BCPL 并非“Before C Programming Language（C 前身编程语言）”的首字母缩写，尽管这是个有趣的巧合。它的确切意思是“Basic Combined Programming Language（基本组合编程语言）”。basic 的意思是“不花哨”，它是由英国伦敦大学和剑桥大学的研究人员合作开发的。Multics 实现了一种 BCPL 编译器。

软件信条

编译器设计者的金科玉律：效率（几乎）就是一切

在编译器中，效率几乎就是一切。当然还有一些其他需要关心的东西，如有意义的错误信息、良好的文档和产品支持。但与用户需要的速度相比，这些因素就黯然失色了。编译器的效率包括两个方面：运行效率（代码的运行速度）和编译效率（产生可执行代码的速度）。除了一些开发和学习环境之外，运行效率起决定性作用。

有很多编译优化措施会延长编译时间，但却能缩短运行时间。还有一些优化措施（如清除无用代码和忽略运行时检查等）即能缩短编译时间，又能减少运行时间，同时还能减少内存的使用量。这些优化措施的不利之处在于可能无法发现程序中无效的运行结果。优化措施本身在转换代码时是非常谨慎的，但如果程序员编写了无效的代码（如：越过数组边界引用对象，因为他们“知道”附近有他们需要的变量）就可能引发错误的结果。

这就是为什么说效率几乎就是一切但也并不是绝对的道理。如果得到的结果是不正确的，那么效率再高又有什么意义呢？编译器设计者通常会提供一些编译器选项。这样，每个程序员可以选择自己想要的优化措施。B 语言不算成功，而 Dennis Ritchie 所创造的注重效率的“New B”却获得了成功，充分证明了编译器设计者的这条金科玉律。

B 语言通过省略一些特性（如嵌套过程和一些循环结构），对 BCPL 语言作了简化，并发扬了“引用数组元素相当于对指针加上偏移量的引用”这个想法。B 语言同时保持了 BCPL 语言无类型这个特点，它仅有的操作数就是机器的字。Thomposon 发明了++和--操作符，并把它加入到 PDP-7 的 B 编译器中。它们在 C 语言中依然存在，很多人天真地以为这是由于 PDP-11 存在对应的自动增/减地址模型，这种想法是错误的！自动增/减机制的出现早于 PDP-11 硬件系统的出现。尽管在 C 语言中，拷贝字符串中的一个字符的语句：

```
*p++ = *s++;
```

可以极其有效地被编译为 PDP-11 代码：

```
moveb (r0)+, (r1)+
```

这使得许多人错误地以为前者的语句形式是根据后者特意设计的。

当 1970 年开发平台转移到 PDP-11 以后，无类型语言很快就显得不合时宜了。这种处理器以硬件支持几种不同长度的数据类型为特色，而 B 语言无法表达不同的数据类型。效率也是一个问题，这也迫使 Thompson 在 PDP-11 上重新用汇编语言实现了 UNIX。Dennis Ritchie 利用 PDP-11 的强大性能，创立了能够同时解决多种数据类型和效率的“New B”（这个名字很快变成了“C”）语言，它采用了编译模式而不是解释模式，并引入了类型系统，每个变量

在使用前必须先声明。

1.2 C 语言的早期体验

增加类型系统的主要目的是帮助编译器设计者区分新型 PDP-11 机器所拥有的不同数据类型，如单精度浮点数、双精度浮点数和字符等。这与其他一些语言如 Pascal 形成了鲜明的对比。在 Pascal 中，类型系统的目的是保护程序员，防止他们在数据上进行无效的操作。由于设计哲学不同，C 语言排斥强类型，它允许程序员需要时可以在不同类型的对象间赋值。类型系统的加入可以说是事后诸葛，从未在可用性方面进行过认真的评估和严格的测试。时至今日，许多 C 程序员仍然认为“强类型”只不过是增加了敲击键盘的无用功。

除了类型系统之外，C 语言的许多其他特性是为了方便编译器设计者而建立的（为什么不呢？开始几年 C 语言的主要客户就是那些编译器设计者啊）。根据编译器设计者的思路而发展形成的语言特性有：

- **数组下标从 0 而不是 1 开始**。绝大多数人习惯从 1 而不是 0 开始计数。编译器设计者则选择从 0 开始，因为偏移量的概念在他们心中已是根深蒂固。但这种设计让一般人感觉很别扭。尽管我们定义了一个数组 a[100]，你可千万别往 a[100]里存储数据，因为这个数组的合法范围是从 a[0]到 a[99]。
- **C 语言的基本数据类型直接与底层硬件相对应**。例如，不像 Fortran，C 语言中不存在内置的复数类型。某种语言要素如果底层硬件没有提供直接的支持，那么编译器设计者就不会在它上面浪费任何精力。C 语言一开始并不支持浮点类型，直到硬件系统能够直接支持浮点数之后才增加了对它的支持。
- **auto 关键字显然是摆设**。这个关键字只对创建符号表入口的编译器设计者有意义。它是意思是“在进入程序块时自动进行内存分配”（与全局静态分配或在堆上动态分配相反）。其他程序员不必操心 auto 这个关键字，它是缺省的变量内存分配模式。
- **表达式中的数组名可以看作是指针**。把数组当作指针，简化了很多东西。我们不再需要一种复杂的机制区分它们，把它们传递到一个函数时不必忍受必须复制所有数组内容的低效率。不过，数组和指针并不是在任何情况下都是等效的，更详细的讨论参见第 4 章。
- **float 被自动扩展为 double**。尽管在 ANSI C 中情况不再如此，但最初浮点数常量的精度都是 double 型的，所有表达式中 float 变量总被自动转换成 double。这样做的理由从未公诸于众，但它与 PDP-11 中浮点数的硬件表示方式有关。首先，在 PDP-11 或 VAX 中，从 float 转换到 double 代价非常小，只要在后面增加一个每个位均为 0 的字即可。如果要转换回来，去掉第二个字就可以了。其次，要知道在某些 PDP-11 的浮点数硬件表示形式中有一个运算模式位(mode bit)，你可以只进行 float 的运算，也可以只进行 double 的运算，但如果想在这两种方式间进行切换，就必须修改这个位来改变运算模式。在早期的 UNIX 程序中，float 用得不是太多，所以把运算模式固定为 double 是比较方便的，省得编译器设计者去跟踪它的变化。

• **不允许嵌套函数（函数内部包含另一个函数的定义）**。这简化了编译器，并稍微提高了 C 程序的运行时组织结构。具体的机理在第 6 章“运动的诗章：运行时数据结构”中详细描述。

• **register 关键字**。这个关键字能给编译器设计者提供线索，就是程序中的哪些变量属于热门（经常被使用），这样就可以把它们存放到寄存器中。这个设计可以说是一个失误，如果让编译器在使用各个变量时自动处理寄存器的分配工作，显然比一经声明就把这类变量在生命期内始终保留在寄存器里要好。使用 register 关键字，简化了编译器，却把包袱丢给了程序员。

为了 C 编译器设计者的方便而建立的其他语言特性还有很多。这本身不是一件坏事，它大大简化了 C 语言本身，而且通过回避一些复杂的语言要素（如 Ada 中的泛型和任务，PL/I 中的字符串处理，C++中的模板和多重继承），C 语言更容易学习和实现，而且效率非常高。

和其他大多数语言不同，C 语言有一个漫长的进化过程。在目前这个形式之前，它经历了许多中间状态。它历经多年，从一个实用工具进化为一种经过大量试验和测试的语言。第一个 C 编译器大约出现在 1970 年，距今 20 多年了[1]。时光荏苒，作为它的根基的 UNIX 系统得到了广泛使用，C 语言也随之茁壮成长。它对直接由硬件支持的底层操作的强调，带来了极高的效率和移植性，反过来也帮助 UNIX 获得了巨大的成功。

1.3　标准 I/O 库和 C 预处理器

C 编译器不曾实现的一些功能必须通过其他途径实现。在 C 语言中，它们在运行时进行处理，既可以出现在应用程序代码中，也可以出现在运行时函数库(runtime library)中。在许多其他语言中，编译器会植入一些代码，隐式地调用运行时支持工具，这样程序员就无须操心它们了。但在 C 语言中，绝大多数库函数或辅助程序都需要显式调用。例如，在 C 语言中（必要时），程序员必须管理动态内存的使用，创建各种大小的数组，测试数组边界，并自己进行范围检测。

与此类似，C 语言原先并没有定义 I/O，而是由库函数提供。后来，这实际上成了标准机制。可移植的 I/O 由 Mike Lesk 编写，最初出现在 1972 年左右，可在当时存在的 3 个平台上通用。实践经验表明，它的性能低于预期值。所以，人们对它又进行了优化和裁剪，后来成为标准 I/O 函数库。

C 预处理器大约也是在这个时候被加入的，倡议者是 Alan Snyder。它所实现的 3 个主要功能是：

• 字符串替换：形式类似“把所有的 foo 替换为 baz”，通常用于为常量提供一个符号名。

• 头文件包含（这是在 BCPL 中首创的）：一般性的声明可以被分离到头文件中，并且可以被许多源文件使用。虽然约定采用“.h”作为头文件的扩展名，但在头文件和包含实现

[1] 本书原版出于 1994 年，当时距 1970 年还不到 30 年。——译者注

代码的对象库之间在命名上却没有相应的约定，这多少令人不快。

• 通用代码模板的扩展。与函数不同，宏(marco)在连续几个调用中所接收的参数的类型可以不同（宏的实际参数只是按照原样输出）。这个特性的加入比前两个稍晚，而且多少显得有些笨拙。在宏的扩展中，空格会对扩展的结果造成很大的影响。

```
#define a(y)  a_expanded(y)
a(x);
```

被扩展为：

```
a_expanded(x);
```

而：

```
#define a (y)   a_expanded (y)
a(x);
```

则被扩展为：

```
(y)    a_expanded (y)(x)
```

它们所表示的意思风牛马不相及。你可能会以为在宏里面使用花括号就像在 C 语言的其他部分一样，能把多条语句组合成一条复合语句，但实际上并非如此。

这里对 C 语言的预处理器并不作太多的讨论。这反映了这样一个观点：对于宏这样的预处理器，只应该适量使用，所以无须深入讨论。C++在这方面引入了一些新的方法，使得预处理器几乎无用武之地。

软件信条

C 并非 Algol

70 年代后期，Steve Bourne 在贝尔实验室编写 UNIX 第 7 版的 shell（命令解释器）时，决定采用 C 预处理器使 C 语言看上去更像 Algol-68。早年在英国剑桥大学时，Steve 曾编写过一个 Algol-68 编译器。他发现如果代码中有显式的“结束语句”提示，诸如 if ... fi 或者 case ... esac 等，调试起来会更容易。Steve 认为仅仅一个 “}” 是不够的，因此他建立了许多预处理定义：

```
#define STRING char *
#define IF if(
#define THEN ){
#define ELSE }else{
#define FI ;}
#define WHILE while(
#define DO ){
```

```
#define OD ;}
#define INT int
#define BEGIN {
#define END }
```

这样，他就可以像下面这样编写代码:

```
INT compare(s1, s2)
     STRING s1;
     STRING s2;
BEGIN
    WHILE *s1++ == *s2
    DO IF *s2++ == 0
       THEN return(0);
       FI
    OD
      return(*--s1 - *s2);
END
```

再看一下相应的 C 代码:

```
int compare(s1, s2)
    char *s1, *s2;
{
    while(*s1++ == *s2){
          if(*s2++ == 0) return(0);
    }
    return (*--s1 - *s2);
}
```

Bourne shell 的影响远远超出了贝尔实验室的范围，这也使得这种类似 Algol-68 的 C 语言变型名声大噪。但是，有些 C 程序员对此感到不满。他们抱怨这种记法使别人难以维护代码。时至今日，BSD 4.3 Bourne shell（保存于/bin/sh）依然是这种记法写的。

我有一个特别的理由反对 Bourne Shell，在我的书桌上堆满了针对它的 Bug 报告！我把它们发给 Sam，我们都发现了这样的 Bug: 这个 shell 不使用 malloc，而是使用 sbrk 自行负责堆存储的管理。在维护这类软件时，每解决两个问题通常又会引入一个新问题。Steve 解释说他之所采用这种特制的内存分配器，是为了提高字符串处理的效率，他从来不曾想到其他人会阅读他的代码。

Bourne 创立的这种 C 语言事实上促成了异想天开的国际 C 语言混乱代码大赛（The International Obfuscated C Code Competition），比赛要求参赛的程序员尽可能地编写神秘而混乱的程序来压倒对手（关于这个比赛，以后还有更详尽的说明）。

宏最好只用于命名常量，并为一些适当的结构提供简捷的记法。宏名应该大写，这样便很容易与函数调用区分开来。千万不要使用 C 预处理器来修改语言的基础结构，因为这样一

来 C 语言就不再是 C 语言了。

1.4 K&R C

到了 20 世纪 70 年代中期，C 语言已经很接近目前这种我们所知道和喜爱的形式了。更多的改进仍然存在，但大部分都只是一些细节的变化（比如允许函数返回结构值）和一些对基本类型进行扩展以适应新的硬件变化的改进。（比如增加关键字 unsigned 和 long）。1978 年，Steve Johnson 编写了 pcc 这个可移植的 C 编译器。它的源代码对贝尔实验室之外开放，并被广泛移植，形成了整整一代 C 编译器的基础。C 语言的演化之路如图 1-2 所示。

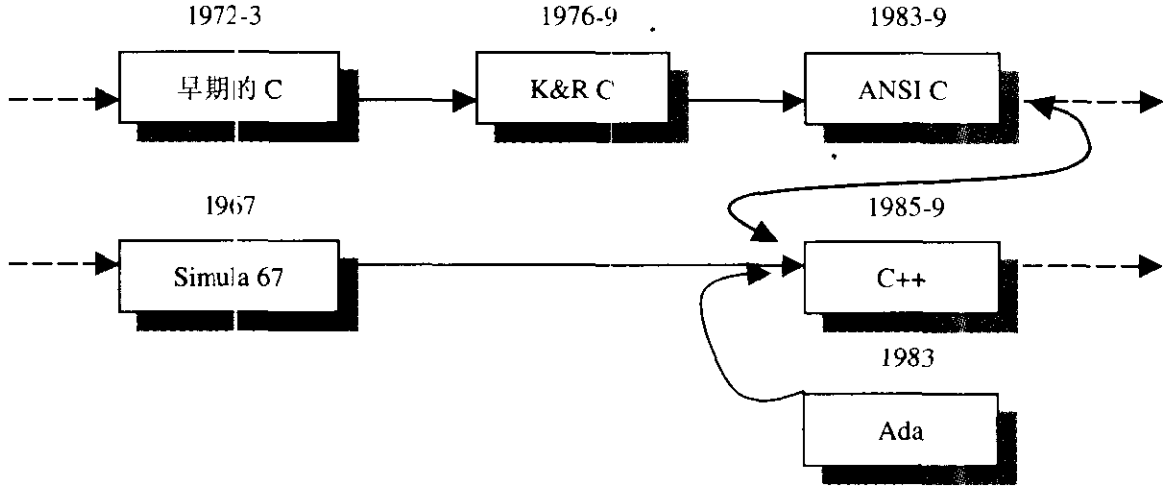


图 1-2　后期的 C

软件信条

一个非比寻常的 Bug

C 语言从 Algol-68 中继承了一个特性，就是复合赋值符。它允许对一个重复出现的操作数只写一次而不是两次，给代码生成器一个提示，即操作数寻址也可以类似地紧凑。这方面的一个例子是用 b+=3 作为 b=b+3 的缩写。复合赋值符最初的写法是先写赋值符，再写操作符，就像：b=+3。在 B 语言的词法分析器里有一个技巧，使实现=op 这种形式要比实现目前所使用的 op=形式更简单一些。但这种形式会引起混淆，它很容易把

```
b=-3;  /* 从 b 中减去 3 */
```

和

```
b= -3;  /* 把-3 赋给 b */
```

搞混淆。

因此，这个特性被修改为目前所使用的这种形式。作为修改的一部分，代码格式器程序

缩进也作了相应修改，用于确定复合赋值符的过时形式，并交换两者的位置，把它转换为对应的标准形式。这是个非常糟糕的决定，任何格式器都不应该修改程序中除空白之外的任何东西。令人不快的是，这种做法会引入一个 Bug，就是几乎任何东西（只要不是变量），如果它出现在赋值符后面，就会与赋值符交换位置。

如果你运气好，这个 Bug 可能会引起语法错误，如：

```
epsilon=.0001;
```

会被交换成：

```
epsilon.=0001;
```

这条语句将无法通过编译器，你马上就能发现错误。但一条源语句也可能是这样的：

```
valve=!open;    /*valve 被设置为 open 的逻辑反*/
```

会悄无声息地交换成：

```
valve!=open;   /*valve 与 open 进行不相等比较*/
```

这条语句同样能够通过编译，但它的作用与源语句明显不同，它并不改变 valve 的值。

在后面这种情况下，这个 Bug 会潜伏下来，并不会被马上检测到。在赋值后面加个空格是很自然的事，所以随着复合赋值符的过时形式越来越罕见，人们也逐渐忘记了缩进曾经被用于“改进”这种过时的形式。这个由缩进引起的 Bug 直到 20 世纪 80 年代中期才在各种 C 编译器中销声匿迹。这是一个应被坚决摒弃的东西！

1978 年，C 语言经典名著 *The C Programming Language* 出版了。这本书受到了广泛的赞誉，其作者 Brian Kernighan 和 Dennis Ritchie 也因此名声大噪，所以这个版本的 C 语言就被称为“K&R C”。出版商最初估计这本书将售出 1000 册左右。截止到 1994 年，这本书大约售出了 150 万册（参见图 1-3）。C 语言成为最近 20 年最成功的编程语言之一，可能就是最成功的。但随着 C 语言的广泛流行，许多人试图从 C 语言中产生其他变种。

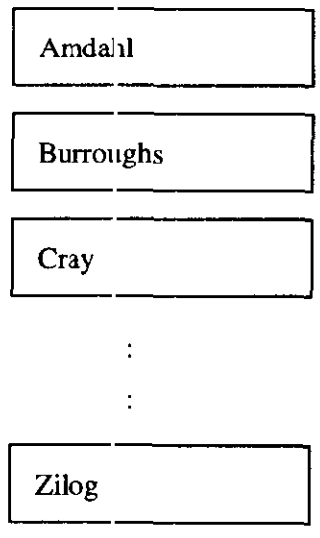


图 1-3　像猫王艾尔维斯一样，C 语言无处不在