**C语言发展历史**

**概要**

C语言作为最初的[Unix操作系统](http://www.techcn.com.cn/index.php?doc-innerlink-Unix%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F)的系统实现语言，在1970年早期诞生。它以无类型的B语言为基础，形成了一个类型结构；它在小型机上被发明，作为一个工具，以改进贫乏的编程环境。在今天，它已经成为一种占统治地位的语言。本文研究它的发展。



*Left:* Ken Thompson, developer of *B  
Right:* Dennis Ritchie, developer of *C*

**简介**

版权提示：*版权所有1993计算机协会公司。此电子版由作者授权发表。有关出版事宜，请联系ACM或作者。本文被用于一九九三年四月，在马萨诸塞州剑桥的第二次编程语言历史会议上的演讲。   
之后被收录进会议记录：编程语言历史，第二版。Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. ACM Press (New York) and Addison-Wesley (Reading, Mass), 1996; ISBN 0-201-89502-1.*

本文是关于C程序设计语言的发展、它所受到的影响以及它诞生的条件。为简洁起见，我略过了对C本身完整的描述、它的起源B[Johnson 73]及更上一代BCPL[Richards 79]，相反集中关注每一种语言的典型特性，以及他们如何发展变化。

C在1969——1973年间与Unix操作系统同时诞生；最富创造性的时期是1972年。另一次大的变化发生在1977到1979年间，当Unix系统的可移植性得到证明时。在后一段时期的中间，这个语言的第一个被广为传播的描述出现了：C程序设计语言，常常被称为白皮书或K&R[Kernighan 78]。最后，在1980年代中期，它被ANSI X3J11委员会正式标准化，作了进一步修改。直到1980年代早期，尽管编译器已在多种机器结构及操作系统上出现，这个语言几乎与Unix特别密切关联；更近一些，它的使用传播得更广，并且在今天它几乎是计算机行业被使用最多的语言。



**历史：环境**

[贝尔电话实验室](http://www.techcn.com.cn/index.php?doc-innerlink-%E8%B4%9D%E5%B0%94%E7%94%B5%E8%AF%9D%E5%AE%9E%E9%AA%8C%E5%AE%A4)的计算机系统研究在1960代年晚期是乱糟糟的[Ritchie 78] [Ritchie 84]。贝尔电话实验室公司正从他和[麻省理工学院](http://www.techcn.com.cn/index.php?doc-innerlink-%E9%BA%BB%E7%9C%81%E7%90%86%E5%B7%A5%E5%AD%A6%E9%99%A2)、[通用电气公司](http://www.techcn.com.cn/index.php?doc-innerlink-%E9%80%9A%E7%94%A8%E7%94%B5%E6%B0%94%E5%85%AC%E5%8F%B8)的合作项目Multics[Organick 78]撤离。到1969年，贝尔实验室管理层和研究人员认为，Multics项目不能按期完成并且代价高昂。在GE-645 Multics机器被撤走之前，Ken Thompson领导一个非正式小组，开始一些其它的研究。

Thompson希望按自己的设计，使用一切有效的方式，创造一个适当的计算环境。他的计划，回想起来是清晰的。组合Multics中的许多创新特征，包括一个作为控制场所的进程的清晰概念、一个树结构文件系统、一个作为用户级程序的命令解释器、文本文件的简单表示和访问设备的通用化。他们排除其余特性，比如对内存和文件的统一访问。开始，他与我们其余人推迟了Multics中的另一个先锋性（但非原创）特征，即只使用[高级语言](http://www.techcn.com.cn/index.php?doc-innerlink-%E9%AB%98%E7%BA%A7%E8%AF%AD%E8%A8%80)来编写。我对Multics实现所用的语言PL/I，并无兴趣，但我们还使用其他语言，包括BCPL，我们对于不能利用汇编之上的高级语言进行编程的优点，比如容易编写、易于理解，感到遗憾。当时我们并未特别注重可移植性；到后来才有了这方面的兴趣。

Thompson面临的是那个时代古怪和难缠的硬件：DEC PDP-7，他在1968年开始使用时，只有8K容量的16位内存，并且没有可用的软件。当时他希望使用高级语言，但还是用PDP-7汇编编写了最初的Unix系统。开始，他并未在PDP-7上编码，相反使用一套GE-635机器上，用于GEMAP汇编器的宏。一个后期处理器生成PDP-7可读的纸带。

这些纸带从GE机器传送到PDP-7进行测试，直到一个原始的Unix内核、一个编辑器、一个汇编器、一个简单的shell（命令解析器），和其它工具（像Unix rm, cat, cp命令）被完成。此后，这个操作系统可以自我支撑：可以编写、测试程序勿需借助纸带，并且程序开发可以在PDP-7上继续进行。

Thompson的PDP-7汇编器在简明性上甚至优于DEC的；它对表达式求值并得到对应的数据位。没有库、装载器或链接器：程序的全部源文件被送给汇编器，输出文件——有一个固定名字——产生后被直接执行（这个名字，a.out，解释了一点Unix的渊源；它是汇编器的输出。甚至在系统有了链接器和显式指定另一个名字的方式之后，它仍被保留作为编译的默认可执行文件）。

在Unix首次在PDP-7运行后不久，Doug McIlroy在1969年创造了新系统的第一个新语言：一个McClure的TMG[McClure 65]实现。TMG是一种编写编译器（更普通来说，TransMoGrifiers）的语言，通过在一个混合过程元素，上下文无关的语法标记的自顶而下，递归降解的风格。McIlroy和Bob Morris使用TMG为Multics编写了早期的PL/I编译器。

为了挑战McIlroy重新生成TMG的技巧，Thmopson决定Unix——当时可能还没有取这个名字——需要一种系统编程语言。在很快用Fortran尝试一番后，相反，他创造了一种他自己的语言，他命名为B。B可以认为是没有类型的C。更准确，它是被挤进了8K字节内存，经过Thompson大脑过滤的BCPL。它的名字最有可能表示BCPL的缩写，尽管另一种理论认为它继承自Bon[Thompson 69]，一个Thompson在Multics的那些日子创造的不相关的语言。Bon可能是以他妻子Bonnie的名字，或者（根据它的手册中的一个encyclopedia引用）以一种具有咕隆咕隆发音的神奇仪式的宗教命名的。

**起源：语言**

BCPL由Martin Richards于1960年代中期在访问麻省理工学院时设计，在1970年代早期被用在几个有趣的项目中，其中包括牛津大学的OS6操作系统[Stoy 72]，和施乐公司PARC研究中心创造性的Alto上的部分工作[Thacker 79]。因为Richards工作过的麻省理工学院的CTSS系统[Corbato 62]被用于Multics开发，我们也熟悉该语言。最初的BCPL编译器被Rudd Canaday和贝尔实验室的一些人们迁移到Multics和GE-635 GECOS系统[Canaday 69]；在Multics的生命在贝尔实验室的最后痛苦挣扎中，它很快成了那些以后参与Unix的人们选择的语言。

BCPL, B和C全都严格符合以Fortran和Algol 60代表的传统过程类型语言。它们都面向系统编程、小、定义简洁，以及可被简单编译器翻译。它们接近机器，它们引入的抽象以传统计算机提供的具体数据类型和操作为基础，它们依赖于输入输出库例程，与操作系统的其它交互。尽管并未成功，它们还使用库程序指定其他有趣的控制结构，如协程和过程关闭。同时，它们的抽象层次足够高，足够用心的话，能达到机器间的可移植性。

BCPL, B和C在语法上差异众多，粗略地说，它们是相似的。程序由全局声明和函数（过程）声明组成。BCPL中的过程能够嵌套，但不能引用包含过程中的非静态对象。B和C避免了这个限制，通过强加一个更严格限制：完全没有嵌套过程。每一种语言（除了早期版本B）都认可分离编译，以及提供了包含指定文件文本的方式。

BCPL中的几个语法和词法机制是优雅和常见的，甚于B和C中的那些。例如，BCPL的过程和数据声明有更一致的结构，并且它提供了一套更完整的循环构造。尽管BCPL程序在概念上是由未被间隔的字符流，聪明的规则允许语句后的行分界处的大多数分号被忽略。B和C忽略了这种便利，并以分号来结束大多数语句。不管这些差异，BCPL的大多数语句和操作符直接对应B和C中的相应语句和操作符。

BCPL和B之间的一些结构化的差异源于介质存储的限制。比如，BCPL声明采用这样的形式

let P1 be command and P2 be command and P3 be command ...

此处的命令表示的程序文本包含完整过程。关联的子声明同时出现，所以名字P3在guochengP1内可见。相似地，BCPL能在一个求得一个值的表达式里包含一组声明和语句，例如

E1 := valof ( declarations ; commands ; resultis E2 ) + 1

BCPL编译器可以容易地处理此类构造，在产生输出前，通过存储和分析内存中一个完整程序解析过的表示。B编译器的存储限制要求一个一步技术，通过它尽可能快生成输出，语法上的重新设计，令这种可能迈进了C。

BCPL中一些不令人满意的地方归因于它的技术问题，在B的设计中它们被有意识的避免了。例如，BCPL使用一个“全局向量”（global vector）机制以在分离编译的程序间通信。在这种模式中，程序员使用一个全局向量的数值偏移量，显式关联每个外部可见过程和数据对象的名字。链接使用这些数值偏移量，在被编译过的代码上完成。B起初坚持，整个程序一次性全部传递给编译器，来规避这个麻烦。B的后期实现，和C的全部实现，使用一个传统的链接器，来解决出现在分离编译文件中的外部名字，而不是把指定偏移量的负担推给程序员。

BCPL到B的转换中引入的其它变化，大概是因为风格的缘故，一些仍是有争议的，例如赋值使用单个字符=代替:=。类似地，B使用/\*\*/来括起注释，而B使用//注释直至行末的文本。这显然是从PL/I继承来的。（C++重新启用了BCPL的注释惯例。） Fortran影响了声明的语法：B的声明以一个auto, static这样的类型指定符开始，跟着一列名字，C不仅遵循这种风格，还把它的类型关键字，加入这种声明的开始处。

在Richards的书中文档化的BCPL与B之间的差别，并非都是经过深思熟虑的；我们是从一个BCPL[Richards 79]的早期版本开始工作的。例如，用于跳离switchon语句的endcase在我们1960年代开始学习该语言时，并没有出现，所以B和C中重复出现的，用于跳离switch语句的关键字break，乃是一种背离的发展，而不是清醒的改变。

对比B产生过程中发生的普遍的语法变化，BCPL的核心语义内容——类型结构和表达式求值——保持不变。它们两种语言都是无类型的，或更恰当地说有一种单一的数据类型，“字”（word）或“单元”（cell），一个固定长度的位模式。这些语言中的内存由此类单元的线形数组组成，每一个单元的内容的含义与应用的运算相关。例如，求和运算符使用机器的整数加法指令，简单相加其运算对象，其它算术运算同样不清楚它们运算对象的含义。因为内存是一个线形数组，只可能解析单元的值为该数组的索引，并且BCPL为这个目的提供一个运算符。在最初的语言中，它被拼写为rv，后来为!，但是B使用一元\*。因此，如果p是单元，包含另一个单元的索引（其地址，或指向的指针），\*p引用被指向单元的内容，作为表达式的值或赋值对象。

因为指针在BCPL和B中只不过是整型内存数组的索引，对它们进行算术运算是有意义的：如果p是一个单元的地址，那么p+1是下一个单元的地址。这种约定是两种语言中数组语义的基础。在BCPL中，一个人这样写

let V = vec 10

或在B中，

auto V[10];

效果是一样的：分配了一个名字为V的单元，然后保留另一组10个连续单元，它们中第一个的内村索引，被存放在V中。按照一般的规则，在B中的表达式

\*(V+i)

把V和i相加，并指向V后第i个位置。BCPL和B都增加了特别的符号，使这种对数组的访问更简洁；在B中的等价表达式是

V[i]

在BCPL中是

V!i

这种引用数组的方法甚至在当时仍是不常见的；C后来同化它为一种更不常规的方式。

BCPL，B或C都没有强烈支持字符数据；每一个都把字符串当作整型数组，并通过一些惯例提供了一些一般规则。字符串字面值在BCPL和B中表示一个使用串内字符初始化的静态区的地址，被包装成单元。在BCPL中，第一个被包装的单元包含串所拥有的字符个数；在B中，没有此计数，字符串以一个特别的字符终结，在B中杯拼写为“\*e”。这个改变部分是为了避免把计数值放在一个8位或9位槽（slot）产生的串长度限制，部分是因为维护这个计数，从我们的经验看来，不如使用一个终结符方便。

在BCPL，串中每个字符的使用，是通过被展开为另一个数组，一个字符对应一个单元，然后进行再次包装；B提供了对应的例程，但人们更多地使用，另外的访问或替换一个串内字符的库函数。

**更多历史**

在TMG版本B工作后，Thompson利用B重写了B（编译器）（一个bootstrapping步骤）。在开发中，他不断与内存限制作斗争：每次语言版本使编译器膨胀令内存几乎不够使用，但每次重写利用语言特征的优点，减少了它的尺寸。例如，B引入通用赋值运算符，使用x=+y来把y加入x。这个符号经过McIlroy引自Algol 68[Wijngaarden 75]，他将它合并到他实现的一个TMG版本。（在B和早期C，该运算符被拼作=+而不是+=；这个由B的词法分析的第一种形式的迷惑捷径导致的错误，在1967年被修复。）

Thompson通过发明自增++和自减--运算符，走出了更深远的一步；它们的前缀或后缀位置决定变更是发生在计算运算对象值之前或之后。它们没有出现在B的最早版本中，而是随后才出现的。人们经常猜测，它们被创造是为了使用，C和Unix在其上首次流行的DEC PDP-11提供的自增和自减地址模式。这在历史上来说是不可能的，因为B被发明的时候还没有PDP-11。PDP-7有一些“自增”内存单元，使用这种特性，一个间接内存引用通过它们来自增单元。这些特征可能提示Thompson创造了那些自增运算符；他把前缀和后缀一般化。甚至，自增单元没有被直接用于实现这些运算符，并且这种创新一个更强烈的动机可能是，他发觉++x的翻译在尺寸上小于x=x+1。

PDP-7上的B编译器不产生机器指令，而是一个由编译器输出组成代码段地址序列，执行基本运算的解释模式的threaded代码[Bell 72]。这些操作——特别对B——典型地运行在一个简单堆栈机器上。

在PDP-7的Unix系统上，除了B本身只有几个东西是B写的，因为这个机器太小和太慢，除了试验而不能做更多事情；完全用B重写操作系统和其它应用程序，是看起来不可行的代价高昂的动作。Thompson在某些地方，通过提供一个利用换页解释器代码和数据，允许解释超过8K字节的程序的“虚拟B”编译器，来释放地址空间，但它对通用程序来说太慢以致不实用。尽管如此，一些用B写的工具还是出现了，包括一个早期版本的，Unix用户熟悉的可变精度计算器dc[McIlroy 79]。我做的最有雄心壮志的工作，是一个把B翻译为GE-635机器指令而非threaded 代码的真正的交叉编译器。它是一个精巧的绝技：一个用本身语言写的，生成在一个，在有4k字长用户地址空间的18位机器上运行的36位大型机代码，完全的B编译器。这个项目能实现，仅仅是因为B的简单性和它的运行时系统。

尽管我们抱有关于实现一个那时，像Fortran, PL/I或Algol 68的主要语言的偶然想法。这样的项目对我们的显得绝望的大：需要更简单和小的工具。所有这些语言都影响我们的工作，但是凭我们自己之力来做这些事情则更有趣。

到1970年时，我们看起来能在Unix项目上，获得一个新的DEC PDP-11。处理器是DEC递交的第一批产品，三个月后，磁盘才到达。通过threaded技巧，使B程序在其上运行只需要为运算符重写代码段，和一个我用B写的简单的汇编器。很快，dc成了在其它操作系统之前，第一个在我们的PDP-11上被测试的有趣的程序。几乎非常快，但仍需等待磁盘，Thompson用PDP-11汇编语言，重写了Unix内核和一些基本命令。最早的PDP-11上的Unix把机器上24K内存中的12K给操作系统，一个很小的空间给用户程序，其余的作为RAM磁盘。这一版本仅是用于测试，而不是实际的工作；这个机器通过枚举关闭的，knight的不同尺寸象棋板的路程，来标记时间。在磁盘到达后，我们把汇编语言转换为PDP-11上的方言，和移植一些B程序，很快移植到它上面去。

到1971年时，我们的微型计算机中心开始有了用户。我们都希望更容易编写有趣的软件。使用汇编显得沉闷，B不管它的性能问题，已经有了一个小的包含有用服务例程的库，并且被用于越来越多的新程序。这段时期的最著名的成果，是Steve Johnson的yacc分析——生成器[Johnson 79a]的第一个版本。

**B的问题**

我们第一次使用BCPL然后是B的机器，是按字寻址的，这些语言的单一数据类型，“单元”，能恰当与硬件机器字互相换算。PDP-11的出现暴露了B的语义模型的一些不足。首先，它从BCPL继承的几乎未作改变的字符处理机制是笨拙的：使用库方法把包装的字符串展开到单个的单元，然后再次包装，或者访问或替换单个字符，在一个面向字节的机器上，开始变得笨拙，甚至愚蠢。

其次，尽管最初的PDP-11没有提供浮点算术运算，制造商承诺将很快提供。浮点运算通过定义特别的运算符，被添加到我们的Multics和GCOS的B编译器，但是这种机制仅在相应的机器上才可能，单个字长足够包含一个浮点数；这在16位PDP-11上是不成立的。

最后，B和BCPL模型在处理指针时，暗中会做得更多：语言规则，通过定义一个指针作为字数组的索引，强迫指针被表示为字索引。每个指针引用生成一个运行时，从指针到硬件要求的字节地址的度量转换。

因为这些理由，看起来需要一个类型模式来处理字符和字节寻址，以及为即将到来的浮点硬件作准备。其它问题，特别是类型安全性和接口检查，看起来并没有变得像以后那样重要。

除了语言本身的问题，B编译器的threaded代码技术得到的程序，比他们对应的汇编语言版本慢很多，以至我们对用B纪录操作系统或它的中心工具的可能性打折扣。

到1971年时，我开始通过添加一个字符类型，并重写它的编译器以生成PDP-11机器指令而非threaded代码，来扩展B语言。因此从B到C的转换，与创造一个同汇编语言竞争，能产生足够快和小的程序的编译器，是同时进行的。我称这个轻微扩展的语言为NB，表示“新B”（new B）。

**C萌芽**

NB只存在了很短时间，以至没有编写一个它的完整描述。它提供类型int和char，它们的数组，指向它们的指针，用典型风格声明如下

int i, j; char c, d; int iarray[10]; int ipoint[]; char carray[10]; char cpoint[];

数组的语义与在B和BCPL中保持一样：iarray和carray的声明产生的单元，被动态初始化为分别指向十个整数和字符序列中的第一个的值。ipointer和cpointer的声明省略了尺寸，以表明没有存储被自动分配。在过程内部，语言对指针的解释与数组变量是一样的：一个指针声明产生一个单元与数组声明的区别仅在，程序员被期望给它赋值，而不是让编译器分配空间和初始化单元。

值存储在数组的单元中，指针是按字节计算的，对应存储区的机器地址。因此通过指针间接引用，不意味着比按比例缩放指针从字到字节的偏移，有更多运行时开销。另一方面，对应数组取下标的机器代码和指针算术依赖于数组和指针的类型：计算iarray[i]或ipointer+i表示按比例缩放加数i与所指向对象的尺寸。

这些语义表示一个来自B的容易转换，我在它们上面实验了几个月。当我尝试扩展类型符号，特别是添加结构化（纪录）类型时，问题变得明显。结构看起来，应该以一种直接的方式影射到机器的内存，但一个结构包含一个数组，没有合适的地方隐藏包含数组基地址的指针，也没有方便的方式安排被初始化的对象。例如，早期unix系统的目录条目，在C可以被描述为

struct { int inumber; char name[14]; };

我希望结构不能仅仅是体现抽象对象的特征，也要描述可能从目录读到的位集合。编译器能在哪里隐藏指向语义要求的name的指针呢？即使结构被想象的更抽象，指针的空间也能以某种方式隐藏，我如何处理在分配一个，可能是一个结构包含数组再包含结构到任意深度的复杂对象时，完全初始化这些指针的技术问题？

这个解决方案形成了一个，在无类型BCPL和类型化C之间进化链中的重要飞跃。它移除了指针在内存的具体化。相反促成数组名出现在表达式中时生成指针。C语言的这个规则一直存在至今，就是数组类型的值当出现在表达式中时，被转换成指向组成数组的对象中的第一个对象的指针。

这个发明使现存的B代码能继续工作，不管下层语言语义的改变。仅有几个程序为了调整它的起点，把新值赋给数组名——在B和BCPL中是可能的，在C中无意义——都被很容易修改。更重要的是，新语言保持一致性和对数组有效的（如果并非常见）可解释性，开辟了通往复杂类型结构的道路。

第二个创新，极力明显地把C与它的前辈们区分开来，那就是更完整的类型结构，尤其是在声明语法中的表达式，NB提供基本的类型int和char，它们的数组，指向它们的指针，但没有更进一步的组合。通用化也被要求：给定一个任意类型对象，描述一个包含它们的新对象，从一个函数求得它，或一个指向它的指针，都是可能的。

对每一个此类复合类对象，已经有了一种讨论下层对象的方式：索引数组，调用函数，间接引用指针。类比推理导致了一种名字镜像声明语法，名字特征出现的表达式语法，因此，

int i, \*pi, \*\*ppi;

声明一个整数，一个指向整数的指针，一个指向指向整数的指针的指针。这些声明的语法反映i，\*pi和\*\*pi用于表达式时，都得到一个整数类型。类似地，

int f(), \*f(), (\*f)();

声明一个返回整型值的函数，一个返回整型指针的函数，一个指向返回整型函数的指针；

int \*api[10], (\*pai)[10];

声明一个整型指针数组，一指向整型数组的指针。在所有这些情况中，一个变量的声明类似它在表达式中的用法，它的类型是在声明中，置于开头的那个。

C语言采用类型组合模式归功于Algol 68，尽管它或许没有以Algol追随者认可的模式出现。我从Algol获取的主要概念，是一个基于原子类型的类型结构（包括结构），组合为数组，指针（引用），和函数（过程）。Algol 68关于union和转换的概念的影响，在后来也表现出来。

创造类型系统之后，我认为这些相关的语法，新语言的编译器需要一个新名字。NB看起来不够有自己的特点。我决定延用单字母风格并取名为C，而并没有肯定答复，关于名字是否表示字母表或是BCPL中的字母顺序的问题。

**C初生**

在语言取名之后，其它改变很快在进行，例如引入||和&&操作符。在BCPL和B中，表达式求值依赖于上下文：if和其它条件语句内，把一个表达式的值与零比较，这些语言对与（&）和或（or）运算符会给与特别解释。在普通上下文中，它们进行按位运算，但在这个B语句中

if (e1 & e2) ...

编译器必须对e1求值并且如果它是非零值，对e2求值，并且如果它也是非零，则执行依赖if的语句。在e1和e2内的&和|运算符的求值要求，以此类推。此类真值上下文中的布尔运算符的短路（short-circuit）语义被期望，但是运算符的过度使用，难以解释和运用。在Alan Snyder的建议下，我提出&&和||运算符，以使这种机制更直接。

它们姗姗来迟的出现，说明了C语言中不合适的优先级规则。一个人在B中这样写

if (a==b & c) ...

来检测是否a和a相等并且c是否非零；在这样的条件表达式中，&比==的优先级低就好得多。在从B转变为C时，一个人想在这中表达式中，用&&代替&；为了使这种转换不那么痛苦，我们决定保持&运算符与==有相等的优先级，仅仅把&&的优先级与&的做了细微区分。今天，看起来变动&和==的相对优先级会更好，这样就能简化一个C通用的惯用法：为了测试一个掩码值和另一个值，人们必须这样写

if ((a&mask) == b) ...

那个内层的圆括弧是需要的，但会被容易忘记。

许多其它改变发生在1972-3年，但是最重要的是引入了预处理器，部分是因为Alan Snyder[Snyder 74]的催促，但也是为了承认在BCPL和PL/I中已存在的文件包含机制。它的原始版本极之简单，仅提供文件包含和简单字符串替换：#include和参数化宏#define。之后很快对它进行了扩展，主要是Mike Lesk的工作，后来是John Reiser，合并了宏与参数还有条件编译。预处理原本是做为语言本身的一个辅助手段。确实，这么多年来，它甚至未被调用除非在源文件的开始处包含了一个特别信号。这种看法一直持续，解释了在早期参考手册中，预处理器语法与语言其它部分的不完整整合，和对它的不精确描述。

**可移植性**

在1973年早些时候，现代C的基础部分已经完成。在那年夏天，语言和编译器已足够强壮，以允许我们在PDP-11上用C重写Unix内核。（Thompson已用C的早期版本，作了一个生成系统代码的简略尝试——在结构类型出现之前——1972年时，但是放弃了那次努力。） 也是在这段时期，编译器被转向了其它临近的机器，特别是Honeywell 635和IBM 360/370；因为语言不能独自存在，现代库原型已被开发。特别是Mike Lesk写了一个后来被修改成为C的“标准I/O”例程的“可移植I/O包”[Lesk 72]。在1978年，Brian Kernighan和我出版了C程序设计语言[Kernighan 78]。尽管这本书没有描述一些添加的，在后来变成通用的特性，它充当了语言标准，直到一个正式的标准在十多年后被采纳。尽管我们在本书上关系密切，在工作中有着很明显的区别：Kernighan编写几乎所有的解释性的内容，我负责参考手册中包含的附录，和与Unix系统的接口的那一章。

在1973-1980年间，语言又有了一些发展：类型结构有了unsigned, long, union和枚举类型，并且结构几乎成为第一类对象（缺少一个字面值符号）。在它的环境和其它伴随技术中，也产生了同等重要的发展。用C编写Unix内核，让我们对语言的有用性、效率有足够的信心，我们开始重新编码系统的应用程序和工具，并且随后把其中最有趣的东西移到其它平台。像在[Johnson 78a]中描述的那样，我们发现传播Unix工具的最困难的问题，并不在C语言和新硬件的接口，而是在适应其它操作系统上的现有软件。因此Steve Johnson开始了在pcc上的工作，一个可被容易移植到新机器上去的C编译器[Johnson 78b]，在他，Thompson和我开始把Unix系统移到Interdata 8/32计算机的时候。

语言在这时期的改变，特别是1977年左右，很大程度上集中在可移植性和类型安全的考虑，是为了努力处理我们预见和观察到的，在相当多的代码移到新的Interdate平台产生的问题。C在那时仍表现很强的无类型根源的迹象。例如，在早期语言手册或存在的代码中，指针很少被与整型内存索引区分开来；字符指针的算术特性和无符号整数的相似性，使抵制识别它们的尝试很难。无符号类型被添加进来使无符号算术有效，不让它与指针操作混淆在一起。类似地，早期语言容许了整数与指针间的赋值，但这个实践并未开始得到鼓励；一个类型转换的符号（在Algol 68的例子中被称作“casts”）被发明，以更明显指示类型转换。由于PL/I例子的诱惑，早期C没有强烈绑定结构指针与它们所指向的结构，并且允许程序员用pointer->member而几乎不用考虑指针的类型；这样的表达式未受批判地，被当作到指针代表的内存区的引用，成员名字仅表示一个偏移量和一个类型。

尽管K&R第一版描述了，把C的类型结构带到当前形式的大部分规则，许多用旧式、更松散风格写的程序也被允许，所以编译器也要忍受它们。为了鼓励人们更多注意正式的语言规则，为了发现合法但不可信任的写法，以及帮助发现分离编译中，简单机制没有察觉的不匹配的接口，Steve Johnson改变他的pcc编译器产生了lint[Johnson 79b]，来扫描一批文件并标记可疑的写法。

**在使用中成长**

我们在interdata 8/32的可移植性试验的成功，很快导致了Tom London和John Reiser在DEC VAX 11/780上的另一次成功。这种机器比Interdata变得更流行，并且Unix和C语言在AT&T公司内部和外面都开始快速传播。尽管在1970年代中期，Unix已经被用于贝尔系统和公司外的一小群以研究为目的的工业企业、学院和政府机构的各种项目，它的真正成长是在达到可移植性之后才开始。特别是AT&T的计算机系统部门基于其开发和研究小组的System III和System V版本系统，和加州大学伯克利分校继承自Bell实验室研究组的BSD系列的出现。

在1980年代，C语言的使用广泛传播，并且编译器出现在几乎每一种机器体系结构和操作系统；特别是它变成一种个人计算机上流行的编程工具，包括对这些机器的商业软件制造商和对编程有兴趣的终端用户。在那十年开始时，几乎每一种编译器都是基于Johnson的pcc；到1985年，有了许多成型的独立的编译器产品。

**标准化**

到1982年时，形势很明显，C需要正式的标准化。K&R第一版最近似一个标准，却不再反应真实使用中的语言；尤其是它都没有提及viod和enum类型。但是它预示了通往结构的更新的方法，仅在语言对他们的赋值，将他们传递给函数和从函数接受他们，及将成员名字与包含它们的结构或union严格关联的支持被发表后。尽管AT&T发布的编译器包含了这些修改，大部分编译器供应商未基于pcc的编译器也加入了他们，对语言仍没有完整、权威性的描述。

K&R第一版在很多语言细节上也不够精确，对于pcc这个“参照编译器”来说，它日益显得不切实际；K&R甚至没有很好表达它索要描述的语言，把后续扩展仍到了一边。最后，C在早期项目中的使用受商业和政府合同支配，它意味着一个认可的正式标准是重要的。因此（在M. D. McIlroy的催促下），ANSI于1983年夏天，在CBEMA的领导下建立了X3J11委员会，目的是产生一个C标准。X3J11在1989年末提出了一个他们的报告[ANSI 89]，后来这个标准被ISO接受为ISO/IEC 9899-1990。

一开始，X3J11委员会在语言扩展上采取了谨慎、保守的态度。他们认真对待他们的目标：“完善一个清晰、一致和无二义性的C程序设计语言标准，它规范C通用、现行的定义，以及促进用户程序在不同C语言环境的可移植性。”[ANSI 89] 委员会意识到，仅仅靠发布一个标准并不会改变这个世界。这超出了我的期望。

X3J11只向语言本身引入了一个真正重要的改变：它使用从C++[Stroustrup 86]借鉴的语法，把形式参数类型添加到函数类型签名中。用以前的风格，外部函数是这样声明的：

double sin();

它仅提及sin是一个返回一个double类型（即是double精度的浮点数）值的函数。在新的风格中，这个更好的声明

double sin(double);

使参数类型明显化并鼓励更好的类型检查和适当的转换。即使这个添加，尽管它产生一个明显更好的语言，也引起了困难。委员会有理由认为，简单的不合法的旧式风格函数定义和声明不可行，然而仍然同意新式更好。这种必然的妥协像它本应该的那样好，尽管允许两种形式使语言复杂化，并且可移植软件作者必须应付不符合标准的编译器。

X3J11也引入了一大堆较小的附加和修改，例如，类型限定词const和volatile，和稍微有些不同的类型提升规则。然而，标准化过程没有改变语言的特征。特别是，C标准没有尝试在形式上指定语言语义，所以在一些细微的地方上还可以存在争议；而且，它很好解释了自最初描述以来，在使用中的改变，并且它对与一个它的基本实现是足够精确的。

因此核心C语言经过标准化过程几乎未受损害，并且标准作为一个更好、仔细的条文出现了，而不是一次新发明。更多重要的改变发生在语言的环境中：预处理器和库。预处理器使用与其余部分语言截然不同的惯例执行宏替换。它和编译器的交互从未被很好描述，并且X3J11企图纠正这种情形。这种结果明显好于K&R第一版中的解释；除了变得更加易于理解，它提供像标记串联的操作，以前只在偶尔的实现中可用。

X3J11正确理解了一个完整和仔细的标准C描述，和它在语言本身上的工作一样重要。C语言本身没有提供输入——输出或任何其它与外界的交互，所以依赖一套标准方法。在出版K&R时，C主要是被当作Unix的系统编程语言；尽管我们提供了可被其它操作系统容易转换的库例程例子，Unix的底层支持是被隐含默认的。因此，X3J11委员会花了大量时间来设计和归档一套，对所有符合标准的实现都可用的库例程。

通过标准过程，X3J11委员会的当前活动被限制出版对现存标准进行的解释。然而，由Rex Jaeschke召集的作为NCEG（C数值扩展小组）的一个非正式组织，被正式接受为附属组X3J11.1，他们继续考虑对C的扩展。像这个名字隐含的那样，这些可能的扩展中的许多，是为了是语言在数值上的使用更合适：例如，边界动态决定的多维数组，加入IEEE算术处理方式，及使语言在具有向量和其它高级结构特征的机器上更有效。并非所有这些可能的扩展都是数值相关的；他们添加了一个结构字面值符号。

**后来者**

C和B有一些直接的后代，尽管他们不能与Pascal在产生后代上竞争。很早有发展了一个分支。当Steve Johnson在1972年休假期间访问滑铁卢大学时，他带来了B。它在那儿的Honeywell机器上变得流行。后来产生了Eh和Zed（加拿大人对“B之后是什么？”的答案）。当Johnson在1973年返回贝尔实验室时，让他感到惊慌的是，那个他在加拿大播种的语言，在他回来后在家里得到了发展；甚至他自己的yacc程序已经由Alan Snyder用C重写了。

更多近代的C的后代可能包括并发C[Gehani 89]、对象C[Cox 86]、C\*[Thinking 90]、尤其是C++[Stroustrup 86]。这个语言也被广泛用于各种各样编译器的中间表示（基本上是当作一种可移植汇编语言），对直接后代C++，以及类似Modula 3[Nelson 91]和Eiffel[Meyer 88]的独立语言。

**批评**

C在它那类语言中的两个最具特征的思想是：数组和指针的关系，声明语法模拟表达式语法。它们也列入它最为常受批评的特征中，并且也常成为初学者的绊脚石。历史的偶然和错误，在这两种情形中，更加剧了它的困难。这些其中最重要的是C编译器对类型错误的容忍。上述历史中应该清楚的是，C由无类型语言进化而来。它不是陡然以一个全新的语言，带着它本身的规则，出现在它最早期的用户和开发者的面前；相反我不得不在语言发展的同时，不断地改变现有程序，并且容忍现有代码。（后来，ANSI X3J11委员会标准化C面临同样的问题。）

在1977年或之后的编译器，没有对整数和指针间的赋值，或使用错误类型的对象来引用结构成员产生抱怨。尽管K&R第一版中的语言定义，在处理它的类型规则上是相当（尽管不完整）一致的，那本书承认已存在的编译器不用坚持那些规则。而且，一些为了简化早期过渡设计的规则，给以后带来了混乱。例如，函数声明中的空方括号

int f(a) int a[]; { ... }

是一个活化石，一个NB残余的声明指针的方式；在这个特殊的情况下，a在C中被解释为一个指针。这种表示，部分是应为兼容性而存活下来，部分是它允许程序与于读者进行交流的理由，向f传递一个数组生成的指针，而不是到一个整型的引用的企图。不幸地是，它所起到的迷惑的作用和它带来的提醒一样多。

在K&RC中，为函数调用提供正确类型的参数，是程序员的责任，而现有的编译器不检查类型约定。原始语言没有在函数类型签名包括参数类型，是一个重大，和真正需要X3J11委员会做出大胆和最痛苦革新的缺陷。早期设计在我对技术问题的预防中被解释了，尤其是分离编译原文件交叉检查，和我在对从无类型语言改进到类型语言的含义的不完整理解。在上面提及的lint程序，尝试缓解这个问题：作为lint的功能之一，lint通过扫描一系列源文件，比较调用时的函数参数类型和在他们的中的定义，检查整个程序的一致性。

对于已察觉到的语言的复杂性，产生了一些意外的语法。间接引用运算符，在C中被写作\*，在语法上是一个前缀运算符，正象在BCPL和B中一样。这在简单的表达式中很有效，但是在更复杂的情况，需要括号来指示分析。例如，为了区分对函数返回值的间接引用与通过指针调用函数，分别写作\*fp()和(\*pf)()。表达式使用的风格贯彻到声明，所以这些名字可以声明为

int \*fp(); int (\*pf)();

在更过分但仍现实的例子中，事情变得更糟糕：

int \*(\*pfp)();

是一个到返回一个整型指针的函数的指针。出现了两个结果。最重要的是，C有一个相当丰富的类型描述（比如说跟Pascal比较）集合。比如在C——Algol 68描述对象，语言中声明与表达式一样，很难理解，只是因为对象它们本身很复杂。第二个结果归因于语法细节。声明在C中，应该用一个从内到外的风格来阅读，可能难以领会[Anderson 80]。Sethi[Sethi 81]发现，如果间接引用运算符被当作一个后缀而不是前缀运算符，许多嵌套的声明和表达式会更简单，但这时改变已经太晚了。

不管它的困难，我认为C声明的方式是合理的，并且觉得它很合适；它是一个有用的一致性原则。

C的其它特征，它对数组的处理，在实际中更令人疑惑，尽管它也有实在的优点。尽管指针和数组间的关系并不通常，它还是能被学会的。而且，语言在描述重要概念方面，表现了相当强大的功能，例如，仅仅使用几个基本规则和惯例，向量长度在运行时可变。特别是，字符串处理被采用与其它数组一样的机制，附加上null字符终结一个串的惯例。把C的那个方法与两个同时期的语言Algol 68和Pascal[Jensen 74]比较，是有趣的。数组在Algol 68中有固定边界，或可变的：很多机制被要求，存在语言定义和编译器中，为了提供灵活数组（并非所有编译器都实现了它们）。原始Pascal只有固定尺寸数组和串，这已被证明是受限制的[Kernighan 81]。后来，它是部分固定，尽管最终的语言没有广泛运用。

C把串当作在管理上用标记终结的字符数组。除了用串字面值初始化的特殊规则，串的语义完全包含于更通常的控制数组的规则，结果是语言比一个把串当作独立数据类型的语言，更易于描述和翻译。它的方式产生了一些开销：某些串操作比在其它设计里需要更多开销，因为应用代码或库例程有时必须查找串尾，因为几乎没有可用的内置运算，也因为串的存储边界管理极大转向了用户。而且，C的串处理方法很有效。

另一方面，C对数组的处理，通常（不仅指串）不幸在优化和以后的扩展有隐含意义。指针在C程序里流行，不论是显式声明或由数组形成，意味优化器必须谨慎，并必须使用小心的数据流向技巧才能得到满意的结果。复杂的编译器能理解大多数指针可能改变，但一些重要对于分析仍然显得困难。例如，带有从数组继承而来的指针参数的函数，在向量机器上很难编译生成有效代码，因为确定一个参数指针，没有重迭另一个参数指向的数据几乎不可能，或者外部可访问。更重要的是，C的定义如此明确描述了数组的语义，以致改变或扩展数组为更基本对象，以及允许把它们当作整体操作，变得不能适用当前语言。甚至允许声明和使用动态决定尺寸的多维数组的扩展，也不是完全直接[MacDonald 89] [Ritchie 90]，尽管它们使用C中编写数值库更容易。因此，C通过一个统一和简单的机制，包含了实践中串和数组的最重要使用，但是把问题留给了高效率实现和扩展。

除了上面讨论的，语言和其描述中存在许多更小的不适当地方。也有常见的批评可以提及，但不是逐条表明。其中首要的是语言和它的一般要求的环境，没有为编写很大的系统提供帮助。命名结构只提供了两个主要的级别：“external”（到处可见）和“internal”（在单个过程内）。可见性的一个中间级别（在单个数据和过程文件内）与语言的关系很微弱。所以，没有对模块化的直接支持，项目设计师不得不使用自己的约定。

类似，C本身两种存储期：“automatic”对象当控制流程存在或低于过程时存在，“static”存在于程序的全部执行期。动态分配仅由一个库例程提供，并且管理它们的负担落在程序员一方。C反对自动垃圾回收。

**因此成功？**

C的成功远超出了早期的期望。哪些品质促进它得到广泛使用呢？

Unix本身的成功无疑是最重要的因素；它让这个语言可以被几十万人使用。相反，C在Unix中的使用和随之发生的到各种各样机器的可移植性，自然对系统的成功非常重要。但是，语言进入其它环境，提供了更重要的价值。

尽管某些方面对初学者甚至偶尔对老手都是神秘的，C不失为一个简单和小的语言，可被简单和小的编译器翻译。它的类型和操作充分依据于真实机器，让人们容易理解机器如何工作，并学会生成时间和空间效率程序的惯用法也不困难。同时，语言充分抽象于机器细节，程序可移植性也可以达到。

同样重要的是，C和它的主要库支持，总是保证能存在一个真实环境中。它不是被设计用来孤立验证某一点，或作为一个例子，而是作为一个用来写有用程序的工具；它总是意味着同一个大型操作系统交互，并用来创建更大工具。一个节俭、实际的方法影响了进入C中的事物：它覆盖了多数程序员的基本需要，但不尝试提供太多东西。

最后，不管自它第一次不正式、不完整描述发布以来经受的变化，真实的C语言，像几百万使用许多不同编译器的用户见到的一样，与那些类似流行的语言，比如Pascal和Fortran比较，保持着显著的稳定和一致性。存在许多不同的C的方言——最显著的是那些由更古老K&R和更新的标准C的描述——但基本上，C比其它语言保持着更自由的属性扩展。或许最重要的扩展，是用于处理某些Intel处理器的怪异之处的“far”和“near”指针限定符。尽管可移植性不是C的原始设计中的一个主要目标，它在从最小的个人计算机到最大的超级计算机的不同机器上，编写程序、甚至操作系统方面取得了成功。

C诡异重重、充满缺陷，却取得了极大成功。尽管历史的机缘的却起了帮助，它显然符合系统实现语言的需要，并足够有效以取代汇编语言，仍然足够抽象和在广泛不同环境中流畅描述算法和交互。

**致谢**

值得简要总结在今天的C的发展中，那些直接贡献者们在语言发展史中的脚色。Ken Thompson在11969-70年创造了B；它直接继承自Martin Richards的BCPL。Dennis Ritchie在1971-73年把B转化为C，保持B的大部分语法，但添加了类型和许多其它改变，并编写了第一个编译器。Ritchie, Alan Snyder, Steven C. Johnson, Michael Lesk和Thompson在1972-1977年间贡献了语言设计思想，Johnson的可移植编译器被广泛使用。在这段时期内，库例程集合得到相当发展，感谢这些人们和贝尔实验室其它许多人。在1978年，Brian Kernighan和Ritchie写作了多年被当作语言定义的那本书。在1983年初，ANSI X3J11委员会标准化了C语言。特别值得注意的是委员会官员Jim Brodie, Tom Plum和P. J. Plauger，和后续草案标准编辑Larry Rosler和Dave Prosser的努力。 我感谢Brian Kernighan, Doug McIlroy, Dave Prosser, Peter Nelson, Rob Pike, Ken Thompson和编程语言历史会议负责人员对准备这篇文章的建议。

**参考**

[ANSI 89]

American National Standards Institute, *American National Standard for Information Systems—Programming Language C,* X3.159-1989.

[Anderson 80]

B. Anderson, `Type syntax in the language C: an object lesson in syntactic innovation,' SIGPLAN Notices **15** (3), March, 1980, pp. 21-27.

[Bell 72]

J. R. Bell, `Threaded Code,' C. ACM **16** (6), pp. 370-372.

[Canaday 69]

R. H. Canaday and D. M. Ritchie, `Bell Laboratories BCPL,' AT&T Bell Laboratories internal memorandum, May, 1969.

[Corbato 62]

F. J. Corbato, M. Merwin-Dagget, R. C. Daley, `An Experimental Time-sharing System,' AFIPS Conf. Proc. SJCC, 1962, pp. 335-344.

[Cox 86]

B. J. Cox and A. J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach,* Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.

[Gehani 89]

N. H. Gehani and W. D. Roome, *Concurrent C,* Silicon Press: Summit, NJ, 1989.

[Jensen 74]

K. Jensen and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag: New York, Heidelberg, Berlin. Second Edition, 1974.

[Johnson 73]

S. C. Johnson and B. W. Kernighan, `The Programming Language B,' Comp. Sci. Tech. Report #8, AT&T Bell Laboratories (January 1973).

[Johnson 78a]

S. C. Johnson and D. M. Ritchie, `Portability of C Programs and the UNIX System,' Bell Sys. Tech. J. **57** (6) (part 2), July-Aug, 1978.

[Johnson 78b]

S. C. Johnson, `A Portable Compiler: Theory and Practice,' Proc. 5th ACM POPL Symposium (January 1978).

[Johnson 79a]

S. C. Johnson, `Yet another compiler-compiler,' in *Unix Programmer's Manual,* Seventh Edition, Vol. 2A, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Johnson 79b]

S. C. Johnson, `Lint, a Program Checker,' in *Unix Programmer's Manual,* Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Kernighan 78]

B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.

[Kernighan 81]

B. W. Kernighan, `Why Pascal is not my favorite programming language,' Comp. Sci. Tech. Rep. #100, AT&T Bell Laboratories, 1981.

[**Lesk** 73]

M. E. **Lesk**, `A Portable I/O Package,' AT&T Bell Laboratories internal memorandum ca. 1973.

[MacDonald 89]

T. MacDonald, `Arrays of variable length,' J. C Lang. Trans **1** (3), Dec. 1989, pp. 215-233.

[McClure 65]

R. M. McClure, `TMG—A Syntax Directed Compiler,' Proc. 20th ACM National Conf. (1965), pp. 262-274.

[McIlroy 60]

M. D. McIlroy, `Macro Instruction Extensions of Compiler Languages,' C. ACM **3** (4), pp. 214-220.

[McIlroy 79]

M. D. McIlroy and B. W. Kernighan, eds, *Unix Programmer's Manual,* Seventh Edition, Vol. I, AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Meyer 88]

B. Meyer, *Object-oriented Software Construction,* Prentice-Hall: Englewood Cliffs, NJ, 1988.

[Nelson 91]

G. Nelson, *Systems Programming with Modula-3,* Prentice-Hall: Englewood Cliffs, NJ, 1991.

[Organick 75]

E. I. Organick, *The Multics System: An Examination of its Structure,* MIT Press: Cambridge, Mass., 1975.

[Richards 67]

M. Richards, `The BCPL Reference Manual,' MIT Project MAC Memorandum M-352, July 1967.

[Richards 79]

M. Richards and C. Whitbey-Strevens, *BCPL: The Language and its Compiler,* Cambridge Univ. Press: Cambridge, 1979.

[Ritchie 78]

D. M. Ritchie, `UNIX: A Retrospective,' Bell Sys. Tech. J. **57** (6) (part 2), July-Aug, 1978.

[Ritchie 84]

D. M. Ritchie, `The Evolution of the UNIX Time-sharing System,' AT&T Bell Labs. Tech. J. **63** (8) (part 2), Oct. 1984.

[Ritchie 90]

D. M. Ritchie, `Variable-size arrays in C,' J. C Lang. Trans. **2** (2), Sept. 1990, pp. 81-86.

[Sethi 81]

R. Sethi, `Uniform syntax for type expressions and declarators,' Softw. Prac. and Exp. **11** (6), June 1981, pp. 623-628.

[Snyder 74]

A. Snyder, *A Portable Compiler for the Language C,* MIT: Cambridge, Mass., 1974.

[Stoy 72]

J. E. Stoy and C. Strachey, `OS6—An experimental operating system for a small computer. Part I: General principles and structure,' Comp J. **15**, (Aug. 1972), pp. 117-124.

[Stroustrup 86]

B. Stroustrup, *The C++ Programming Language,* Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.

[Thacker 79]

C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. R. Boggs, `Alto: A Personal Computer,' in *Computer Structures: Principles and Examples,* D. Sieworek, C. G. Bell, A. Newell, McGraw-Hill: New York, 1982.

[Thinking 90]

*C\* Programming Guide,* Thinking Machines Corp.: Cambridge Mass., 1990.

[Thompson 69]

K. Thompson, `Bon—an Interactive Language,' undated AT&T Bell Laboratories internal memorandum (ca. 1969).

[Wijngaarden 75]

A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. Koster, M. Sintzoff, C. Lindsey, L. G. Meertens, R. G. Fisker, `Revised report on the algorithmic language Algol 68,' Acta Informatica **5**, pp. 1-236.

Copyright ? 2003 Lucent Technologies Inc. All rights reserved.