The C Programming Lauguage
2nd Edition

Brian W.Kernighan Dennis M.Ritchie

2015 年 3 月 9 日

目录

前言

在 *The C Programming Language* 在 1978 年出版之后, 计算机已经经历了许多变化. 大型计算机变得得更大, 而个人计算机也拥有了十年前大型机的运算能力. 在这段时间内, C 也有些许变化, 虽然不多, 但是与最初作为 UNIX 操作系统语言相比已经, C 语言已经传播得非常广泛了.

鉴于 C 语言变得越来越流行, 并且在这些年间该语言也发生了一些变化, 以及某些组织开发的编译器, 越来越有必要对语言作出更加精确并且符合现代观点的定义, 与第一版相比. 1983 年, 美国国家标准组织建立了一个委员会, 指在建立一个” 明确的, 独立于机器的 C 语言”, 同时保留它的精髓. 结果就是产生的 C 语言的 ANSI 标准.

标准明确了第一版中提到但没有详细描述的解释, 例如结构与与枚举体赋值. 它提出一种新的函数定义形式, 该形式允许在使用时交叉检验函数的定义. 它指定了一个标准函数库, 字符串操作和一个类似的工作. 它使得原来比较模糊的特点更加地明确, 同时显式地说明要保留语言的机器无关性.

The C Programming Language 第 2 版用 ANSI 标准来描述 C 语言, 对于该语言已经演变的部分, 我们会用一种新格式来写出来. 对于大部分, 这并没有什么本质的区别; 最主要的区别就是函数定义与声明的格式. 现代的编译已经支持标准中的大多数特性.

我们尽量保持与第 1 版一样的简洁性. C 不是一种很复杂的语言, 用一本很厚的书籍来描述它并不合适. 对于 C 语言的重要特性 (例如指针, 它是 C 语言的精髓) 我们会着重阐述. 我们已经增加并修改了许多示例. 例如, 如何对待复杂的声明是由程序添加的, 该程序会把声明转换到字中, 反之亦然. 与以前一样, 所有的例如都被测试过.

附录 A 中的参考手册并不是标准, 但是我们想要在更小的空间中传送标准的重要之处. 这于程序员来说这很好理解, 但对于编译器开发者来说这此手册中的内容并不等同于定义, 定义的责任属于标准. 附录 B 总结了标准库函数. 同样, 该参考只对程序员有意义, 而不针对于函数实现人员. 附件 C 简单地概括了一下自从第一版以来, C 语言所发生的变化.

正如我们在本书第一版前言中所说的那样, ”C wears well as on’s experience with it grows”. 在十多年的实践中, 我们对这名话感受颇深. 我们希望这本书可以帮助你学好并用好 C 语言.

我们非常感谢那些帮助我们写这本书的朋友. Jon Bently, Doug Gwyn, Doug McIlroy, Peter Nelson 和 Rob Pike 对本书的草稿提出了许多宝贵的意见. 我们也很感谢 Al Aho, Dennis Allison, Joe Campbell, G.R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford 和 Chris van Wyk, 以上这些人非常认真地读了这本书. 我们也从以下这些人收到了许多有助的建议, 他们是 Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo 和 Peter Weinberger. Dave Prosser 回到了有关 ANSI 标准的许多细节问题. 我们使用了 Bjarne Strostrup 的 C++ 翻译器测试了我们的程序, 并且 Dave Kristol 向我们提供了一个 ANSI C 编译器来做最终的测试. Rich Drechsler 在打字方面帮助很大.

感谢所有的人.

Brian W.Kernighan

Dennis M. Ritchie

第一版前言

C 语言是一种通用编程语言, 支持表达式, 现代控制流, 数据结构, 以及一系列的操作符. C 语言并不是一种” 非常高级的编程语言”, 它并不” 大”, 以不是专门针对某一应用领域. 由于 C 语言缺少许多限制以及通用性, 使得它与所谓的更厉害的语言相比, 在完成许多任务上更加的方便与高效.

C 语言最开始是为了在 PDP-11 上实现 UNIX 操作系统, 主要是由 Dennis Ritchie 完成的. 操作系统, C 编译器, 所有重要的 UNIX 应用程序 (包括所有的用来写这本书的软件) 都是由 C 开发的. 除此之外, C 还在其他机器上开发了编译器, 包括 IBM System/370, Honeywell 6000 和 Interdate 8/32. C 语言并不特定了某些硬件或系统, 然后, 开发那些将要在不同的, 但是支持 C 语言的机器上运行的程序是很容易的.

这本书指在帮助读者使用 C 语言开发程序. 该书包含一个教程, 能让新手尽快地起步, 根据 C 语言的几个主要特性来划分章节, 另外还有一个参考手册. 读者主要地工作就是阅读, 写代码和回顾, 而不是仅仅拘泥于语法上. 在大多数情况下, 示例程序都完整真实的程序, 而不是一个片断. 所有的例子都被测试过. 除了展示如何高效地使用语言外, 我们也会尽可能地描述一些有用的算法和设计开发风格或原则.

这本书并不是一个介绍编程的手册; 我们假设读者已经拥有一些编程的基本知识, 例如变量, 赋值语句, 循环和函数. 无论如何, 一个初学者应该有能力阅读下去, 向其他人寻求帮助也是一个不错的方法.

在我们的实践中, C 已经展现出了它的优雅, 丰富和多功能. 它很容易学习, 实践越丰富, 它就表现地越好. 我们希望这本书可以帮助你很好地使用它.

许多意见与建议已经添加进了这本书, 而且我们也很乐于这么做. 尤其是 Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy Bill Roome, Bob Rosin 和 Larry Rosler, 他们都非常认真地阅读了资料. 我们同时也感谢 Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, marion harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill plauger, Jerry Spivack, Ken Thompson 和 Perter Weinberger, 他们在不同阶段都提出了宝贵的意见. 同时也感谢 Mile Lesk 和 Joe Ossanna 的帮助与打字.

Brian W. Kernighan

Dennis M. Ritchie

第一章　教程式的介绍

让我们开始一个对 C 的快速介绍. 我们的目标是在真实的程序中展现语言最核心的特点, 但是不会陷入到语言的细节, 规则与异常中. 在这一点上, 我们不会去完整或者精确地描述 (这一点会到正确的例子时再去详述). 我们会尽快让读者到达这样的水平: 可以写出有用的程序, 为了达到这样的水平, 我们要把重点放在基础上: 变量与常量, 算术表达式, 控制流, 函数, 输入和输出的基本原理. 我们计划在这章忽略关于 C 的重要特点, 而这些特点对于写大程序来说是非常重要的. 这些特点包括指针, 结构体, C 的操作符, 几个控制流语句, 以及标准库函数.

这种编排方案有他的缺点. 值得注意的是, 关于任何特性的完整故事并不会出现在这本书中, 教程会尽量简洁, 可能会造成误解. 由于示例程序并没有用到 C 语言的全部力量, 能够用到 C 语言的全部特性的程序不够简洁. 我们会尽量减少由于前而造成的影响, 但是我们会有所提醒. 另外一个缺点是之后的一章我们会对这章的内容有所重复. 我们期望重复会帮助到你, 而不是让你感到厌烦.

在任何情况下, 经验丰富的程序员应该有能力从本章的材料中推断出对他们编程有益的东西. 初学者应该写一些小的, 类似的程序来补充. 两人者都可以使用这些示例作为框架, 通过这些框架来理解描述的细节, 而这些细节会在第二章中提到.

1.1　开始

学习一门新的编程语言的唯一方法就是用它来写程序. 第一个程序对所有编程语言来说都是同一个:

Print the words

```
hello, world
```

这是一个大障碍; 为了跨越这个障碍, 你必须要创建一个程序文本, 成功地编译它, 加载它, 运行它, 并且找到它的输出. 掌握了这此机械化的细节, 其他的事情就相对容易了.

在 C 语言中, 打印 “hello, world” 的程序是

```
#include <stdio.h>

main()
{
    printf(”hello world\n”);
}
```

如何运行这个程序取决于你所用的系统. 作为一个特定的例子, 在 UNIX 系统中在一个文件 (文件名以 “.c” 结尾) 创建一个程序, 例如 hello.c, 然后用下而这个命令来编译它:

```
cc hello.c
```

如果你没有搞糟任何东西, 例如漏了一个字母或是拼写错了什么, 编译过程会很安静地进行, 并且生成一个叫作 a.out 的可执行文件. 如果你输入以下命令来运行程序:

```
a.out
```

它将会输出:

```
hello, world
```

在其他的系统上, 规则可能会有些不同, 具体情况具体分析.

现在, 开始解释一下这个程序. 一个 C 语言程序, 无论它有多大, 都是由*变量*和*函数*组成的. 一个函数包括*语句*, 语句指定了将要被计算的操作, 以及计算期间存值的变量. C 语言的函数类似于 Fortran 中的子程序和函数的概念, 也类似于 Pascal 中的例程和函数. 我们的例子是一个叫作 `main` 的函数. 当然你们也可以给任何你喜欢的名字, 但是 `main` 很特殊 —-你的程序是从 `main` 开始执行的. 这意味着任何一个程序都要有一个 `main` 在某个地方.

`main` 通常会调用其他函数来完成工作, 这些函数有些是你写的, 有些是库函数提供的. 程序中的第一行,

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

告诉编译器去包含关于标准 IO 的库信息; 这一行经常出现在许多的 C 语言源代码中. 标准库函数在第七章和附件 B 中描述.

在函数之间传递数据的一个方法是在调用函数时提供给它一系列的值, 这些值称为 *参数*. 函数名之后的圆括号包裹住了参数列表. 在这个例子中, `main` 被定义为一个没有参数的函数, 这是通过声明一个空列表 `( )` 来实现的.

```
#include <stdio.h>              /* include information about standart */
/* library */
main()                           /* define a function called main
                                    that received no argument values */

{                                /* statements of main are enclosed
                                    in braces */
    printf(”hello world\n”);     /* main calls library function printf
                                    to print this sequence of chacters
                                  \n represents the newline chacters */
}
```

第一个 C 程序

函数的语句被包围在花括号 `{ }` 之中. 函数 `main` 只包含一个语句,

```
printf(”hello, world\n”);
```

函数通过函数名调用, 后跟圆括号括起来的参数列表, 对函数 `printf` 的调用使用的参数是 `”hello, world\n”` . `printf` 是一个库函数, 它的作用是打印输出, 在这里是将一个用双引号括起来的字符串.

被双引号括起来的字符序列, 例如 `”hello, world\n”`, 叫作*字符串*或*字符串常量*. 我们使用字符串的唯一场合是作为 `printf` 或其他函数的参数.

字符串中的序列 `\n` 是 C 语言的一个记号, 叫作*换行符*, 它的作用是在打印完后, 将输出提到下一行的左边空白. 如果不没有输入 `\n` (这值得一试), 你会发现打印完后, 没有换到下一行. 你必须使用 `\n` 在 `printf` 的参数中包含进换行符; 如果你写成了下面这个样子

```
printf(”hello, world
”);
```

C 编译器就会产生一个错误信息.

`printf` 不会自动提供一个换行符, 所以可以通过分阶段地若干次调用来产生一个输出. 我们的第一个程序也可以写成这个样子

```
#include <stdio.h>

main()
{
    printf(”hello, ”);
```

```
    printf(”world”);
    printf(”\n”);
}
```

来产生相同的输出.

注意 `\n` 仅仅代表一个字符. 一个*转义序列*, 比如 `\n`, 提供了一种通用和可扩展的方法, 来表示一些不可输入或不可见的字符. 这些字符包括制表符 `\t`, 退符符 `\b`, 双引号 `”` 和反斜杆 `\\`. 完整的列表在第 2.3 节.

习题 1.1 在你的系统上运行程序 `”hello, world”`, 尝试删除掉程序中的某些代码, 看看编译器会报什么错误.

习题 1.2 实验一下, 当 `printf`'s 的参数包含不同的转义字符 `\`*c* 时 (*c* 是上文中没有列出的字符), 会发生什么.

1.2　变量和数学表达式

下面一个程序利用公式 $°C = (5/9)(°F - 32)$ 来打印下面这张表格, 这张表格包含了温度的摄氏表示及与其对应的华氏表示:

```
1       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

这个程序仍然由一个单独的函数 `main` 构成. 它要比打印 `hello, world` 的程序要长一些, 但并不复杂. 在这里介绍几个新概念, 包括注释, 声明, 变量, 算术表达式, 循环和格式输出.

```
#include <stdio.h>
/*
 * print Fahrenheit-Celsius table
 * for fahr = 0, 20, ..., 300
 */
 main()
 {
    int fahr, celsius;
    int lower, upper, step;

    lower   = 0;    /* lower limit of temperature scale */
    upper   = 300;  /* upper limit */
    step    = 20;   /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr - 32) / 9;
        printf(”%d\t%d\n”, fahr, celsius);
        fahr = fahr + step;
    }
 }
```

这四行

```
/*
 * print Fahrenheit-Celsius table
 * for fahr = 0, 20, ..., 300
 */
```

是*注释*, 在这个情况下是在简短地解释这个程序在做什么. `/*` 与 `*/` 之间的任意都会被编译器忽略; 它们的作用仅仅是让程序更容易地被人理解. 注释可以出现在任何空白, 制表符或换行符可以出现的地方.

在 C 语言中, 所有的变量必须先声明再使用, 声明通常是在函数的开始位置, 先于任何可执行语句之前. 一个*声明*宣告了变量的属性; 它由名字和一连串的变量组成, 例如

```
int fahr, celsius;
int lower, upper, step;
```

类型 `int` 意味着后面列出的变量是整数; 反面是 `float`, 意味着浮点数, 等等, 数字可能含有可分段的部分. `int` 与 `float` 的表示范围取决于你所用的机器: 16 位的 `int` 在 -32768 到 +32767 之间, 这是很常见的, 另外还有 32 位长的 `int`. 一个 `float` 的典型字长为 32 位, 大小范围在 10^{-38} 至 10^{38} 之间.

C 语言在 `int` 与 `float` 之外, 还提供了几种不同的类型, 包括: 这些数据类型的长度同样是跟

`char`	字符, 单字节长
`short`	短整型
`long`	长整型
`double`	双精度浮点

机器相关的. 这里还有由这些基本类型进一步衍生的*数组*, *结构体*和*共用体*, 指向它们的*指针*, 返回它们的*函数*, 在之后的课程中, 我们都会遇到这些.

在温度格式转换程序中, 计算机是由*赋值语句*开始的

```
lower   = 0;
upper   = 300;
step    = 20;
```

这些语句是在给变量赋初值. 单独的语句是以封号结束的.

表格中的每一行都是用同样的方法计算得到的, 所以我们使用一个循环, 每次输出一行; 这是 `while` 循环的目标

```
while (fahr <= upper) {
    ...
}
```

`while` 的工作过程是: 圆括号中的条件被测试. 如果测试结果为真 (`fahr` 的值小于等于 `upper`), 循环体 (花括号中的三行语句) 就会被执行. 然后条件重新被测试, 如果还是真的, 那么循环体再次被执行. 当测试结果为假时 (`fahr` 的值大于 `upper`), 循环就会结束, 接着执行循环体后面的语句. 因为在循环体之后没有更多的语句, 于是程序结束.

`while` 循环体的语句可以有一条, 或者是被花括号包围的多条语句 (例如温度转换程序), 只有一条语句时可以没有花括号, 如下所示

```
while (i < j)
    i = 2 * i;
```

如果可以的话, 我们总是将 `while` 控制下的语句缩进一个制表符的宽度 (在这里显示的是四个空格宽), 这样一眼看过去就可以知道哪些语句是在循环中. 缩进加强了程序的逻辑结构. 虽然 C 编译器关心代码的样子, 但良好的缩进与空格有益于人们读懂程序. 我们推荐每行只写一条语句, 在操作符的两边的使用空格来使分组变得清楚. 空格的位置并不是很重要, 虽然人们对此怀有强烈的信仰. 我们已经从几种比较流行的编程风格中选择了一种, 选一个适合你的, 然后经常用它.

大部分工作在循环体内就可以做完. 通过下面这条语句, 摄氏温度被计算并赋值给变量

```
celsius = 5 * (fahr - 32) / 9;
```

之所以是先乘以 5 再除以 9, 而不是直接乘以 5/9, 是因为在 C 语言中, 整数除法会被*截断*: 小数部分被丢弃. 因为 5 和 9 是整数, 所以 5/9 的值是零, 于是所有计算得到的摄氏度都是零.

这个例子也展现了 printf 是如何工作的更多的一些信息. printf 是一个通用的格式化输出函数, 在第七章我们会详细讨论. 它的第一个参数是一个字符串, 这个字符串将会被打印, 每一个% 指明了它的每一次出现都会被后面的参数替换掉, 而且还指明了打印的格式. 例如 %d 指定了一个整型数, 语句

```
printf(”%d\t%d\n”, fahr, Celsius);
```

导致两个变量 fahr 和 celsius 被打印输出, 在它们之间还插入的一制表符 (\t).

在 printf 中第一个参数中的每一个 % 都对应于第二个参数, 第三个参数, 等等; 它们必须在数值与类型上匹配得当, 否则你将会得到错误的答案.

另外, printf 并不是 C 语言的一部分; 在 C 语言中并没有定义输出与输入. printf 不过是一个标准库函数中的一个有用的函数. printf 的行为在 AMSI 标准中定义, 然而, 正因为它在标准库中定义, 因此在任何一个编译器中, 它的行为总是符合标准的.

为了把注意力集中到 C 语言自身, 在第七章之前我们并不会过多地谈及输入与输出. 特别是, 我们会推迟关于格式化的输入. 如果你想知道如何输入数字, 阅读 7.4 节中关于 scanf 的内容. scanf 类似于 printf, 除了它是读入数据而不是输出数据.

在温度转换程序中有两个问题. 比较简单的一个是问题是输出并不是很好看, 因为数字并不是向右对齐的. 这很容易解决; 如果我们在 printf 语句的每个 %d 添加一个宽度, 那么打印的数字将会是向右对齐的. 例如

```
printf(”%2d, %6d\n”, fahr, celsius_;
```

将会打印第一个数字, 这个数字所占的宽度为 3 个字符, 第二个数字将会占 6 个字符, 就像下面这个样子:

```
  0       -17
 20        -6
 40         4
 60        15
 80        26
100        37
...
```

更重要的问题是, 因为我们使用了整型的数字表达式, 摄氏温度并不是非常精确, $0°F$ 大概是 $-17.8°C$, 而不是 -17. 为了得到更加精确的答案, 我们必须使用浮点型的算术表达式而不是整型. 为此我们必须对程序加以修改. 这是修改后的第二个版本:

```
#include <stdio.h.
/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300; floating-point version */
main()
{
    float   fahr, celsius;
    float   lower, upper, step;

    lower   = 0;    /* lower limit of temperature scale */
    upper   = 300;  /* upper limiet */
    step    = 20;   /* step size */

    fahr = lower;
    while (fahr <= upper) {
        Celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf(”%3.0f %6.1f\n”, fahr, celsius);
        fahr = fahr + step;
    }
}
```

这个版本与之前的相比非常的类似, 除了 fahr 和 celsius 被声明为浮点型, 转换公式的写法更加地自然. 我们在之前的版本中无法使用 5/9, 因为整型之间的除法会造成截断. 常量中的一个小数点表明这是一个浮点数, 因此, 5.0/9.0 不会被截断因为这是两个浮点数之间的比率.

如果一个操作符的操作数是整型的, 那么所执行的操作将会是整型的. 如果一个算术操作符包含了一个浮点操作数和一个整型操作数, 那么那个整型操作数将被转化为浮点数. 如果我们写了 (fahr-32) 这样的语句, 那么 32 将会自动转化为浮点数. 无论如何, 即使一个浮点数带整型常量的值, 那么它也是浮点型的.

在第二章中我们会阐述将整型数转化为浮点数的细节. 就目前来说, 只要注意这样的赋值语句

```
fahr = lower;
```

和条件测试

```
while (fahr <= upper)
```

都会按照最自然的方法工作 —-在操作完成之前, 将 int 转化成 float, printf 的约束 %3.0f 指示一个浮点数 (这里是 fahr), 在打印时至少要占 3 个字符的宽度, 不要打印小数点与小数部分. %6.1f 描述另外一个数 (celsius) 在打印时至少要占 6 个字符的宽度, 并且要带有一位小数. 输出如下

```
  0  -17.8
 20   -6.7
 40    4.4
...
```

宽度和精度可以不用指明: %6f 说明一数至少要占到 6 个字符的宽度; %.2f 规定要带有两位小数, 但是数的宽度并没有约束; %f 仅仅规定以浮点数的格式打印数字.

%d	以十进制数格式打印
%6d	以十进制数格式打印, 至少占 6 个字会宽度
%f	以浮点数格式打印
%6f	以浮点数格式打印, 至少占 6 个字符的宽度
.2f	以浮点数格式打印, 带 2 位小数
6.2f	以浮点数格式打印, 至少占 6 个字符的宽度, 并且带 2 位小数

除此之外, printf 以 %o 指示八进制数, %x 指示十六进制数, %c 指示字符, %s 指示字符串, %% 表示一个百分号.

习题 1.3 修改温度转换程序来打印表头.

习题 1.4 写一个程序, 将摄氏温度转换成华氏温度.

1.3　for 语句

为了编程完成一项任务有多种方法. 现在来看一下温度转换程序的变种.

```
#include <stdio.h>
/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
}
```

这个程序的运行结果与前一个相比一模一样, 但是它看起来有点不同. 最主要的变化是消除了许多变量; 只保留了 fahr, 它的类型为 int. 上界, 下界与步进值只是作为常量出现在 for 语句中, 给了它一个新的构造, 计算摄氏温度的表达式本来是作为一个单独的表达式出现, 但是现在是作为 printf 的第三个参数.

最后提到的变化是一个通用规则的实例 —-在一个可以出现某种类型的值的地方, 都可以放一个该类型的更复杂的表达式. 因为 printf 的第三个参数 %6.1f 必须匹配一个浮点数, 所有一个值为浮点数的表达式可以放在这里.

for 语句是个循环, 比 while 更加一般化. 如果你将它与之前的 while 比较, 那么它的操作就清楚得多了. 在圆括号内分成三个部分, 用封号分开. 第一部分是初始化

```
fahr = 0
```

只执行一次, 在进行循环之前. 第二部分是控制循环的条件测试:

```
fahr <= 300
```

这个条件被计算; 如果值为真, 则执行循环体 (在这里是一条 printf 语句). 步进

```
fahr = fahr + 20
```

被执行, 然后条件被重新评估. 当条件为假时, 循环结束. 跟 while 一样, 循环体可以是单条语句, 也可以是花括号包围的一组语句. 初始化, 条件和步进可以是任何表达式.

选择 while 还是 for 随你的便, 取决于使用哪个更方便. 如果初始化与步进语句是单条语句, 且在逻辑上联系, 则 for 会更加合适, 因为它会比 while 更加的紧凑.

习题 1.5 修改温度转换程序来逆向打印表格, 即从 300 度到 0 度.

1.4　符号常量

在离开温度转换程序之前, 还有最后一步观察. 在程序中写诸如 300 和 20 这样的 “魔数” 是很不好的习惯; 这些数给呆会儿要阅读源代码的人只会传递非常少的信息, 而且他很系统地改变源代码. 一个解决魔数的方法是给他们一个有意义的名字. 一个 #define 命令可以把一个*符号名字*或*符号常量*定义为一串特定的字符串:

第二章　类型, 操作符和表达式