The C Programming Lauguage
2nd Edition

Brian W.Kernighan Dennis M.Ritchie

June 17, 2015

Contents

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

Chapter 1

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2 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, 调用 `printf` 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输出.

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 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机器相关

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

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2 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` 说明一个数至少要占到 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:

`#define` *name replacement list*

然后, *name* 的每一次出现 (*name* 不能在被双引号包围, 也不能是另外一个名字的一部分) 都会被替换为 *replacement list*. *name* 拥有和变量一样的形式: 数字与字母的序列, 并且以字母开头. *replacement text* 可以是任意的字符串, 并不局限于数字.

```
#include <stdio.h>

#define LOWER 0     /* lower limit of table */
#define UPPER 300   /* upper limit */
#define STEP  20    /* step size */

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

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

`LOWER`, `UPPER` 和 `STEP` 的值是符号常量, 不是变量, 所有它们不能出现在声明中. 符号常量通常使用大写字母, 以区别于小写字母的变量. 注意, 在 `#define` 的行尾并没有封号.

1.5 字符输入与输出

现在我们要开始考虑一系列的有关如何处理字符数据的程序. 你将会发现许多程序只不过是我们将要讨论的程序的扩展版本.

标准库函数支持的输入输出模型非常简单. 文件输入或输出, 不管它们是从哪来, 要到哪去, 都会被当作字符流处理. 一个*字符流*是一个字节序列, 它们可以被划分成一行一行; 每一行包含零个或多个字符, 后面再跟上一个换行符. 标准库函数有责任为每一个输入或输出流确认它们的模型; C 程序员无需关心在程序之外行是如何表示的.

标准库函数提供了几个函数用于一次读写一个字符, 其中 `getchar` 与 `putchar` 是最简单的两个. `getchar` 从文本流读入*下一个输入字符*并返回它的值. 也就是说, 当执行完

```
c = getchar();
```

之后, 变量 `c` 就会包含下一个输入字符. 输入字符通常来自键盘; 从文件输入将会在第七章讨论.

每次调用 putchar 都会打印一个字符:

```
putchar(c);
```

将变量 c 的值以字符的格式打印出来, 通常打印在屏幕上. 对 putchar 与 printf 的调用可能是交错进行的, 输出将会按照调用的顺序出现.

1.5.1 文件复制

只要有了 getchar 和 putchar, 你就可以在无需知道更多关于输入与输出的情况下写出一个惊人的程序. 最简单的程序是从输入每次复制一个字符到输出:

read a character
while (*character is not end-of-file indicator*)
    output the character just read
    read a character

翻译成 C 语言就是

```
#include <stdio.h>

/* copy input to output; 1st version */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

关系操作符 != 的意思是 "不相等".

在键盘或屏幕上什么将成为字符是想当然的, 但是就像其他所有的东西一样, 在计算机内部仅仅是二进制位而已. char 是专门用来存储这个字符数据的, 但也可以用整型. 我们会因为某些非常微妙但很重要的原因而使用 int.

程序的关键是从有效的输入数据中识别出输入的结束. 解决办法是当 getchar 读不到更多输入时, 就会返回一个与众不同的值, 这个值不能与其他真正的字符混淆. 这个值就是EOF, 意为"End Of File"(文件结束). 我们必须把c 声明成足够存放任何 getchar 返回的值的类型. 我们不用 char 是因为变量 c 必须足够来存放 EOF, 而不仅仅是 char, 所有我们使用了 int.

EOF 是一个在 <stdio.h> 中声明的整数, 具体是什么值并不重要, 只要跟任何一个字符都不相等即可. 使用符号常量, 使得我们可以保证程序不依赖于特定的数值.

有经验的 C 程序员会把复制程序写得更加的简练. 在 C 语言中, 任何一个赋值语句, 例如

```
c = getchar();
```

都是一个表达式, 并且代表着一个值, 赋值结束后, 这个值就是左边变量的值. 这就意味着赋值语句可以出现在一个更大的表达式中. 如果把对变量c 的赋值放在 while 循环的条件判断中, 复制程序就可以写成

```
#include <stdio.h>
/* copy input to output; 2nd version */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

while 先是得到一个字符, 并将其赋值给 c, 然后测试这个字符是不是文件结束标志. 如果不是文件结束, while 便会进入循环体, 如何循环. 当输入的是文件结束标志时, while 终止, 于是 main 也就结束了.

这个版本的复制程序专注于输入 ----现在只有对 getchar 的一次引用 ----程序也被缩短了. 现在得到的程序更加的紧凑, 一旦掌握了方言, 程序就很容易读懂. 在后面你会经常看到这样的风格. (把代码写得非常难懂是有可能的, 但是我们会有所倾向.

在条件判断中, 赋值表达式两边的圆括号是必须的. != 的优先级 高于=, 这就意味着如果没有了圆括号, 那么条件测试会先于赋值. 所以语句

```
c = getchar() != EOF
```

等价于

```
c = (getchar() != EOF)
```

这样就会产生我们没有预料到的效果: 取决于 getchar 是否读到了文件结束标志, c 的值要么是 0, 要么是 1.(更多的内容请看第2 章.)

习题 1.6 验证表达式 getchar () != EOF 的值是 0 或 1.

习题 1.7 写一个程序来打印EOF 的值.

1.5.2 字符计数

接下来这个程序将会对字符进行计数; 与复制程序有点类似.

```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
    long    nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

语句

```
++nc;
```

向我们展现了一个新的操作符, ++, 意味着增一. 你应该用 ++nc 来代替 nc = nc + 1, 因为前者更加简练与更加高效. 有一个对应的减一的操作符 --. ++ 与 --既可以作用前缀操作符 (++nc), 也可以是后缀操作符 (--nc); 在表达式中这两种形式具有不同的值, 将会在第 2 章演示, 但是 ++nc 与 nc++ 都会使 nc 增一. 在这我们只讨论前缀形式.

字符计数程序在一个 long 类型的变量中累积字符的个数, 而不是 int, long 至少有 32 位长. 虽然在某些机器上, int 与 long 的长度是一样的, 但是在有些机器上 int 的长度是 16 位, 16 位长的整型数最大可表示 32767, 因为使它溢出的数据相对来说较小. 格式说明符 %ld 告诉 printf 对应的参数的类型是 long.

可以用一个 double(双精度浮点数) 类型的变量来支持更大的数. 我们还使用了 for 而不是 while, 以此来说明另一种写循环的方式.

```
#include <stdio.h>
/* count character in input; 2nd version */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

float 与 double 在 printf 中都是用 %f 来说明; %.0f 会阻止打印小数点与小数部分.

for 循环的循环体是空的, 因为所有的工作都在条件测试与递增部分中做好了, 但是 C 语言的语法规则要求 for 语句必须要有一个循环体. 单独的封号, 叫做*空语句*, 只是为了满足语法要求. 我们把它放在一个单独的行中来使它可见.

在我们离开字符计数程序之前, 我们再看一下如果输入中没有字符, while 或 for 在第一次调用 getchar 时就会失败, 于是程序就产生一个零值, 这是正确的答案. 这很重要. while 与 for 的优点之一就是在进入循环体之前会先测试条件. 如果循环条件一开始就为假, 那就什么都不要做, 这甚至意味着根本就没有进入循环体内. 当没有输入时, 程序应该表现的聪明点. while 和 for 有助于保证在边界条件时程序表现合理.

1.5.3 行计数

下面这个程序对输入的行数进行计数. 正如我们之前谈到的, 标准库函数保证一个输入文本流以一个行序列的形式出现, 每一行都是以换行符终止. 因此, 对行计数就是对换行符计数:

```
#include <stdio.h>

/* count lines in input */
main()
{
    int c, nl

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

while 的循环体内组成了一个 if, 它用来控制 ++nl. if 语句对括号内的条件进行判断, 如果结果为真, 则执行随后的语句 (或花括号内的语句块). 我们仍然用缩进来表达控制关系.

== 是 C 语言 "相等" 的记号 (相当于 Pascal 语言中的 =, 或 Fortran 中的 .EQ.). 采取这样的记号是为了与 C 语言中的赋值运算符 = 相区别. 请注意: C 语言新手偶尔会写成 =, 虽然他们心里想的是 ==. 我们将会在第2 章看到, 结果是一个合法的表达式, 我们不会得到警告.

把一个字符写在一对单引号之间, 这表示的是一个整型值, 值的大小等于该字符在机器字符集中的数值. 这叫做一个*字符常量*, 虽然这只是另一个表示一个小整数的方法. 例如, 'A' 是一个字符常量; 在 ASCII 字符集中它的值是 65, 这个整型值是 A 的内部表示. 当然写成 'A' 要比写在 65 要好一点: 这样写意义很明确, 而且独立于特定的字符集.

在字符串常量中的转义序列同样也是合法的字符常量, 所以 \n 代表着换行符的值, 在 ASCII 中它的值为 10. 你必须注意, '\n' 是一个字符, 与字符相对, 我们将在第2 章讨论字符串.

习题 1.8 写一个计算空格, 制表符和换行符的程序.

习题 1.9 写一个程序, 将输入中多于一个空格转换成一个空格, 并输出.

习题 1.10 写一个程序, 将输入中的制表符用 \t 替换, 将退格符用 \b 替换, 将反斜杆用 \\ 替换, 于是, 输入中的制表符与退格符就会以一种无歧义的方式显现出来.

1.5.4 词计数

我们的第四个程序是对行, 字符和单词计数, 单词的比较松散的定义是一个不包含空格, 制表符或换符符的字符序列. 下面这个程序是 UNIX 程序 wc 的主干程序.

```
#include <stdio.h>

#define IN  1   /* inside a word */
#define OUT 0   /* outside a word */
```

```
/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

每次程序碰到单词的第一个字符时, 就会计数一次. 变量 `state` 记录着当前程序是否在一个单词内; 初始状态是 "不在单词内", 于是被赋值为 `OUT`. 相对于字面常量 0 和 1, 我们更喜欢用符号常量 `IN` 和 `OUT`, 因为这样会使程序的可读性更好. 在一个小程序中, 这样做直到的作用很小, 但是在一个大程序中, 从一开始就值得为代码清晰而付出一番努力. 你会发现要想扩展程序只要在符号常量那么修改一下魔数就够了.

这一行:

```
nl = nw = nc = 0;
```

把所有三个变量都赋为零. 这并不是一个很特别的情况, 但是可以从这推论赋值表达式是一个带有值的表达式, 运算顺序是从右到左. 如果我们写成:

```
nl = (nw = (nc = 0));
```

效果是一样的. 操作符 `||` 的意思是 OR, 所以这一行:

```
if (c == ' ' || c == '\n' || c == '\t')
```

是说 "如果 `c` 是空格符或换行符, 或制表符". (回想一下, `\t` 是制表符的可视化表示.) 有一个对应的 AND 操作符 `&&`; 它的优先级要比 `||` 高. 用 `&&` 或 `||` 连接的表达式从左至右运算, 而且只要能够判断出整个表达的真值, 那么判断过程就会马上停止. 如果 `c` 是空格符, 那么就没有必要再去判断它是不是换行符或制表符, 所有后面的这些判断都不会执行. 虽然在这里不太重要, 但是在更加复杂的情况下就非常重要, 我们以后就会看到.

这个例子还演示了 `else` 的使用, 如果 `if` 语句的条件判断为假, 就会去执行 `else` 中的语句. 通常的形式是

`if` (*expression*)
    statement$_1$
`else`
    statement$_2$

`if-else` 有且仅有一个会被执行. 如果 *expression* 判断结果为真, *statement*$_1$ 就会执行; 否则, *statement*$_2$ 执行. 每一个 *statement* 都可以是一条语句或用花括号包围起来的几条语句. 在单词计数程序中, `else` 后面的那个 `if` 控制着花括号内的两条语句.

习题 1.11 你如何测试单词计数程序? 如果程序有 bug 的话, 你应该准备什么样的输入来使它显露出来?

习题 1.12 写一个程序, 从输入中每行打印一个单词.

1.5.5 数组

让我们写一个程序来统计文本中数字, 空白符 (包括空格, 制表符与换行符) 以及其他字符的出现次数. 虽然这是人为构造了, 但是这可以让我们在一个程序中同时阐述 C 语言的某些特征.

输入文本中含有十二种字符, 因为使用数组来存储每个数字的出现次数是很方便的, 而不是用十个单独的变量, 这是程序的其中一个版本:

```
#include <stdio.h>

/* coutn digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c - '0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
}
```

把源代码作为输入, 则程序的输出为

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

声明:

```
int ndigit[10];
```

宣布 `ndigit` 是一个包含 10 个整数的数组. 在 C 语言中, 数组的下标总是从 0 开始, 所有数组中的元素是 `ndigit[0], ndigit[1], ..., ndigit[9]`. 这在初始化与打印数组的 `for` 循环中已经反映了出来.

下标可以是任何值为整数的表达式, 包括整型变量, 如 `i`, 与整型常量.

这个程序依赖于数字字符的属性. 例如, 测试

```
if (c >= '0' && c <= '9')
```

判断 `c` 是否是一个数字字符, 该数字字符的数值是

```
c - '0'
```

这个表达式正常工作的前提是 `'0', '1', ..., '9'` 的数值是连续递增的. 幸运的是, 这个规则对所有字符集都是适用的.

在定义上, `chars` 仅仅是一个小整数, 所以在自术表达式中, `char` 类型的变量与常量, 和 `ints` 是一样的. 这个规则用起来非常方便而且自然; 例如 `c - '0'` 的值是一个整数, 大小在 0 到 9 之间, 对应于存放在 `c` 中的 `'0'` 到 `'9'`, 而且这个值也是数组 `ndigit` 的有效下标. 判断某个字符是否是一个数字, 空白符还是别的什么, 是在下面这几行中做的:

```
if (c >= '0' && c <= '9')
    ++ndigit[c - '0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nother;
else
    ++nother;
```

模式

```
if (condition1 )
    statement1
else if (condition2 )
    statement2
    ...
    ...
else
    statementn
```

在程序中会经常出现, 这是一种做多路选择的方式. *condition* 按照从上到下的顺序求值, 直到某些 *condition* 满足; 这个时候, 对应的 *statement* 开始执行, 于是整个构造就结束了. (任何一个 *statment* 都可以是用花括号括起来的几条语句.) 如果没有一个条件满足, 如果最后有一个 `else`, 那么它后面的语句就会执行. 如果最后没有 `else` 和它的 *statement*, 在这个单词计数程序中, 什么动作也不会执行. 在第一个 `if` 和最后一个 `else` 之间可以有任意数量的

```
else if ( condition )
    statement
```

这种组合.

作为一种既成风格, 建议以我们展现的方式组织这种结构; 如果每一个 `if` 都要在前一个 `else` 的基础上再往右缩进的话, 一个较长的判断就会超出页面的边缘.

将在第 4 章讨论的 `switch` 语句提供了另一种表达多分支的方法, 尤其是条件的值是在某个整数或字符集合中. 作为对照, 我们将在 3.4 节展现一个这个程序的 `switch` 版本.

习题 1.13 写一个程序打印一个柱状图, 这个柱状图表达的是输入中各个单词的长度. 柱状图的水平方向很容易画, 但是在垂直方向稍有挑战性.

习题 1.14 写一个程序打印一个柱状图, 这个柱状图表达的是输入各个字符的数目.

1.5.6 函数

在 C 语言中, 一个函数与 Fortran 中的子程序或函数是等价的, 与 Pascal 中的过程或函数同要如此. 函数提供了一个方便的方法, 来封闭一些计算, 然后就可以使用它而不用考虑它的具体实现. 如果函数设计得合理, 那么就可以忽略工作是如何做的; 只要知道它能做什么就足够了. C 语言把函数的诉求做得非常简单, 方便和高效; 你将会经常看到有些短函数只被定义且只使用一次, 这仅仅是为了让代码更加清晰.

到目前为止, 我们只用了库函数已提供的函数, 例如 `printf`, `getchar` 和 `putchar`, 现在是时候由我们自己来写一些函数了. 因为 C 语言中没有像 Fortran 那样的指数操作符 `**`, 所有就让我们通过编写函数 `power(m, n)` 来阐述函数定义的结构. `power(m, n)` 的作用是计算 m^n 的值, 也就是说 `power(2, 5)` 的值是 32. 这个函数并不是一个实用的指数函数, 因为它只处理了小整数的正整数次幂, 但是这个例子足够用来阐述 C 语言的函数结构. (标准库函数包含一个计算 x^y 的函数 `pow(x, y)`.)

这是函数 `power` 和用来测试它的主程序, 因此你可以一次看到全部的结构:

```
#include <stdio.h>

int power(int m, int n);

/* test power function */
main()
{
    int i;
```

```
    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
    return 0;
}

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

一个函数定义的形式是:

```
return-type function-name(parameter declarations, if any)
{
    declarations
    statements
}
```

函数定义可以以任意的顺序出现, 也可以在一个或几个文件中, 但是同一个函数不能被分割在几个文件中, 必须在一个文件内定义完毕. 如果一个程序的代码分散在几个文件中, 与一个文件相比, 你可以说前者需要更多的编译与载入, 但这是操作系统操心的事, 并不是语言的属性. 在这里, 我们假设两个函数都在同一个文件内, 所以前面你所学的 C 语言知识仍然有用

函数 power 在 main 函数里被调用了两次, 都在一行完成的:

```
printf("%d %d %d\n", i, power(2, i), power(-1, i));
```

每次调用 power 都要传递给它两个参数, 每次调用都会返回一个整数, 这个整数被格式化地打印输出. 表达式 power(2, i) 的值是一个整数, 就像 2 和 i 一样. (并不是所有的函数都返回整数; 我们会在第 4 章讨论地更深一点.)

第一次就是 power 它自己:

```
int power(int base, int n);
```

这一行声明了函数的名字, 返回值类型和参数类型. power 参数是 power 的局部变量, 在其他函数中是不可见的: 这意味着在其他函数中可以使用名字一样的变量而不会发生冲突. 这对变量 i 和 p 也是成立的: power 里面的 i 与 main 里的 i 没有关系.

我们通常会为函数参数列表里的每一个变量名字使用一个*参数*. 术语 *形式参数* 与 *实际参数* 有时候会不加区分.

在 main 中返回 power 的值是通过 return 语句完成的. 在 return 后面可以跟上任何的表达式:

return *expression*

函数可以不需要返回值; 一个返回语句而后面没有跟上表达式, 会使得控制流返回到主调函数, 但不会带回值, 在执行到函数的终止右花括号也会导致返回. 主调函数也可以忽略函数的返回值.

你可以已经注意到在 main 的末尾有一个 return 语句. 因为 main 与其他函数相比没什么两样, 它也会给主调函数返回一个值, 而这个值返回给执行该程序的环境. 典型来说, 返回一个 0 值意味着正常终止; 非 0 值意味着不寻常或错误的终止条件. 为简便叙述, 到这里为止我们都忽略了 main 的 return 语句. 从现在开始我们将会在 main 中包含 return 语句, 作为返回给执行环境的提示信息.

声明:

```
int power(int base, int n);
```

放在 main 的前面是为了说明 power 是一个参数, 它需要两个 int 参数并且它的返回值也是 int. 这个声明, 专业点说是 *函数原型*, 必须与 power 的声明与调用一致. 如果函数的定义或调用与它的函数原型不一致, 都将导致一个错误.

参数的名字可以不用相同. 实际上, 函数原型中的参数名字是可有可无的, 所以函数原型我们也可以写成这个样子:

```
int power(int, int);
```

然面认真选择的名字对一个好文档来说是非常必要的, 所有我们在函数原型中都会写出参数的名字.

版本变化注意事项: 从 C 语言的早先版本到 ANSI C 的最大变量就是函数的定义与声明. 在 C 语言的原始定义中, power 会被写在这个样子:

```
/* power: raise base to n-th power; n >= 0 */
/* (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

参数在贺括号之中命名, 它们的类型在开花括号之前声明, 未被声明的参数会被当成 int. (函数体与之前的一样.)

在程序开头的 power 的声明就像这个样子:

```
int power();
```

没有出现参数列表, 因为编译器无法检查 power 是否被正确的调用. 确实, 因为默认 power 会返回整型值, 因此整个声明也可能被忽略.

新的函数声明语法可以让编译器更加容易地侦测到参数个数或类型的错误. 老的声明风格与定义在 ANSI C 中仍然可以工作, 至少在过渡时期内是这样的, 但是我们强烈建议你使用新的语法, 如果你的编译器支持的话.

习题 1.15 重写 1.2 节中的温度转换程序, 使用一个函数来完成转换工作.

1.5.7 参数 ---- 按值传递

对使用其他语言, 尤其是 Fortran 的程序员来说, C 语言函数中有一个特征是比较新鲜的. 在 C 语言中, 所有函数的参数都是按值传递. 这意味着在被调函数中, 参数的值会存放在一个临时变量中, 而不是它们原先所在的变量. 这跟按引用传递的语言 (例如 Fortran, Pascal) 相比, 有很大的不同. 在那些语言中, 被调用的子例程可以访问原来的参数, 而不是它们的副本.

按值传递是一个很有用的特性, 但不是责任. 这个特性经常可以使带有少量外来变量的程序更加的紧凑, 因为参数在被调函数中可以被当作已被初始化的局部变量来使用. 例如, 这是使用了这个特性的 power 版本:

```
/* power: raise base to n-th power; n >= 0; version 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

n 被当作一个局部变量来使用, 它的值向下递减 (for 向下循环), 直到变为零; 变量 i 变得不再需要. 在 power 无论对 n 做什么操作, 都不会影响到 power 被调用时, 传递给它的参数.

如果有必要的话, 在被调函数中是可以修改主调函数中的变量的. 主调函数必须提供要被改值的变量的地址 (从技术上讲其实是指向变量的指针, 并且被调函数也要把参数声明为指针类型, 通过指针可以间接地访问变量, 我们将在第 5 章讨论指针.

但是对于数组来说就不一样了. 如果数组的名字被当作一个参数的话, 传递给函数的值将是数组首元素的地址 -- 并没有把整个数组的元素传递进去. 通过下标访问元素, 函数可以访问并修改元素中的任意一个元素. 这是下一节将要讨论的话题.

1.5.8 字符数组

C 语言中最常用到的数组是字符数组. 为了阐述字符数组的使用, 以及操作它们的函数, 让我们写一个程序, 这个程序从文本行中计入, 并打印其中最长的一行. 提纲足够简单:

```
while (there's another line)
    if (it's longer that the previous longest )
print longest line
```

使用提纲就可以很自然地把程序分割成块. 一个模块获取一个新行, 另一个存储它, 最后一个控制处理过程.

因为事情被分割地很好, 那么在写程序时按照这个思路来写就容易多了. 首先, 让我们先写一个函数 getline, 这个函数会获取输入中的下一行. 我们努力使这个函数可以适应多种环境. 在最简单的环境下, getline 当有可能读到文件末尾时, 会返回一个信号来告知这件事. 0 是一个可接受的表示文件结束的值, 因为没有一个有效的行的长度是 0. 每行文本都至少有一个字符, 即使是一个只含有换行符的行, 它的长度也是 1.

如果我们找到一个比之前最长的行还要长的, 我们必须把它存放在某个地方. 这引出了第二个函数, copy, 把一个文本行复制到一个安全的地方.

最终, 我们需要一个主程序来控制 getline 和 copy. 这是最终的程序:

```
#include <stdio.h>
#define MAXLINE 1000    /* maximum input line length */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print the longest input line */
main()
{
    int     len;                /* current line length */
    int     max;                /* maximum length seen so far */
    char    line[MAXLINE];      /* current input line */
    char    longest[MAXLINE];   /*longest line saved hear */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0) /* there was a line */
        printf("%s", longest);
    return 0;
}

/* getline: read a line into s, return length */
int getline(char s[], int lim)
{
    int c, i;
```

```
    for (i = 0; i < lim - 1 && (c = getchar()) != EOF && c != '\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: copy 'from' into 'to' assume to is big enough */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

函数 getline 与 copy 在程序的开头声明, 我们假设这两个都在同一个文件中.

main 和 getline 通过一对参数交流, 并返回一个值. 在 getline 中, 参数是用这样声明的:

```
int getline{char s[], int lim);
```

这行代码指明了第一个参数 s 是一个数组, 第二个参数 lim 是一个整数. 在声明数组的时候, 提供数组的大小是为了分配存储空间. 在 getline 函数中, 数组 s 的长度并不是必须的, 因为它的大小在 main 中设置. getline 使用 return 把返回值发送给调用者, 就像 power 做的那样. 这行代码也声明了 getline 返回一个 int; 因为 int 是默认的返回类型, 所以返回值类型可以省略.

有些函数会返回一个有用的值; 而其他, 例如 copy, 仅仅是为了它们的使用效果而用它们, 因此并没有返回值. copy 的返回类型是 void, 这是在显式地声明该函数不会返回任何值.

getline 将字符 '\0'(空字符, 值为零) 放到它所创建的数组的末尾, 以此来标记字符串的结尾. 这个转换同样被 C 语言使用: 当一个字符串常量, 如

```
"hello\n"
```

出现在 C 程序中时, 它被当成一个字符数组存储起来, 在这个数组以 '\0' 终止. printf 的格式说明符 %s

h	e	l	l	o	\n	\0

希望对应的参数是一个字符串, 这个字符串同样以 '\0' 终止. copy 同样也依赖于输入参数以 '\0' 终止, 并将这个字符复制到输入中.

即使是一个很小的程序, 说明它的设计技巧也是很有必要的. 例如, 当 main 遇到一个超过上界的行应该怎么办? getline 工作起来很安全, 因为当数组满的时候它就停止了搜集, 即使没有看到换行符. 通过测试长度和最后一个返回的字符, main 可以判断某一行是不是过长, 然后再按它所希望地那样来处理. 为简洁起见, 我们没有考虑这个问题.

使用 getline 的用户无法提前知道输入的行会有多长, 所以 getline 会检查溢出. 在另一方面, copy 的用户已经知道 (或有办法知道) 一个字符串会有多长, 所以我们决定不要添加错误处理代码.

习题 1.16 修改最长行程序的主要例程, 使得它可以正确地打印任意输入字符串的长度, 尽量对文本也适用.

习题 1.17 写一个程序, 打印出所有长度超过 80 个字符的行.

习题 1.18 写一个程序, 从输入中移除行首与行尾的空白符, 同时也删除空白行.

习题 1.19 写一个函数 reverse(s) 将字符串 s 的顺序逆转. 使用它来写一个程序, 将输入的每一行逆转.

1.5.9 外部变量与作用域

main 中的变量, 例如 line, longest 等等, 是局部的. 因为它们是在 main 中声明的, 没有其他函数可以直接访问到它们. 对其他函数中的变量也是一样的; 例如, getline 中的变量 i 与 copy 中的变量 i 是没有关系的. 函数内的局部变量只有在函数被调用的时候才会存在, 当函数调用结束时, 也就消失了. 所以这些变量才会被称为*自动变量*, 这个词在其他语言中也是一个术语. 我们仍然使用自动变量来引用那些局部变量.(第 4 章讨论 static 关键字修饰的变量, static 修饰的变量在函数调用之间保持不变.)

因为自动变量是伴随着函数的调用与结束, 所以它们无法在调用之间保持值, 在每次使用它们的值之间必须初始化, 否则它们值就是垃圾值.

作为自动量的一个选择, 可以定义一个对所有函数来说都是*外部*的变量, 也就是说, 该变量可以被任何函数按名访问. (这个特点很像 Fortran 中的 COMMON, 或者是 Pascal 中在最外部声明的变量.) 因为外部变量是全局可访问的, 所以它们可以取代函数参数, 而在函数之间通信. 而且, 因为外部变量一直存在, 并不会随着函数的调用与结束而出现与消失, 它们可以保持值, 即使在函数为它们赋值之后即返回.

一个外部变量必须被*定义*, 只能被定义一次, 在任何函数之外定义; 这会它分配一个内存空间. 如果函数想要访问这个变量的话, 必须事先加以*声明*, 这表明了变量的类型. 声明可能是显式的 extern, 也可以通过上下文隐式地声明. 为了使讨论更加的具体, 让我们重写最长行程序, 把 line, longest, max 定义为外部变量. 必须把三个函数的调用, 定义和函数体都修改一下.

```
#include <stdio.h>
#define MAXLINE 1000    /* maximum input line size */

int max;                /* maximum length seen so far */
char line[MAXLINE];     /* current input line */
char longest[MAXLINE];  /* longest line saved here */

int getline(void);
void copy(void);

/* print longest input line; specialized version */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0) /* there was a line */
        printf("%s", longest);
    return 0;
}

/* getline: specialized version */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE - 1
            && (c = getchar()) != EOF
            && c != '\n'; ++i)
        line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
```

```
    return i;
}

/* copy: specialize version */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

在上面的例子中, `main`, `getline` 和 `copy` 函数中的外部变量在程序的开头定义, 定义使得外部变量具有了类型与存储空间. 从语法上来讲, 外部变量的定义与局部变量的定义是一样的. 在一个函数可以用外部变量之前, 必须使得该变量对函数来说是可知的; 声明与之前的一样, 除了关键字 `extern`.

在确定的情况下, `extern` 关键字可以省略. 如果某个外部变量在被函数使用之前已经在源程序中定义了, 那么就没有必要再在函数中用 `extern` 去声明. 所以 `main`, `getline` 和 `copy` 中的 `extern` 是多余的. 实际上, 在本书所有的示例程序中, 外部变量的定义都放在源文件的开头, 所以, 本书中的所有函数都可以省略对外部变量的声明.

如何函数分散在几个源文件中, 某个变量在*文件 1* 中定义, 但在*文件 2* 和*文件 3* 中用到了, 那么在*文件 2* 与*文件 3* 中需要用 `extern` 来声明该变量. 通常来说, 把对函数与变量的外部声明放在一个单独的文件中, 这个文件传统上叫做*头文件*, 这就是在源文件开头用 `#include` 包含的文件. 后缀 `.h` 是头文件的传统后缀. 标准库函数中的函数, 例如, 在 `stdio.h` 这样的头文件中的声明. 更详细的讨论在第 4 章, 库函数的讨论在第 7 章和附录 B.

因为 `getline` 与 `copy` 没有输入参数, 因此建议在声明它们时, 将它们声明成 `getline()` 与 `copy()`. 但是为了兼容以前的 C 编译器, 标准会把空参数列表当作旧风格的声明, 于是把所有的参数列表检查都关闭了; 关键字 `void` 在显示声明空列表时就很必要了. 我们将会在第 4 章详细讨论.

你必须注意到在本节当我们谈到外部变量时, 我们总是非常小心地用*声明*与*定义*, " 定义" 用于指明变量放在哪里, 或者给变量分配一块内存空间. " 声明" 只是指明变量的本质, 但是没有分配内存空间.

顺便说一下, 把变量都声明成 `extern` 似乎是一种简化函数交流的趋势. 当你想要参数列表与变量时, 它们就在那儿, 随时可用. 可是即使你不想要全局变量时, 它们也在那儿. 太过依赖于全局变量是一种很危险的行为, 因为对程序来说, 数据联系是什么样的并不是非常明显, 变量可以在不希望, 甚至不经意间被改变, 后果是程序难以修改. 最长行程序第二版并不如第一版, 部分原因是因为全局变量的使用, 另一部分原因是破坏了这两个程序的能用性, 因为使用到了外部变量.

在这个时候, 我们已经谈到了 C 语言中的一些传统核心. 由于这个程序的帮助, 现在可以在可接受的程序大小内写出一些有用的程序, 如果你为些花了足够长的时间来写这个程序, 那么这是个好的方面. 这些习题比前面谈到的程序要稍微的复杂点.

习题 1.20 写一个程序 `detab`, 这个程序的功能是用恰当数量的空格来替代制表符. 假设一个抽表符表示第 *n* 列, 那么 *n* 应该用一个变量还是宏来表示?

习题 1.21 写一个程序 `entab`, 这个程序是把字符串中的空格用最少数量的制表符来替代. 使用和 `detab` 中相同的制表停止位. 当使用一个制表符或一个空格都可以到达一个制表停止位时, 应该优先用哪一个?

习题 1.22 写一个程序, 来 "折叠" 一个过长的行, 折叠成两行或更多的短行, 折叠位置是在第 *n* 列之前的最后一个非空白符. 确定你的程序在遇到非常长的, 并且在特定列之前没有空白或制表符的行时表现得智能点.

习题 1.23 写一个程序, 将 C 源代码中所有的注释删除. 不要忘记处理被引号包围的字符串和字符常量. C 的注释不嵌套.

习题 1.24 写一个程序, 来检查 C 源代码中最基本的语法错误, 例如未匹配的括号, 中括号和花括号. 不要忘了引号, 包括号单引号与双引号, 转义序列和注释.(如果你想把这个程序写得通用些的话, 挺难的.)

Chapter 2

类型, 操作符和表达式

在一个程序中, 变量和常量是基本的被操作数据对象. 变量的声明列表, 指明了它们的类型, 也可能指明了它们的初始值. 操作符指明了对它们所进行的操作. 表达式将变量和常量联合在一直, 产生一个新的值. 一个对象的类型决定了它所能具有的值, 以及它们所能进行的操作. 这正是这章要讨论的内容.

ANSI 标准对基本类型与表达式作了一些小小的修改与增添. 对所有的整数都可以添加 `signed` 与 `unsigned` 修饰符, 以及无符号常量与十六进制字符常量的记号. 浮点操作可以在单精度范围内完成; 同时也有了双 `long` 来支持扩展精度. 字符串常量可以在编译时连接. 枚举成为了语言的一部分, 对于枚举的正式化经历了很长的一段时间. 一个对象可以用 `const` 修饰, 这个修饰符会阻止变量的值被改变. 对算术类型的自动类型转换支持更多的类型.

2.1 变量名字

虽然我们没有在第 1 章谈到, 但是对变量与符号常量的取名有一些限制. 名字由数字与字母组成; 第一个符号必须是字母, 这里所说的字线包含下划线 `"_"`; 它对帮助理解长变量名很有帮助. 不要让名字以下划线开始, 这是因为以下划线开始的名字常常保留给库函数使用. 大写字母与小写字母是不同的, 所以 `x` 与 `X` 两个不一样的名字. 传统上, 在 C 语言中, 变量用小写字母, 符号常量用大写字母.

在内部名字中, 只有名字的前 31 个字符才是有意义的. 对于函数名与外部变量名, 可能少于 31 个字符, 因为外部名字可能被汇编器和装载器使用, 而语言对此无法控制. 对于外部名字, 标准只对 6 个字符和一个单一的大小写保证唯一性. 关键词, 例如 `if`, `else`, `int`, `float` 等等, 是被保留给语言的, 用户不能使用它们给变量或函数命名. 它们必须是小写的.

在对变量进行取名时, 根据变量的用途来命名是很好的做法, 而且在印刷上也不容易混淆. 我们对局部变量使用短名字, 特别是循环下标, 对外部变量使用长名字.

2.2 数据类型和大小

在 C 语言中只有几种基本类型: `char` 单字节, 在本地字符集上只能存储一个字符 `int` 一个整数, 典型大小是机器的字长 `float` 单精度浮点型 `double` 双精度浮点型另外, 还有很修饰符可以应用到这些基本类型上. `short` 和 `long` 修饰整数:

```
short int   sh;
long int    counter;
```

在上面的定义中, `int` 可以省略, 如果你本来就想写 `int` 的话.

`short` 与 `long` 拥有不同的长度; `int` 通常是机器的字长. `short` 通常是 16 位, `int` 既可以是 16 位也可以是 32 位. 编译器可以自由选择类型的长度, 唯一的要求是 `short` 与 `int` 至少有 16 位, `long` 至少

32 位, `short` 不能比 `int` 长, 同时 `int` 也不能比 `long` 长.

既定符 `signed` 与 `unsigned` 可以应用到 `char` 或其他整型. `unsigned` 类型的数字总是大于等于 0, 但最大不能超过或等于 2^n, n 是变量类型的位长. 例如, 因为 `char` 是 8 位, 因此 `unsigned char` 的范围在 0 到 255 之间, 而 `signed char` 的范围在 -128 到 127 之间 (在一个以二进制补码存数的机器上). `char` 是 `signed` 还是 `unsigned`, 这取决于机器, 但是可打印的字符总是正的.

`long double` 声明了一个扩展精度浮点数. 同整数一样, 浮点数的字长依赖于机器的实现; `float`, `double` 和 `long double` 可以表示三种不一样的长字.

标准头文件 `<limits.h>` 和 `<float.h>` 为所有类型长度定义了符号常量, 还包括一些与编译器和机器有关的性质. 这在附录 B 讨论.

习题 2.1 写一个程序, 确定 `char`, `short`, `int` 和 `long` 的表示范围, 包括 `signed` 和 `unsigned`, 可以通过打印标准头文件中的常量与直接计算得到. 确定浮点型变量的表示范围, 如果你是通过计算来完成的话, 这会更加难一点.

2.3 常量

一个整型常量, 如 `1234`, 类型是 `int`. 一个 `long` 的常量要在末尾添加 `l` 或 `L`, 例如 `12345678L`; 如果一个整型常量过大而无法放在 `int` 的话, 它就会被当成 `long`. 无符号整数要在末尾添加 `u` 或 `U`, 后缀 `ul` 或 `UL` 表示 `unsigned long`.

浮点常量包含一个小数点 (`123.4`) 或指数 (`1e-2`), 也有可能两者都有; 它们的默认为 `double`, 除非显式指明. 后缀 `f` 或 `F` 表示一个 `float` 常量; `l` 或 `L` 表示 `long double`.

一个整数的值可以用八进制或十六进制表示, 而不限于十进制. 以 `0` 开始的数字表示八进制数; 以 `0x` 或 `0X` 开始的数字表示十六进制. 十进制数 `31` 可以写成八进制数 `037`, 与十六进制数 `0x1f` 或 `0x1F`. 八进制数与十六进制数常量末尾也可以跟一个 `L`, 使得它们成为 `long`, 或者 `U` 使它们成为 `unsigned`; `0XFUL` 表示一个类型为 `unsigned long`, 值为 15(十进制) 的常量.

一个字符常量是一个整数, 用一对单引号括起来的字来表示, 例如 `'x'`. 一个字符常量的数值大小是机器字符集中对应字符的值. 例如, 在 ASCII 字符集中字符常量 `'0'` 的值为 48, 与数字 0 并没有关系. 如果我们用 `'0'` 而不是 48 来表示字符 `'0'` 的值, 那么程序就更加容易阅读, 也独立于特定的字符集. 字符常量可以像整数一样参与数学运算, 虽然它们在字符比较中用得最频繁.

特定的字符可以用转义序列来表示, 就像 `\n` 表示换行符; 虽然它们看起来像是两个字符, 但其实只表示一个. 另外, 任意一个字节宽度的位模式可以用

```
'\ooo'
```

来指定. *ooo* 表示 1 到 3 个八进制数 (0 ... 7), 或者

```
'\xhh'
```

hh 表示一个或两个十六进制数 (0 ... 9, a ... f, A ... F). 所以我们可以这样写

```
#define VTAB    '\013'  /* ASCII vertical tab */
#define BELL    '\007'  /* ASCII bell character */
```

或者是十六进制:

```
#define VTAB    '\xb'   /* ASCII vertical tab */
#define BELL    '\x7'   /* ASCII bell character */
```

完整的转义序列有:

\a	alert(bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

字符常量 \0 表示值为 0 的字符, 空字符. 为了强调某些表达式的字符属性, 我们会使用 \0, 而不是 0, 虽然空字符的值就是 0.

一个**常量表达式**只牵涉到常量. 这种表达式可以在编译时求值, 而不是运行时. 常量表达式可以出现在常量可以出现的任何地方, 例如:

```
#define MAXLINE 1000
char line[MAXLINE + 1];
```

或者

```
#define LEAP 1 /* in leap years */
int days[31 + 28 + LEAP + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31];
```

一个**字符串常量**, 或者简称**字面字符串**, 是一个被双引号包围的 0 个或多个字符组成的序列, 例如

```
"I am a string"
```

或者

```
""  /* 空字符串 */
```

双引号并不是字符串的组成部分, 仅仅是起分隔作用. 在字符中使用的转义充列同要可以出现在字符串中; \" 表示双引号. 字符串常量可以在编译时连接:

```
"hello, " "world"
```

等价于

```
"hello, world"
```

分割长字符串时非常用用.

从技术上讲, 一个字符串常量是一个字符数组. 在字符串的内部表示中, 在末尾有一个空字符 '\0', 所以字符串的实际存在空间要比出现在双引号中的字符数量要多一个字节. 这种表示法意味着对字符串的长度没有限制, 但是程序为了确定字符串的长度, 需要从头遍历到尾才行. 标准库函数 strlen(s) 返回字符串参数 s 的长度, 不包括末尾的'\0'. 这是我们自己的版本:

```
/* strlen: return length of s */
int strlen(char s[])
{
    int i;

    while (s[i] != '\0')
        ++i;
    return i;
}
```

strlen 和其他字符串函数在标准头文件 <string.h> 中声明.

注意分别字符常量与只包含一个字符的字符串的区别: 'x' 与 "x" 是不一样的. 前者是一个整数, 用于产生字母 *x* 在机器字符集的数值. 后者是一个字符数组, 这个字符数组包含一个字母 *x* 与 '\0'.

还有一种常量叫作**枚举常量**. 一个枚举是一个整型常量列表, 例如:

```
enum boolean { NO, YES };
```

enum 的第一个名字的值是 0, 第二个是 1, 以些类推, 除非显式指明. 如果不是所有的值都有显式声明, 那么未显式声明的名字会在最后一个显式声明的基础上继续递增, 正如第二个例子:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAG = '\v', RETURN = '\r' };
enum moths { JAN = 1, FEB, MAR, APR, MAY, JUM,
             JUL, AUG, SEP, OCT, NOV, DEC };
             /* FEB = 2, MAR = 3, etc. */
```

不同的枚举类型中是不能出现相同的名字. 在同一个枚举类型中也不能出现相同的名字.

枚举类型提供了一个非常方便的方法, 使得常量与名字联系起来, 这是对 #define 的一种候选方案, 优点是你可以为你为自己产生值. 虽然 enum 类型的变量可能需要声明, 但是编译器不需要检查你往这个变量里存的是不是一个有效的值. 无论如何, 枚举类型的变量提供了检查的机会, 大多数情况下要比 #define 要好. 另外, 调试器可以用枚举类型的符号名来打印枚举变量的值.

2.4 声明

所有的变量在使用前都必需声明, 虽然有些声明可以通过隐式的上下文来做. 一个声明指明了类型, 以及一个或多个此类型的变量, 例如

```
int lower, upper, step;
char c, line[1000];
```

变量可以以任何风格分布; 上面的声明也可以写成

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

后面这种形式占据了更多的空间, 但是这种形式容易为每一种变量添加声明, 为了方便后面作修改.

一个变量也可以在声明的时候初始化. 如果在变量名的后面跟着一个等号和一个表达式, 表达式提供了一个初始值, 例如:

```
char esc = '\\';
int  i = 0;
int  limit = MAXLINE + 1;
float eps = 1.0e-5;
```

如果变量不是自动类型的, 那么初始化只做一次, 在概念上要早于程序的运行, 初始值必需是一个常量. 一个自动类型的变量, 在每次进入函数体或语句块内时, 就会显式的初始化一次; 初始值可以是任意的表达式. 外部与静态变量默认初始化为 0. 未初始化的自动变量的值是未定义的 (即垃圾值).

限定符 const 可以应用到任何变量的声明中, 这个限定符用来说明变量的值是不可变的. 对于一个数组来说, const 限定符将会使得数组中的每个元素都是不可变的.

```
const double e = 2.71828;
const char msg[] = "warning: ";
```

const 也可以用在数组参数中, 指明函数不能修改数组:

```
int strlen(const char[]);
```

若尝试修改一个 const 类型的变量, 那么结果是依赖于实现的.

2.5 数学运算符

二元的数学运算符有 +, −, *, / 以及取模运算符 %. 整数的除法会将小数部分截断. 表达式:

```
x % y
```

的值是 x 除 y 的余数, 当 x 能够被 y 整除时, 余数为 0. 例如, 判断某一年是否是润年的方法是, 该年可以被 4 带队, 但不能被 100 整除, 或者可以被 400 整除, 因此条件表达式为

```
if ((year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

操作符 % 不能运用在 float 与 double 类型的变量上, 整除时截断的方向, 以及当取模运行符 % 运用在负数上时, 余数的符号依赖于特定机器的实现, 这是溢出或下溢出时采取的动作.

二元运算符 + 与 − 拥有相同的优先级, 比 *, /, % 的优先级低, 而它们的优先级又比一元运算符 + 与 − 的优先级低. 数学运算符从左到右计算. 章末的表 2.1 总结了所有运算符符的优先级与结合性.

2.6 关系与逻辑运算符

关系运算符包括:

```
>  >=  <  <=
```

它们的优先级都相同. 优先级刚好比它们低的是相等性运算符:

```
==  !=
```

关系运算符的优先级低于算术运算符, 表达式 `i < lim-1` 会被当作 `i < (lim-1)`, 这正是人们所期望的.

比较有意思的是逻辑运算符 && 与 ||. 通过 && 与 || 连接的表达式的值由左向右计算, 一旦可以判断出整个逻辑表达式的真值, 求值过程随即停止. 大部分 C 程序都会依赖这些性质. 下面这个循环取自我们在第 1 章写过的 getline 函数:

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

在读取新字符之前, 需要检查一下数组中是否还有空间存储新字符, 所以条件 `i < lim-1` 要先判断. 更进一步, 如果条件判断结果为假, 我们不能再继续下去读字符.

类似的, 我们不能在调用 getchar 之前测试变量 c 是否等于 EOF; 相反, 我们需要在测试之前调用函数并给变量 c 赋值.

&& 的优先级高于 ||, 但两者低于关系与相等运算符, 所以像这样的表达式:

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

不需要额外的括号. 但由于 != 的优先级高于赋值符, 表达式

```
(c=getchar()) != '\n'
```

中的括号是必需的: 先给 c 赋值, 再将它的值与换行符作比较.

根据定义, 如果关系为值, 那么关系或逻辑表达式的值为 1, 否则为 0.

一元取反运算符 ! 将一个非零值的操作数转化为 0, 一个零值的操作数转化为非 0. ! 的一个比较常见的操作是

```
if (!valid)
```

而不是

```
if (valid == 0)
```

很难判断两种形式哪种更好. 像 !valid 这样的构造读起来非常顺口 ("if not valid"), 但是稍微复杂一点的将会很难理解.

习题 2.2 不要使用 && 与 ||, 写一个写上面的 for 循环等价的一个循环.

2.7 类型转换

当两个不同类型的操作数进行运算时, 根据少量的规则, 它们会转换成一种共同的类型. 一般来说, 唯一自动进行的转换是将较窄宽度的操作数转换为较宽的类型, 这种转换不会丢失精度, 在表达式 `f + i` 中, 将整型数转换为浮点数. 无意义的表达式, 例如将浮点数当作下标访问数组元素, 这是不允许的. 将一个较宽的类型转换成一个较窄类型时, 例如将一个长整型转换成短整型, 将一个浮点类型转换成一个整型, 将会丢失精度, 这会产生一个警告, 但不是非法的.

一个 `char` 是一个小整数, 所以 `char` 类型的变量或常数可以出现在任何需要整数的表达式中. 这就具有了很大的灵活性, 在字符类型转化时.

附录 A

2.8 介绍

这个手册描述的 C 语言指的是在 1988 年, 10 月 31 日提交给 ASNI 的草稿所指定的 C 语言, 由 "美国信息交换标准委员会" 批准. 这个手册是标准的翻译, 而不是标准本身, 虽然作者已经尽力让这个手册成为语言的可靠的指导.

在大多数情况下, 这个手册遵循标准的整体提纲, 而这个标准主要参考了本书的第一版, 虽然 ANSI 对语言作了些许修改. 除了对某些产品的重新命名, 以及没有正式阐述词汇记号与预处理的正式定义, 这里给出的语法与标准是等价的.