C 程序设计奥秘

张自力等译

云南科学技术出版社

第一章 C语言的历史

C是一种奇妙的语言，它有缺点，但获得了巨大的成功。

——Dennis Ritchie

本章主要内容

C语言的诞生
标准I/O库和预处理程序
现在的C
对应用编译工具的限制
了解标准的好处
开发工具定义的特性对语言的影响
早期的C语言
K&R C
标准规定了些什么
ANSI C语言标准的结构
小的变动

1.1 C语言的诞生

从某种意义上说是一次失败导致了C语言的诞生。那是在1969年，Multics计划显然是陷入了困境，此风险计划是由通用电气公司、MIT和贝尔实验室一起进行的，其目的是建立一个新的操作系统。他们希望这个系统又快又方便，但结果却令人失望。虽然开发小组最终让系统动了起来，但由于系统过于庞大，无法在能力相对较弱的硬件上实现。这个系统是解决工程问题的理想工具，也为优美短小的C语言展示其风采铺平了道路。

Multics计划被放弃以后，贝尔实验室原先从事这项工作的人们开始寻求新的任务。一个叫Ken Thompson的研究员对于另外再搞一个操作系统很有兴趣，并向贝尔实验室的管理层多次提出建议，可惜都被婉言拒绝了。在等待任务的这段时间，Thompson和他的同事Dennis Ritchie一起把Thompson的“太空旅行”软件移植到一台刚启用不久的PDP－7机上。“太空旅行”模拟了太阳系中的主要天体，并把它们显示在屏幕上，另外还有一艘太空飞船，可以在各个行星上起落。同时，Thompson还为这台PDP－7精心制作了一个新的操作系统。这个系统比Multics简短得多，完全用汇编语言写成，Brian Kernighan 1970年给它取名“UNIX”，以纪念从Multics中所获得的教训。图1－1展示了早期的C语言、UNIX及相关的硬件。

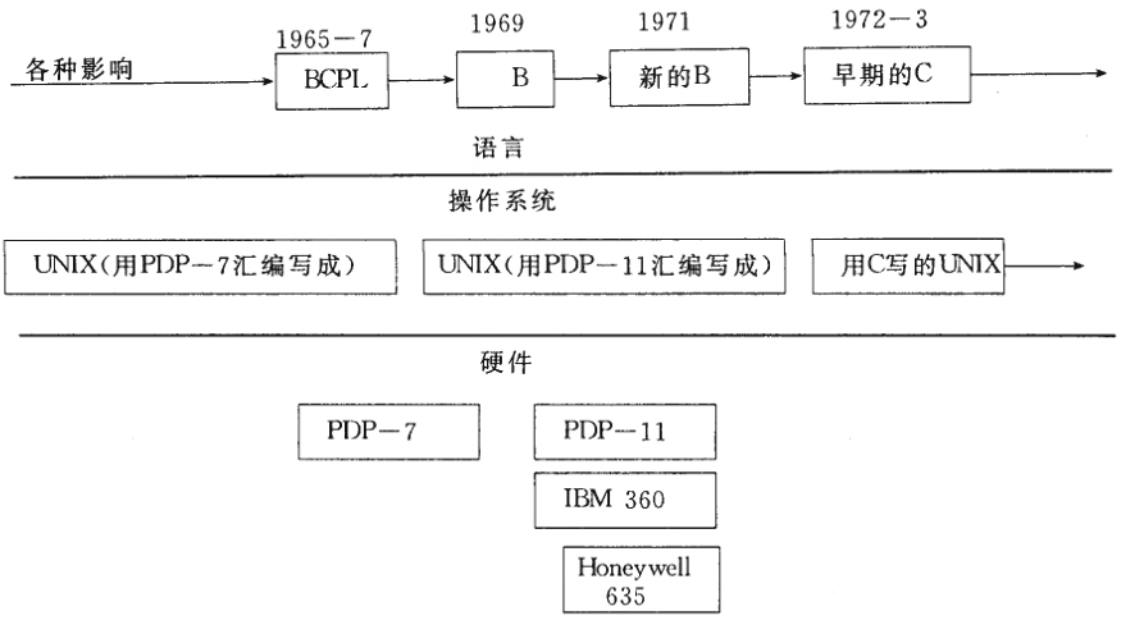

图 1－1　早期的 C、UNIX 及相关硬件

UNIX 与 C 语言的关系有点象鸡生蛋还是蛋生鸡的问题，UNIX 的确比 C 语言先出现，UNIX 系统以 1970 年 1 月 1 日作为系统的起始时间，这正是这种系统的大致出现时间。然而，我们在这里并不想讨论谁先谁后，还是来看看 UNIX 的编写过程吧。

用汇编语言编程有很多不便，如为创建各种数据结构要花很多时间，整个程序不易调试和理解等等。Thompson 希望有这样一种语言，它既有高级语言的优点，又不象 PL/I 语言那样性能糟糕，也没有他在 Multics 中看到的那些复杂问题。在搞了一阵 Fortran 而一无所获之后，他通过简化一种叫 BCPL 的语言(Basic Combined Programming Language 基本组合编程语言，由英国伦敦大学和剑桥大学的研究人员共同开发，在 Multics 上就有一个 BCPL 语言编程工具)而发明了一种叫 B 的语言，B 的解释程序可以装入 PDP－7 只有 8K 机器字的内存中。由于机器的内存只能装下解释程序而装不下编译程序，B 没有取得成功。低性能使 B 未被用于 UNIX 系统的编制。

软件信条

对于编译程序编写者而言：性能(几乎)就是一切！

对编译程序来说，性能几乎就是一切，除此之外还应考虑有意义明确的出错信息、完整的说明文档和产品支持。但这几条与用户对速度的关心相比是不足道的。编译程序的性能包括两个方面：运行时间(编译产生的代码运行起来快不快)及编译时间(产生代码需要多少时间)。除非是在开发或是在进行研究，运行时间通常是最重要的因素。

许多编译程序的优化操作会使编译时间增长但能缩短运行时间。有些优化程序有诸如删除无效代码，放弃运行时检查等功能，既能加快编译和运行，还能减少内存需求。进行优化的坏

处在于使一些错误难于被发现。优化程序总是很小心地进行完全的转换，但程序员可能会写出有问题的代码而引起错误的结果。例如：程序员可能会因为“知道”他们想要使用的变量正好在一个数组的后面而使引用超过了数组的边界。

这就是为什么只能说性能几乎就是一切，如果你最终不能得到正确的运行结果，不论算得多快都没有意义。编译程序编写者通常会提供编译选项，让每个程序员选择想要的优化操作。

在 Dennis Ritchie 写出一个叫“新 B”的高性能编译程序之前，B 语言一直没有成功，正说明速度的重要性。

B 语言去掉了 BCPL 中的某些特征（如嵌套过程和某些跳转结构）并提出了对数组的引用应被“分解”成“指针＋偏移量”的形式的想法。象 BCPL 一样，B 语言也是无数据类型的，唯一可用的操作数是机器字。Thompson 想出了＋＋和－－运算符并把它们加到 PDP－7 机上的 B 语言编译器中。一种流行的说法是：C 语言中之所以有＋＋和－－运算符是因为 PDP－11 计算机的构造非常适合这种自增自减编址方式，这种说法是错误的。自增和自减在 PDP－11 计算机还没造出来之前就有了。由于在 C 语言中将一个字符复制到一个字符串中的语句：

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

可以被特别高效地编译成 PDP－11 的指令：

```
mov (r0)+,(r1)+
```

而使有些人产生了这一错误看法。

1970 年推出 PDP－11 计算机时，无类型的语言就不好用了。新的处理器支持不同长度的多种数据类型，而 B 语言无法表示它们，加上对性能的考虑，Thompson 没有用 B 语言而是用 PDP－11 的汇编语言重新编写了 UNIX 操作系统。Dennis Ritchie 在比 PDP－7 功能更强的 PDP－11 计算机上实现了“新 B”语言，同时解决了多种数据类型问题和性能问题。“新 B”（这个名字很快就演变成了“C”）是编译型的而不是解释型的，它引入了类型系统，每个变量在使用之前都应被定义。

1.2 早期的 C 语言

引入类型系统的目的主要是使编译程序编写者能把浮点数、双精度数、字符和 PDP－11 中的机器字区分开。这与 Pascal 语言中的情况有所不同，Pascal 中的类型系统是为了限制对数据项的有效操作从而保证程序的正确性而建立的。由于设计思想的不同，C 语言不强制类型相同，它允许程序员根据需要进行不同数据类型之间的赋值。

除了类型系统以外，C 语言的许多特征是为编译程序编写者的方便而引入的。这样的特征主要有：

·数组下标从 0 而不是从 1 开始

大多数人都习惯从 1 开始计数，而不是从 0 开始，程序编写者们之所以从 0 开始是因为他们习惯于以位移的方式考虑问题，非专业人员可能对此有些难于理解。虽然在数组定义中写的是 a[100]，你却不能往 a[100]中存放数据，因为数组元素是从 a[0]到 a[99]。

·C 语言的基本数据类型直接对应于硬件

C 语言中不象 Fortran 那样有复数类型，编译程序编写者用不着把精力花在那些不能被硬件直接支持的东西上。在硬件直接支持浮点数之前，C 甚至不支持浮点数。

此为试读，需要完整PDF请访问：www.ertongbook.com

·可以不再使用关键词 auto

对编译程序编写者而言，auto 还有点用，它标识了一个符号表的入口，说明在进入这个块后以自动变量方式分配存储单元，而不是以全局静态变量方式分配或以动态方式从堆中分配。对一般程序员而言完全可以不管它，因为缺省的存储单元分配方式就是自动分配。

·表达式中的数组名被转成了指针

把数组名当指针一样对待使很多问题得到了简化。我们用不着引入一套复杂的机制将数组当作复合对象来处理，也用不着在数组被用作函数参数时对整个数组进行复制。但注意不要错误地认为数组和指针是一回事。在第四章中将对此进行讨论。

·在任何地方都把浮点数扩展成双精度数

虽然在目前的 ANSI C 中情况已有所变化，以前所有表达式中的实型常量和浮点变量都会被转化为双精度型，很少有人对此给出解释。其实这与 PDP－11 计算机的结构有关。首先，在 PDP－11 或 VAX 机上将浮点数转化为双精度数是很简单的一件事：只需要加上一个所有位都为零的机器字；要转换回去只需去掉第二个机器字就行了。另外，有些 PDP－11 计算机的浮点运算硬件中有一个模式位，可以用它来控制硬件是做单精度还是双精度的运算。由于大多数早期的 C 程序都没有用浮点数，把模式位始终置成双精度方式比较方便，编译程序编写者就用不着考虑这个问题了。

·没有嵌套的函数(包含在别的函数中的函数)

这使 C 语言编译程序得以简化并加快了程序的运行速度，对此第六章中有进一步的描述。

·关键字 register

这个关键字告诉编译程序哪些变量经常被使用，程序员希望能把它们存放在寄存器中以提高程序运行的速度。事实证明这样做不好。如果让编译程序根据需要为变量分配寄存器，最终得到的程序比让变量一直保存在寄存器里的程序好。使用 register 使编译程序得以简化，但加重了程序员的负担。

C 语言还有许多特征也是为了方便编译程序编写者而发明的。这并不是什么坏事，通过简化语法，C 语言极大地得到了简化，好学又好用，速度极快。

与大多数高级语言不同，C 语言在长期实践中不断变化，产生了很多变种，最终才发展成今天的形式。第一个 C 语言编译程序大约出现在 1972 年，距今已超过二十年了。随着 UNIX 操作系统的不断传播，C 语言也随之普及开来。C 语言注重硬件直接支持的低级操作，带来了高速度。其可移植性又反过来推动了 UNIX 的传播。

1.3　标准 I/O 库和预处理程序

下面先看一下 C 语言的函数，它既可以由用户自定义也可以由函数库提供，在其它一些语言中，编译程序隐含地产生调用函数和其它支持代码，因而程序员用不着考虑这一问题。但在 C 中，几乎所有函数和支持都必须由程序员自己定义。在必要时，程序员要自己来管理内存、处理可变大小的数组、检查数组下标、查看数据是否越界等等。

输入/输出没有包含在 C 语言中，而是由标准库函数提供。1972 年，Mike Lesk 编写了一个可移植的 I/O 库并在三种硬件平台上实现。因其性能不够理想，后又进行了一些调整，最终成为标准 I/O 库。

与此同时，根据 Alan Snyder 的建议，C 语言中开始出现预处理程序，它主要完成以下工作：

·串替换。如把所有的 f00 替换成 baz，常用于为常量提供一个名字。

·源文件包含。公共定义可以被分离出来，写在一个头文件中，并被多个源文件引用，一般用.h 作为头文件的扩展名，但目前还没有产生一种约定，使头文件和包含相应代码的代码库联系起来。

·常参数宏扩展。与函数不同，宏在被调用时宏参数可以是不同的类型，宏扩展在 C 中出现的时间晚于串替换和源文件包含，并且给 C 语言带来了一些麻烦。如在宏扩展中，空格有时会使结果完全错误：

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

将被展开为：

```
a_expanded(x);
```

然而：

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

被转化成了：

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

宏处理程序本来可以象 C 中其它部分那样，用花括号把一些符号括起来，把它们说明成一个块，但宏处理程序实际上并没有这样做。

这里不再对 C 语言预处理进行更多的讨论，使用的时候小心一些就是了。C++中已对 C 的预处理做了大量改进，引入了一些约定。在 C++中已完全可以不再使用预处理了。

软件信条

C 语言不是 Algol

70 年代后期，Steve Bourne 在贝尔实验室编制 UNIX 第七版的命令解释程序时，为了使 C 类似于 Algol－68 而使用了 C 语句预处理。早年在英国剑桥大学工作时，Steve 曾经写过一个 Algol－68 编译程序，他觉得显式地出现结束语句，如 if…fi 和 case…esac，将有助于程序的调试，而直接查看是否匹配太麻烦了。于是他建立了下列定义：

```
#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   }
```

这使他在编写程序时，可以这样写：

```
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
```

这在C语言中，相当于：

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

随着*Bourne*的命令解释程序在社会上广泛传播，这种变体*Algol*—68也成了一种准标准，给一些C程序员带来了麻烦。他们报怨它使得别人难以对代码进行维护。现在*BSD* 4.3版的*Bourne*解释程序（放在*\bin\sh*目录下）就是用这种“*Algol*语言”写的。

Bourne的C语言版本实际上是鼓励了“世界糊涂C程序大赛”。这是一个古怪的比赛，程序员们尽力编出神秘难懂的程序以压倒对手（后面我们还会提到它）。

宏主要被用于给文字串取名和为结构提供缩写。宏名都采用大写字母，以便在使用时和函数调用区分开来。在C中，完全避免预处理错误是不可能的。

1.4　K&R C

在70年代中期，C语言已经基本形成了我们现在所看到的样子。后来又经历了一系列完善，主要是针对某些细节（如允许函数返回一个结构），并随着硬件的发展增添了基本数据类型

(如加入了关键词 unsigned 和 long)。1978年 Steve Johnson 编写了可移植的 C 语言编译程序 pcc,它的使用范围很快超出了贝尔实验室,成为整整一代 C 语言编译程序的共同基础。图1—2表明了 C 语言的发展历程。

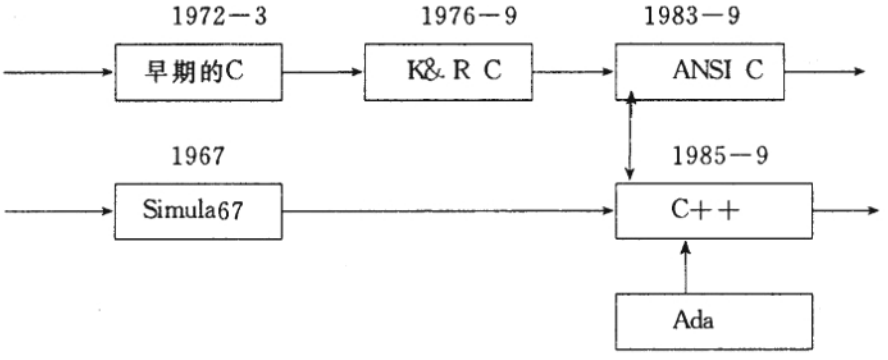

图1—2 后来的 C

软件信条

一个不常见的问题

C 中的复合赋值运算符是从 Algol—68继承来的,它使本需要写两次的操作数可以只写一次。例如:可以把 b=b+3缩写成 b+=3。最初复合赋值运算符中是先写等号而不是先写运算符的,即写成 b=+3;在 B 语言的语义分析程序中"等号+运算符"这种形式比现在通用的"运算符+等号"形式更易实现,但这种形式极易引起混淆。例如:

```
b=-3;
```

可以理解成 b 等于 b 减去3,也可理解成把—3赋值给 b。

因此这一形式被改成了现在通用的"运算符+等号"的形式。由于这一改变,原有的代码规范化程序 indent 也做了调整,以识别以前的"等号+运算符"格式并将其转换成"运算符+等号"格式。这样的调整实在是太糟了,代码规范化程序不应该对程序做这样的修改。程序员碰到的问题是:除了变量以外等号右边的任何东西都有可能被移到等号前面来。

如果你运气好的话,当

```
epsilon=.0001;
```

被转化成

```
epsilon. =0001;
```

时,将会产生一格式错误信息。

但语句:

```
valve=!open; /* valve 等于 open 逻辑取反 */
```

将被转化成:

```
valve!=open ;/* valve 不等于 open */
```

这样的转化在编译时不会产生任何错误,valve 却没有被赋值。

这样的错误很不好检查。如在等号后面加上一个空格的话,事情就好办了。随着"等号+运

算符"这种旧形式在应用中不断减少，人们渐渐忘记了这个用于修正的代码规范程序。80年代中期有时还能看到它引起的错误，真是害人不浅！

1978年，C语言的经典著作《The C Programming Language》发表了，大家都赞成以作者Brian Kernighan和Dennis Ritchie的名字来命名这个C语言版本，即"K&R C"。出版商估计只能卖一千本，实际上到1994年已卖了一百五十万。现在大部分计算机上都有C语言工具，它是过去20年中最成功的编程语言之一。随着C语言的传播，它自身还在不断发生变化。

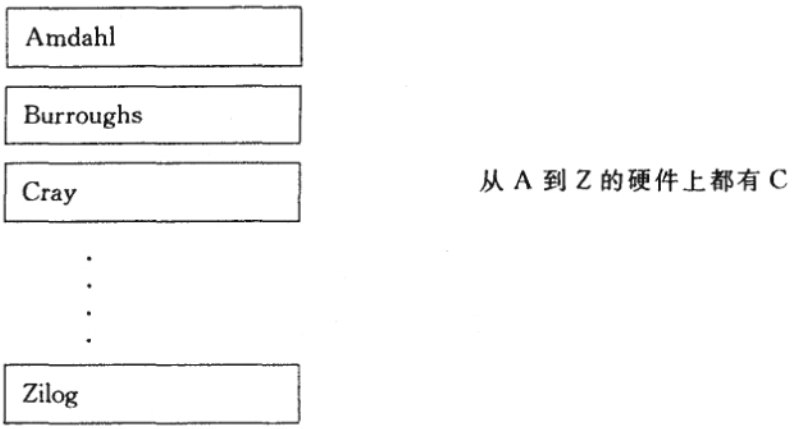

图1—3　无处不在的C

1.5　现在的C：ANSI C

到了80年代初，C语言在工业上得到了广泛的应用。当PC机用户发现C语言强于BASIC后，又掀起了一个新的浪潮。Microsoft为IBM PC提供了C语言开发工具，并引入了一些新的关键词(far，near等)以便在Intel 80x86芯片上实现指针操作。随着非基于pcc的C语言工具的大量出现，C语言有可能重蹈BASIC的覆辙，成为一种版本众多而版本间差异较大的语言。

这时显然需要有一个正式的C语言标准。所幸的是在标准化方面已有了许多成功的先例，大多数成功的编程语言都已有了标准。但标准化手册只有在已懂得了其中的内容的时候你才能读懂它。如果用英语来写这一手册，那么越是写得严谨，篇幅就越长，内容越多越难理解。而如果用数学符号来定义这一语言，大多数人根本无法阅读这种手册。

随着时间的流逝，定义编程语言标准的手册往往会越来越长，但并不因此而变得易懂。Algol—60比C语言相对复杂一些，其参考手册大约有18页；定义Pascal用了35页；Kernighan和Ritchie关于C的最初报告用了40页，其中还有几处漏洞，但对大多数编程者已经足够了。ANSI C的手册是厚厚的一大本，超过200页。这本书相当多的部分描述了ANSI标准文档中没有仔细讲解的东西在具体实践中的用法。

1983年，在美国国家标准学会(ANSI)的支持下，成立了一个C语言工作小组。他们的主要工作是确定C语言中的共同特征，也进行了一些修改并引入了一些新的特征。关键词far和near就曾被讨论了很长时间，最终没有被写入以UNIX为中心的ANSI标准。虽然全世界有5千万台PC机，PC是C语言编程者最多的工作平台，工作小组并不想为了某种特定的硬件结构而修改语言。

手边的启发

使用哪个C语言版本

无论是学习还是使用，都应用ANSI C而不是K&R C。

1989年12月，ANSI最终接受了C语言标准文本。随后国际标准化组织ISO接受了ANSI C但删去了很有用的“原理(Rationale)”一章，并对格式和段落的编号做了一些修改。作为一个国际组织，ISO当然比ANSI更有权威性，因此，1990年初ANSI重新接受ISO C取代它原有的版本。所以，从理论上讲，我们可以说ANSI C就是ISO C。被删去的“原理”一章对理解标准有很大帮助作用，后来它被作为一个单独的文件得以发表。

手边的启发

如何获得C语言标准

C语言标准的正式名称是：ISO/IEC 9899—1990. ISO/IEC是国际标准化组织国际电子委员会的缩写。标准的售价是130美元。如果你是在美国，你可以寄钱给：

American National Standards Institute
11 West 42nd Street
New York, NY 10036
Tel:(212)642—4900

如果你不在美国，可以寄钱给：

ISO Sales
Case postale 56
CH—1211 Genève 20
Switzerland

注意应写清楚你需要哪种语言的文本。

另一个办法是买一本Herbert Schildt著的《The Annotated ANSI C Standard》(New York, Osborne McGraw—Hill, 1993)，书中有一份完整的C语言标准，是用照像技术缩排下来的。

在ISO成立处理C语言的第14工作小组之前，“ANSI C”一词已在广泛使用。工作小组一

此为试读，需要完整PDF请访问：www.ertongbook.com

开始就把标准中的技术发展问题交给了ANSI的X3J11委员会。后来为了使最后的标准能同时为两个组织接受，工作小组与委员会又一起来解决技术问题。为了解决标准中的一些国际化问题，标准的出台还被推迟了近一年。

清楚C语言标准产生过程的人都认为它还是应叫做ANSI C。ANSI C同时也是欧洲标准(CEN 29899)及X/OPEN标准。1991年3月被美国国家标准技术委员会接受为联邦信息处理标准(FIPS 160)。到目前为止C语言还在不断地变化，比如，人们正在讨论向C中加入复数类型。

1.6 标准规定了些什么

ANSI C在某些方面是较为独特的，它定义了一些术语来说明开发工具的特点，了解这些术语有助于了解C语言。前两个术语是关于不可移植代码的，而后两个是关于"坏"代码的，最后两个是关于可移植代码的。

1. 关于不可移植代码

开发工具定义的：由编译程序编写者决定如何处理，并把处理的办法写入文档。

例：在右移一个整数时，符号位(最高位)应如何变化。

未指定的：某一正确的操作，但标准中没有指定其具体实现方式。

例：函数参数的计算次序。

2. 关于"坏"代码：

未定义的：某一操作不正确，但标准中没有给出处理方法。这时，可以发生任何事情：也许平安无事，也许是程序终止并发出警告，也许是CPU损坏，也许是发射原子弹(如果你的计算机可以控制原子弹的发射的话)。

例：在有符号整数溢出时计算机应作的处理。

约束：它是必须遵守的规定，如果你不遵守，程序的行为将会变成"未定义的"。要讲清楚一个东西是不是约束很容易，在标准中每一个主题后都有一个叫"约束"的段落，列出了所有有关的约束。令人吃惊的是标准只规定编译程序在遇到格式错误和违反约束时必须产生错误信息！这意味着可以使用任何不违反约束的语义规则，这时计算机的行为是"未定义的"，编译程序无需向你发出警告就可以为所欲为。

例：%运算符的操作数必须是整型，如果使用非整型量将产生出错信息。

不是约束的例子：C语言标准头文件中的所有标识符对于应用开发工具来讲都是保留字，因此你不能定义名为malloc()的函数，因为标准头文件中已有此函数名了。但由于这条规则并不是一个约束，如果你违反它，编译程序不会为此向你发出警告！关于此问题的更多讨论可参看第五章中的"干涉"。

软件信条

在IBM PC中未定义的行为可能引起硬件损坏。

有未定义的行为的软件会使硬件损坏这一观点并不难于理解，下面来看一个例子。最初的

IBM PC计算机的显示器中有一块视频控制芯片控制水平扫描速率，在此速率下，回扫变压器（一个产生高压来加速射向荧光屏幕电子束的装置）有合理的扫描频率。然而如果有一个软件把视频控制芯片的扫描速率置成零（这在理论上是可行的），将使变压器的输入电压保持恒定。这相当于把一个电阻得到的能量全部转化成热而不是通过电子传到荧光屏上。几秒钟之内，显示器就会被烧坏。

警告：有未定义的行为的软件可能损坏计算机系统！

3. 关于可移植代码

严格一致的：一个严格一致的程序应具有以下特征：

1）只使用指定的特征（不使用**未指定的**特征）。

2）不使用任何开发工具定义的特征。

3）不产生任何开发工具定义的或未指定的，或未定义的特征所规定的结果。

这是在尽力描述可移植程序，这样的程序不论在什么机器上运行都产生相同的结果。由于这类程序相对较少，人们对它们并不太感兴趣。下面的程序就不是严格一致的：

```
#include <limits.h>
#include <stdio.h>
int main(){(void) printf("biggest int is %d",INT_MAX);return 0;}
/* 不是严格一致的，输出是开发工具定义的 */
```

在本书以下部分，我们并不刻意追求程序是严格一致的，那样只会把文字搞复杂，不利于看清讨论中的要点。程序的可移植性很有价值，在实际编程时应注意这一点。

一致的：一致的程序依赖于第一开发工具的不可移植的特征。一个程序是否是一致的是由特定的开发工具决定的，换一个开发工具它就可能变为不是一致的。无论如何变化，一致的程序不会变成严格一致的，但你不要指望编译程序在发现你的程序不是一致时会向你发出警告。

上面的例子程序就是一致的。

1.7 对应用编译工具的限制

对编译程序能够处理的程序中各种语言因素的下限ANSI C标准实际只有很少一点限制。这些限制大都出现在标准的5.2.4.1节，主要讲一行中可以有多少字符，以及对数组维数的规定。在编译语言的标准中写清楚各种语言因素的下限虽也有先例，但极为少见。标准委员会的先生们认为给出一些参考性的数值还是很有意义的。

每个ANSI C语言编译程序至少应支持：

·函数定义中形式参数可达31个

·函数调用时可以有31个实际参数

·一行源程序可以有509个字符

·一个表达式中可以有31级嵌套的小括号

·long int的最大允许值不小于2,147,483,647（即长整型数据至少是32位的）等等。

另外，一个一致的编译程序必须能一次性编译通过具有上述所有特征的程序。使人惊讶的是：这些限制只是“应该的”，而不是“约束”，编译程序可以不遵守它们。

对编译程序的限制一般被认为是一个“实现质量”问题，在ANSI C标准中包含这一部分

内容是因为如果所有的应用工具都具有较相似的特性，代码的移植就会更容易一些。当然，一个好的应用工具是不应该有任何限制的，只有诸如可利用内存大小，磁盘大小等外部因素才对它的能力有所影响。

1.8 ANSI C 语言标准的结构

翻翻 ANSI C 语言标准很有好处。标准有四个主要章节：

第4节：术语简介与定义(5页)。

第5节：环境(13页)。

这一节是关于运行 C 语言的系统的，包括程序开始及结束时应做什么及符号和浮点运算等等。另外还给出了对各种语言因素的规定和字符集。

第6节：C 语言(78页)

标准的这一部分是根据 Dennis Ritchie 的"C 语言参考手册(The C Reference Manual)"写成的。"C 语言参考手册"在很多出版物中都可以看到，在《The C Programming Language》的附录中就有。如果你把附录中的版本与标准中的内容比较的话，会发现大部分标题都是一样的，而且连程序都一样，只是标准中格式更整齐些。如图1－4所示。

ANSI C 标准中段落的一般形式	ANSI C 标准中段落的例子
段落编号 主题	6.4节 常量表达式
格式	**格式**
格式说明	常量表达式： 常量条件表达式：
描述	**描述**
对特征的一般描述	一个常量表达式可以在编译时而不是运行时进行计算，并可以出现在任何可以使用常量的地方
约束	**约束**
如果违反了此处的任何一条规则，编译程序都必须产生出错信息	常量表达式中不能包含有赋值、自增、自减和逗号运算符，也不能有函数调用，除非它们是在 sizeof 运算符的操作数中。每个常量表达式的计算结果应该是一个常量，其值应在相应数据类型的可表示范围内。
语法	**语法**
特征的含义或作用	计算结果为常量的表达式在很多地方都可能出现。如果在编译时计算了一个浮点表达式，其精度和……
例子	
说明特征的一段代码……	

图1－4　ANSI C 标准的格式

附录中的内容只有40页，而标准中这一节比它长一倍。

第7节：C语言运行库(81页)

这一章是库函数列表，列出了开发工具必须提供的、完成各种功能的标准函数。这一节是根据/usr/group 1984标准编写的，只是去掉了其中有关UNIX的部分。/usr/group起初是一个国际性的UNIX用户组织，1989年被更名为“UniForum”，现是一个非赢利性的行业协会，致力于UNIX操作系统的普及工作。

Uniforum从行为角度定义UNIX所获得的成功激励了一批相关的项目，如：X/Open可移植性指南(第4版，1992年10月发表)，IEEE POSIX 1003，系统V接口定义和ANSI C标准库，大家都和ANSI C工作小组协调，确保每一个标准都是相互一致的，这真是太好了。

ANSI C标准还有一些非常有用的附录：

附录F：公共警告信息

产生警告信息对编程常常会有所帮助。

附录G：可移植性问题。

这里把标准中有关可移植性的建议收集在一起，包括有关未指定的、未定义的和开发工具定义的操作的信息。

软件信条

标准中也可能有错误

ANSI C是国际标准，但这并不能保证它是完全的，一致的，甚至不能保证它是正确无误的。

在IEEE POSIX 1003.1—1988标准(它是一个定义类UNIX行为的操作系统标准)中就有如下的矛盾：

“[路径名]……最多由包括结束符NULL在内的PATH_MAX个字节组成”——第2.3节

“PATH_MAX是一个路径名中的最大字节数(不是串长度，不包括结束符NULL)”——第2.9.5节

因此PATH_MAX个字节既包含又不包含NULL结束符。这里就需要有个说法。IEEE std 1003.1—1988/INT，1992辑的解释是：这里的确是出现了不一致，两者都可以说是正确的(有点奇怪是不是？一般大家会认为不可能两个都正确)。

产生这一问题是因为在起草标准过程中曾作过的一个修改，本来应把相应的描述全部改过来，但实际上没有全部改到。产生和修改标准的过程都是极为严格规范的，这里的错误只有等专家小组下一次投票表决以后才能改正。

C语言标准中也有类似的错误。在它的一个脚注中引用了“原理”一节的内容。实际上，“原理”一节在ISO接手标准时被删去了，没有出现在后来的标准中。

手边的启发

K&R C与ANSI C的区别

如果你了解K&R C，你也就知道了ANSI C 90%的内容。ANSI C与K&R C的区别主要有四个方面，现按其重要程度排列如下：

1、第一部分是新的，有重大区别并且很重要的东西。这种东西只有一个，那就是原型——把形参类型作为函数说明的一部分。原型使编译程序能方便地对函数调用和定义进行检查。

2、第二部分是新的关键词。有的关键词被正式加在标准中了，如针对枚举类型的enum和const，volatile，signed，void及与之相应的语义。在C中从不被使用的关键词entry已经没有了，这显然是疏忽所致。

3、第三部分是一些小改动。包含一些依然有效但意义略有变化的特征。这种特征很多，但都不是很重要，除非你在实践中碰上了，一般可以忽略。例如，现在的预处理规则比以前更为严格，有一条新加入的规则是：邻近的文字串是联在一起的。

4、余下的东西属于第四部分。包含一些在标准化时反复讨论但你在实际编程中几乎肯定没有碰到过的东西。例如三元组，它是一种用三个字符表示一个特定计算机字符集中没有的字符的方法，就象用二元组'\t'表示"tab"一样，可用三元组'??('表示"{"。

标准中最重要的新特征是原型，这一特征是从C++那里学来的。原型是对函数说明的扩展，使其不但包含函数名和返回值，还包含所有参数的类型。这使编译程序能够检查参数在实际调用和定义中是否一致。原型不完全是"一个函数名及其所有参数"的意思，最好应象在Ada语言中那样，将其视为"函数标识体"或"函数说明体"。

软件信条

关于原型的协定

使用原型就是在说明一个函数时不仅给出函数名和返回值，还要包括函数参数类型，从而使编译程序能根据参数类型来检查函数调用时的实参。在K&R C中，这种检查在链接时进行或者干脆不进行。早期头文件中的说明格式是：

```
int strcpy();
```

现在都被改成了：

```
int strcpy (char *dst,const char *src );
```

你可以省略形式参数的名字，只写类型：

```
int strcpy (char *,const char *);
```

虽然编译程序不需要形式参数的名字，但它们常常含有有关参数的信息，最好不要省略。

同样，函数的定义也从：

```
int strcpy (dst,src)
char *dst, *src;
{……}
```

变为：

```
int strcpy (char *dst,const char *src)/* 注意这里不再有分号！*/
{……}
```

函数头后面直接跟作为函数体的复合语句，不再带分号。

现在写函数都应写上原型并保证函数调用与它一致。你用不着在以前按K&R C标准写的代码中加入原型，除非是碰上了缺省类型转化问题。关于这一点，在第8章中将进一步讨论。

1.9 了解标准的好处

有时需要读一读ANSI C标准，从中能找到某些问题的答案。曾有一位销售工程师把下面的这段代码送到Sun的编译程序小组进行测试：

```
1 foo (const char **p) {}
2
3  main (int argc,char **argv)
4 {
5          foo(argv);
6 }
```

在编译时编译程序产生如下警告信息：

```
line 5:warning:argument is incompatible with prototype
```

(行5:警告:参数与原型不匹配)

这位工程师想知道为什么会产生警告，另外，ANSI C标准中哪一部分对此有所说明。他认为：

实际参数char *s是可以和形式参数const *p相匹配的。运行库中的所有串函数就都是这样。实际参数char **argv就不能与形式参数const char* *p匹配了吗？

的确如此。要理解这一结论需要花一点时间。下面的分析是Sun公司的一位专家给出的：

在ANSI C标准6.3.2.2节的约束部分有这样一句：

每一个实际参数都应有一个类型，它的值可以赋给一个对象，这个对象的类型是与实际参数相对应的形式参数的类型的非限定形式。

这说明实际参数的传递可以被看成是赋值。

因此，如果把char**类型的值赋给const char**类型的对象就一定会产生诊断信息。为了搞清楚哪种赋值合法，翻到关于赋值的6.3.16.1节，可以看到如下约束：

以下几点应该遵守：……

此为试读，需要完整PDF请访问：www.ertongbook.com