

第一部分 C 语言初步

第 1 章 初识 C 语言····· 3

- 1.1 什么是程序 4
- 1.2 编写 C 程序需要什么 5
- 1.3 编程过程····· 6
- 1.4 使用 C 语言·· 7

第 2 章 从何处入手·· 11

- 2.1 概述····· 12
- 2.2 main()函数····· 13
- 2.3 数据类型·· 14

2.3.1 字符····· 14

2.3.2 数字····· 16

- 2.4 小结····· 18

第 3 章 代码注释····· 21

- 3.1 给代码加注释····· 22
- 3.2 注释详述·· 23
- 3.3 留白艺术·· 25
- 3.4 另一种注释方式·· 26
- 3.5 小结····· 27

第 4 章 输出····· 29

- 4.1 printf()做了什么·· 30
- 4.2 printf()的格式····· 30
- 4.3 打印字符串····· 31
- 4.4 转义序列·· 32
- 4.5 转化字符·· 33
- 4.6 小结····· 35

第 5 章 变量····· 39

- 5.1 变量类型·· 40
- 5.2 变量的命名····· 41
- 5.3 定义变量·· 41
- 5.4 在变量中存储数据·· 42
- 5.5 小结····· 44

第 6 章 字符串·· 47

- 6.1 字符串结束符····· 48
- 6.2 字符串的长度····· 49
- 6.3 字符数组：字符的列表····· 49
- 6.4 初始化字符串····· 52
- 6.5 小结····· 53

第 7 章 #include 和#define····· 55

- 7.1 包含文件·· 56
- 7.2 在哪里放置#include 指令·· 58
- 7.3 定义常量·· 59
- 7.4 小结····· 61

第 8 章	输入	63
8.1	简述 scanf()	64
8.2	与 printf() 一起使用	65
8.3	使用 scanf() 的问题	65
8.4	小结	68
第 9 章	C 怎么做数学运算	71
9.1	基本知识	72
9.2	运算符的优先级	74
9.3	用括号打破规则	75
9.4	多重赋值	75
9.5	小结	77
第二部分	操作空间	
第 10 章	表达式还能用来做什么	81
10.1	复合赋值	82
10.2	小心优先级	84
10.3	强制类型转换	85
10.4	小结	86
第 11 章	关系运算符	89
11.1	测试数据	90
11.2	使用 if 语句	91
11.3	否则.....: 用 else 语句	92
11.4	小结	94
第 12 章	逻辑运算符	97
12.1	获取逻辑	98
12.2	逻辑运算符的优先级	103
12.3	小结	104
第 13 章	更高级的运算符	107
13.1	条件运算符	108
13.2	运算符 ++ 和 --	110
13.3	运算符 sizeof()	111
13.4	小结	112
第三部分	保持控制	
第 14 章	循环	117
14.1	while 循环	118
14.2	使用 while 语句	119
14.3	使用 do-while 语句	120
14.4	小结	122
第 15 章	其他循环	125
15.1	为了重复	126
15.2	使用 for 循环	128
15.3	小结	130
第 16 章	终止循环	133
16.1	使用 break 语句	134
16.2	使用 continue 语句	135

16.3	小结	137
第 17 章	测试多个值	141
17.1	使用 switch 语句	142
17.2	switch 语句中使用 break 语句	144
17.3	妙用 case 语句	144
17.4	小结	145
第 18 章	输入和输出	149
18.1	putchar()和 getchar()函数	150
18.2	关于换行符的思考	152
18.3	getch()函数	153
18.4	小结	155
第 19 章	更多有关字符串的内容	157
19.1	字符测试函数	158
19.2	大小写测试函数	158
19.3	大小写转换函数	159
19.4	字符串函数	159
19.5	小结	162
第 20 章	更高级的数学函数	165
20.1	简单的数学函数	166
20.2	更多的转化	167
20.3	三角函数和对数函数	168
20.4	获取随机数	169
20.5	小结	171
第四部分	C 程序和大量数据	
第 21 章	C 语言如何处理列表	175
21.1	复习数组	176
21.2	数组赋值	179
21.3	有关数组的更多内容	180
21.4	小结	180
第 22 章	搜索数据	183
22.1	填充数组	184
22.2	搜索	185
22.3	小结	188
第 23 章	排序	191
23.1	排序	192
23.2	加快搜索	196
23.3	小结	199
第 24 章	我的名字叫指针	203
24.1	内存地址	204
24.2	定义指针变量	205
24.3	使用取值运算符*	207
24.4	小结	208
第 25 章	数组和指针有什么不同	211
25.1	数组名是指针	212

25.2	在数组中取值	213
25.3	字符和指针	214
25.4	小心字符串的长度	214
25.5	指针数组	216
25.6	小结	217
第 26 章	有效管理内存空间	221
26.1	使用堆	222
26.2	为什么需要堆	223
26.3	分配堆	224
26.4	堆内存分配失败	227
26.5	释放堆内存	227
26.6	多次分配	228
26.7	小结	230
第 27 章	结构体	233
27.1	定义结构体	234
27.2	在结构体变量中存放数据	238
27.3	小结	240
第五部分	用函数组织程序	
第 28 章	把数据存到硬盘上	245
28.1	硬盘文件	246
28.2	打开顺序文件	247
28.3	访问顺序文件	248
28.4	小结	251
第 29 章	另一种保存文件的方式	255
29.1	打开随机文件	256
29.2	访问随机文件	257
29.3	小结	260
第 30 章	用函数来组织程序	263
30.1	用 C 函数来组织程序	264
30.2	局部变量和全局变量	267
30.3	小结	269
第 31 章	在函数间共享数据	273
31.1	传递参数	274
31.2	传递实参的方法	274
31.2.1	按值传递	275
31.2.2	按地址传递	276
31.3	小结	280
第 32 章	让函数更完美	283
32.1	返回值	284
32.2	返回的数据类型	286
32.3	声明函数原型	287
32.4	结语	289
32.5	小结	289
附录 A	你可以飞得更高	291

虽然总是有些人认为学习和使用 C 语言很困难，但你很快就会发现他们错了。C 语言被认为是一种神秘的编程语言，而它确实也可以变成这样。但是风格良好的 C 程序与用其他语言写出的程序一样易于理解。现在社会对 C 程序员的需求很旺盛，而且这种需求似乎永不衰竭。

如果你从未写过程序，本章会从最基本的知识开始，教你入门的编程概念，解释什么是程序，并简要介绍 C 语言的历史。你很兴奋吧？C 语言可是个功能强大的编程语言。

1.1 什么是程序

电脑并不聪明。在你智商最低的时候，你的智力也超出电脑几光年。电脑的唯一优势是它遵守你的指令。电脑可以连续几天地处理你提供的数据，既不会感到厌烦也不会要加班费。

电脑不能自己决定做什么。电脑不能独立思考，所以程序员（programmer，告诉电脑要做什么的人）必须给电脑极其详细的指令。不发指令，电脑就毫无用处。若没有详细的指令，电脑无法处理你的工资表，正如汽车不能自己发动并在街上行驶一样。为了让电脑执行某项具体任务而提供给它的详细指令集合就是程序（program）。

说明 字处理程序、电脑工资表系统、电脑游戏和电子数据表都只不过是电脑程序。没有这些程序，电脑只能呆在那里，不知所措。比如，字处理程序就包含了一系列详细的指令，它们用计算机语言（如 C 语言）写成，能准确地告诉电脑怎样处理文字。编程就是让电脑执行程序中提供的指令。

你可以为电脑购买成千上万个程序，但是若公司想让电脑执行某项特定的任务，就要雇用程序员来编写符合要求的程序。你可以让电脑做很多事情，但却有可能找不到一个完全满足需求的程序。本书将把你从这种困境中拯救出来。学习了 C 语言之后，你就可以编写程序了，程序中包含的就是告诉电脑如何运转的指令。

线索 电脑程序告诉电脑如何执行你想要的操作。就像厨师需要食谱才能做成一道菜一样，程序需要指令来产生结果（见图 1-1）。食谱就如同一系列详细的指令，写得好的话，就能清楚描述完成一道菜所需步骤的正确顺序和做法。程序之于电脑也是如此。

当运行（run）或执行（execute）程序时，会产生输出（output）。做好的菜是食谱的输出，而工资表或字处理程序是运行程序产生的输出。

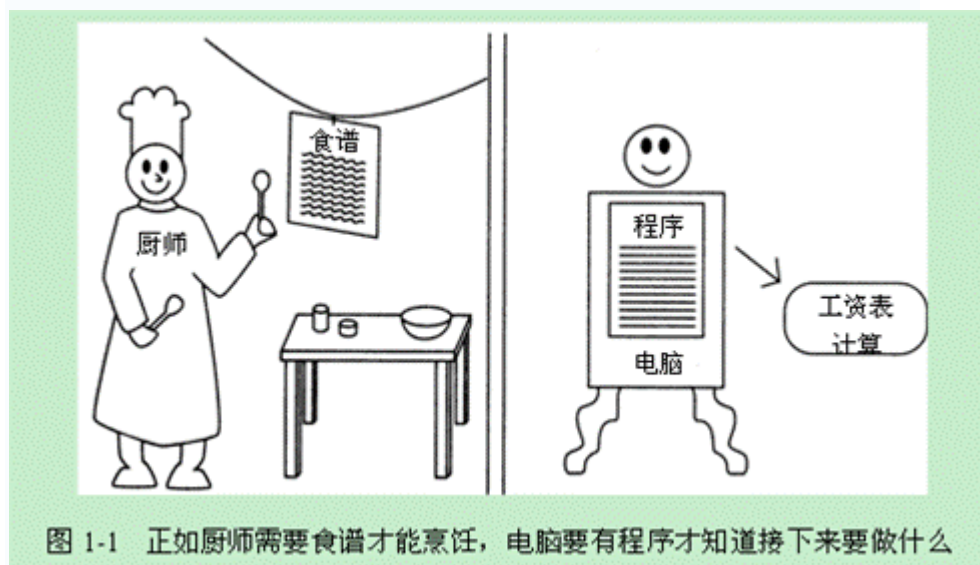


图 1-1 正如厨师需要食谱才能烹饪，电脑要有程序才知道接下来要做什么

1.2 编写 C 程序需要什么

安装了 C 编译器 (C compiler) 之后, 才能够编写并执行 C 程序。C 编译器获取你编写的 C 程序并编译 (compile, 是一个技术术语, 指把程序转换为电脑可识别的内容), 它会在你准备查看结果的时候运行编译后的程序。如今的 C 编译器比几年前的高级许多。它们提供了全屏编辑、下拉菜单以及在线帮助, 为初级程序员提供了许多的帮助。

现在几乎找不到单独的 C 编译器了。大多数情况下, C 编译器捆绑 (bundled, 也是个术语, 即包括) 在 C 的一个更高版本 C++ 中。因此, 当你去买 C 编译器时, 几乎总会发现 C 和 C++ 编译器组合在一起。买下组合的 C++/C 软件包, 你就拥有了 C 编译器, 甚至为日后学习 C++ 也做好了准备。

说明 本书最后一章简要描述了 C++ 与 C 的不同之处, 以及它对 C 的改进。

如今最流行的 C 编译器是 *Turbo C++* 和 *Borland C++*, 二者都是 Borland 公司[①]开发的。*Borland C++* 在 *Turbo C++* 的基础上增加了许多额外的程序, 供高级的 C 和 C++ 程序员使用。

微软公司提供了强大的 C++ 和 C 编译器 *Visual C++*。像 *Borland C++* 一样, *Visual C++* 提供了高级的 DOS、Windows 和 Windows NT 编程。也可以用 *Turbo C++* 编写 Windows 程序, 但 *Turbo C++* 没有像 *Borland C++* 和 *Visual C++* 那样附加很多的 Windows 编程工具。

顺便提一下, 编写 Windows 程序非常难, 尤其对于编程初学者来说。别想一口吃个胖子。首先学会编写简单的基于 DOS 的 C 程序 (本书就够用了), 然后可以逐渐迁移到更难的 C++ 和 Windows 编程。

市场上还有其他的 C 编译器厂商, 但是 Borland 公司和微软公司的 C 编译器在 C 程序员中最受欢迎。

警告 你编写的 C 程序叫做源代码 (source code)。编译器获取 C 源代码并将其翻译为机器语言 (machine language)。电脑是由成千上万个开 (on) 或关 (off) 的电路开关组成的。因此, 电脑最终接收的指令必须是二进制 (binary) 形式。前缀 *bi* 的意思是二 (two), 而电路的两种状态被称为二进制状态 (binary state)。用 C 编译器来把你的 C 程序转换为 1 和 0 (表示内部的开或关) 自然要比你自己亲手愚公移山要容易得多。

1.3 编程过程

大部分人写程序时按照下面的基本步骤进行。

(1) 确定程序要做什么。

(2) 使用一种编辑器 (editor) 编写并保存编程语言指令。编辑器类似于字处理器 (尽管没有那么花哨), 可以创建和编辑文本。所有流行的 C 编译器都包含一个集成在内的编辑器和编程语言编译器。所有的 C 程序文件名都以 .C 扩展名结束。

(3) 编译程序。

(4) 检查程序错误。如果存在错误, 改正它们并返回步骤 3。

(5) 执行程序。

说明 电脑程序中的错误称为 bug。修正错误称为调试 (debugging) 程序。

如今的 C 编译器, 如 *Turbo C++*, 使你可以很容易地执行这 5 个步骤, 所有步骤都在同一环境下完成。例如, 如果安装了 *Turbo C++*, 就可以使用 *Turbo C++* 的编辑器, 编译程序、查看并修改错误、运行程序以及查看结果, 所有这些都发生在同一屏幕下, 使用统一的菜单。请跳过这里, 这有点儿难。如果你从未编过程序, 那么上述这些可能听不太明白。不必紧张。现在的大多数 C 编译器带有方便的教程, 告诉你怎样使用编译器的编辑器和编译命令。

简而言之, 编程时大多数 C 编译器只需要你做这些: 启动编译器, 键入程序, 然后选择菜单中的 Run。Turbo C++ 有快捷键 Alt+R (按住 Alt 键并按下 R 键, 然后一起松开), 之后回车编译并运行程序。编译器负责程序的编译和执行, 遇到错误时它会通知你。

线索 许多时候, C 编译器可以找出程序中的 bug。例如, 如果你拼错了一条命令, C 编译器会在编译程序时通知你。

万一现在你还没有完全明白编译器的作用，我打个比方。可以将源代码看成电脑需要的原始材料，编译器就像一台机器，负责把原材料转化成最终产品，即电脑可理解的编译后的程序。

1.4 使用 C 语言

C 语言是现今最流行的一种编程语言。由于 C 语言的版本多种多样，ANSI 委员会为所有版本开发了一套规则（ANSI C）。只要你使用 ANSI C 编译器运行程序，那么可以确保能在几乎所有装有 ANSI C 编译器的电脑上编译此 C 程序。只要设置恰当，你可以使大多数的编译器（包括 Borland 公司的和微软公司的）与 ANSI C 兼容。（请参阅编译器手册。）

线索 只要你编译了 C 程序，就可以在任何一台与你的电脑兼容的电脑上运行编译后的程序，而不管此电脑上是否有 ANSI C 编译器。

C 语言比大多数编程语言更高效。它也是一种相对较小的编程语言。换句话说，你不必学习太多的 C 语言命令（command）。通过学习本书你将了解到 C 语言的命令和其他元素，如运算符、函数和预处理器指令等。

警告 仔细考虑一下并把思想状态转移到 C 语言上，因为下一章将开始你的第一个 C 程序之旅。

奖励

I 获取 C 编译器并在你的电脑上安装它。大多数编译器带有快速教程，能帮助你将编译器装载到电脑硬盘上。

I 学习 C 编程语言。这正是本书的目的！当你逐步熟悉 C 语言知识后，注意要用 ANSI C 命令，不要用某些编译器专用的 C 函数，因为这些函数在其他编译器上可能不存在。

陷阱

不要紧张，因为 C 语言很容易编写，并且通常 C 编译器有很多特性你不必知道。

第 2 章 从何处入手

本章中，你将看到第一个 C 程序！不要试图理解这里讨论的 C 程序的每一个字符。放松些，先熟悉一下 C 的外观。马上你就要开始认识 C 程序中常见的元素了。

2.1 概述

本节向你展示一个简短但完整的 C 程序，并讨论附录 B 中出现的另一个程序。两个程序中包含了相同元素，也有不同的元素。第一个程序相当简单，如下所示：

```
/* Prints a message on the screen */
#include <stdio.h>
main()
{
    printf("This C stuff is easy!\n");
    return 0;
}
```

如果你在 C 编译器的编辑器中键入这段程序，编译程序并运行它，将会在屏幕上看到这条消息：

```
This C stuff is easy!
```

说明 看上去要做很多工作才能输出这条消息！实际上，在这 7 行代码中，只有以 `printf` 开头的那一行产生了输出，其他行提供的只是大部分 C 程序共有的例行内容。

线索 真正的长程序可以看附录 B。尽管附录 B 中 21 点游戏用了好几页纸，但是它也包含了你刚看到的这个小程序里的共同的元素。

浏览刚才讨论的两个程序，找找它们的相似点。你注意到的第一个相似处可能是大括号 {}、圆括号 () 和反斜杠 \ 的使用。向 C 编译器键入程序时要小心。C 语言很挑剔，它不能容忍在需要圆括号时键入方括号 []。

说明 C 语言并非对每样东西都很挑剔。例如，在 C 程序中的大多数空格是为了让人更易读，而不是让编译器更易读。在程序中加上空行和代码缩进可以帮助改善程序的可读性，使你更容易查找。

C 要求所有命令和预定义函数使用小写字母(2.2 节将学习什么是函数)。只有在 `#define` 行和打印的消息中才使用大写字母。

2.2 main()函数

C 程序中最重要的是 `main()` 函数。前面讨论的两个程序都有 `main()` 函数。`main()` 是一个 C 函数 (function) 而不是 C 命令，虽然现在这个区分不是很重要，但你还是应该知道这一点。函数是指 C 自带的或者由你编写的执行某些任务的例行程序。C 程序由至少一个函数构成。每个程序必须总是包含一个 `main()` 函数。函数名后面带有括号，可以由此与命令相区别。下面这些是函数：

```
main    calcIt()  printf()   strlen()
```

而下面这些是命令：

```
return  while   int      if      float
```

在其他 C 编程的书或手册中，作者可能会省略函数名后面的括号。例如，你可能会看到 `printf` 函数而不是 `printf()`。你很快就会识别函数名，所以这种区别不会给你带来太大影响。大多数时候，作者都会想尽可能地区分函数和非函数，所以你看到的更多是带括号的函数。

警告 刚才列出的一个函数 `calcIt()` 含有一个大写字母。然而，2.1 节说过在 C 语言中，所有命令和预定义函数应该使用小写字母。如果一个名称有多个部分，如 `doReportPrint()`，通常把每个分隔词以大写字母开头，这样可以提高可读性（函数名中不允许出现空格）。不能把单词全部用大写，为了可读性偶尔用一个大写字母还是可以的。

线索 必需的 `main()` 函数和所有 C 语言提供的函数名必须使用小写字母。自己写的函数可以用大写字母，但是大多数 C 程序员遵循使用小写字母函数名的惯例。

正如我们总是从一本书的第 1 章开始阅读，电脑也总是从 `main()` 开始运行程序。就算 `main()` 在程序中不是排在第一个函数的位置，`main()` 仍然决定着程序执行的开始。因此，在你编写的每个程序中都要让 `main()` 函数排在第一个。后面几章中的程序都只有一个 `main()` 函数。等你具备了一些 C 编程知识后，就会知道还可以在 `main()` 函数后面添加其他函数，提高程序的运行能力。

在单词 `main()` 后面，总会看见左大括号 `{`。当你看到一个对应的右大括号 `}` 时，`main()` 就结束了。`main()` 函数内部也可能会有其他的大括号对。作为练习，可以再看一下附录 B 中的长程序。`main()` 是第一个带有代码的函数，还有其他几个函数跟在后面，每个都有大括号和代码。

说明 几乎每个 C 程序都需要语句 `#include <stdio.h>`。它用来打印和获取数据。目前，总是把这条语句放在 `main()` 前面。在第 7 章中，你将明白为什么 `#include` 很重要。

2.3 数据类型

C 程序中的数据必须由数字、单词和字符构成。程序将数据处理成有意义的信息。虽然有很多不同类型的数据，但是以下 3 种数据类型是目前在 C 编程中最常用的：

- I 字符
- I 整数
- I 浮点数（也称实数）

请跳过这里，这有点儿难。你也许会埋怨：“我得学习多少数学知识？我觉得不太划算！”不用担心，因为 C 语言为你解决了数学问题。你不必先学会 2 加 2 才写 C 程序。但是，你必须理解数据类型，这样才能在程序中正确使用它们。

2.3.1 字符

C 语言中的字符 (character) 是电脑能表示的任意一个字符。电脑能识别 256 个不同的字符, 每个字符都能在 ASCII 表中找到, 请参阅附录 C。电脑能表示的任何东西都能作为字符。下面列出的全部都能看成是字符。

A a 4 % Q ! + =]

线索 空格键也会产生一个字符。就像 C 语言需要记录下字母表的字母、数字和所有其他字符一样, 它也必须记录下程序所需的任何空格。

可以看到, 每个字母、数字和空格对 C 语言来说都是字符。当然, 4 似乎是一个数字, 是的, 有时候它是数字, 但有时候也可以当作字符。如果你指明某个 4 是字符, 那么就不能用它进行数学计算; 如果你指明另一个 4 是数字, 就可以用这个 4 进行数学运算。对特殊符号来说也是一样的。加号 (+) 是一个字符, 但它也可以执行加法运算。不好意思, 说着说着又谈到数学方面了。

C 语言的所有字符数据都括在撇号 ' 中。一些人把撇号称为单引号。撇号把字符数据与其他类型的数据区分开来, 如数字和数学符号。例如, 在 C 程序中, 下面所列的都是字符数据:

'A' 'a' '4' '%' ' ' '\'

下面所列的都不是字符数据, 因为它们都没有带撇号:

A a 4 % -

线索 下面所列的都不是有效字符。只有单个字符可以放在撇号中, 多个字符不可以。

'C is fun' 'C is hard' 'I should be sailing!'

本章的第一个程序包含字符 '\n'。一开始, 你可能认为 \n 不是一个字符, 实际上, 它是少数的几个两字符组合, C 语言将这些组合解释成单个字符。稍后会详细介绍这一点。

如果你需要指定多个字符 (除了将要学习的特殊字符外, 还有比如刚才描述的 \n), 则需要把字符放在引号 (") 中。多个字符的组合称为字符串 (string)。下面是一个 C 语言字符串:

"C is fun to learn."

说明 关于字符和字符串知道这些就够了。本书稍后将介绍如何在程序中使用它们。当你读到如何在变量中存储字符时, 就会明白撇号和引号为何如此重要了。

2.3.2 数字

你以前可能没想到过, 数字有许多不同的大小和形状。不论数字什么样子, C 程序必须有一种存储数字的方式。你必须把数字存储在数值变量中。在查看变量之前, 检查一下数字的类型会对你有所帮助。

整数是指没有小数的完整的数。整数没有小数点。(记住这条规则: 就像国会多数议员说话没有要点一样, 整数没有小数点。) 任何没有小数点的数都是整数。下面所列的都是整数:

10 54 0 -121 -68 752

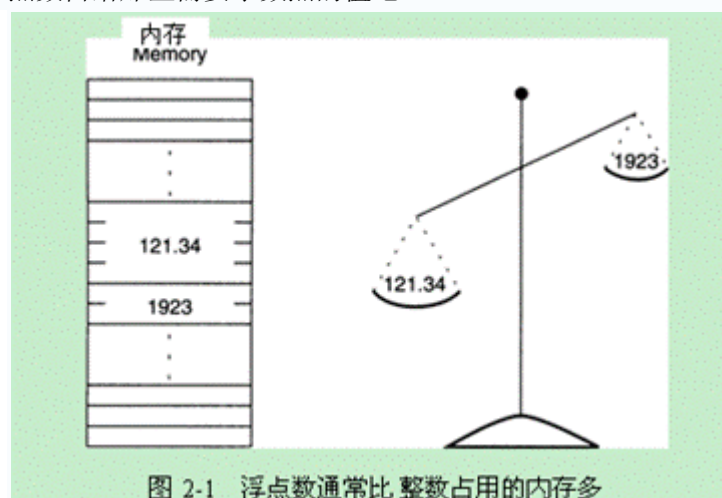
警告 不要让整数以 0 开头 (除非数字是 0), 否则 C 语言会认为你输入的数字是 16 进制的或 8 进制的。16 进制和 8 进制有时也称为以 16 为基和以 8 为基, 它们都是表示数字的奇特的方式。053 是一个 8 进制数, 0x45 是一个 16 进制数。如果你不明白这些, 现在只需要记住如果在整数前加上 0, C 语言就会打开 16/8 进制的开关。

带小数点的数字称为浮点数 (float point number)。下面所列的都是浮点数:

547.43 0.0 0.44384 9.1923 -168.470 .22

线索 如你所见, 浮点数允许以 0 开头。

选择使用整数还是浮点数取决于程序要处理的数据。某些值（如年龄和数量）可以用整数，而某些值（如钱的数量）则需要用浮点数。在内部，C 语言存储整数与存储浮点数的方式不同。如图 2-1 所示，浮点数占用的内存通常是整数的两倍。[②]因此，如果可能请尽量使用整数，把浮点数留给那些需要小数点的值吧。



说明 图 2-1 展示了无论存储的值是大还是小，整数通常比浮点数占用的内存少[③]。不管什么时候，大的邮政信箱都有可能会比小信箱收到的邮件少。信箱的容量并不会影响信箱存放的东西。C 语言的数字存储内存的大小不会受到数值的影响，但会受到数字类型的影响。

不同的 C 编译器为整数和浮点数分配不同大小的内存。随后你将会看到，有几种办法可以查明 C 编译器为每种类型的数据到底分配了多少内存。

奖励

- l 让大小写锁定键处于关闭状态！大多数 C 命令和函数要求使用小写字母。
- l 在 C 程序中加入空格使其更具可读性。
- l C 函数在名字后面必须有括号。C 程序由一个或多个函数构成。总需要有一个 `main()` 函数。C 语言在执行其他函数之前要执行 `main()`。
- l 如果使用字符，要用单引号引起来。字符串要在引号中。整数不带小数点。浮点数带有小数点。

陷阱

- l 输入代码的时候一定不要马虎。当 C 语言需要某个特定的字符比如单括号时，就一定不能错写为方括号。
- l 不要让整数以 0 开头，除非整数是 0 本身。

2.4 小结

本章主要使你熟悉 C 程序的大体样式，主要是熟悉包含了可执行的 C 语句的 `main()` 函数。如你所见，C 语言格式比较自由，对空格的使用并没有过分讲究。但是，C 语言很讲究用小写字母，它要求所有的命令和函数用小写字母拼写，如 `printf()`。

现在，请不必担心在本章中看到的代码的细枝末节。本书的后续部分会解释所有这些细节。

1. 代码范例

```

/* Prints a character and some numbers */
#include <stdio.h>
main()
{
    printf("A letter grade of %c\n", 'B');
    printf("A test score of %d\n", 87);
    printf("A class average of %.1f\n", 85.9);
    return 0;
}

```

2. 代码分析

这个短程序只是在屏幕上打印了 3 行消息。每行消息中包含了本章提到的 3 种数据类型中的一种，即字符（B）、整数（87）和浮点数（85.9）。

程序中唯一由程序员写的函数是 `main()` 函数。左右大括号总是把 `main()` 的代码括起来，而任何你添加到程序中的其他函数也都是这样。你将会看到另一个函数 `printf()`，它是内置的 C 函数，用于产生输出。下面是程序的输出：

```

A letter grade of B
A test score of 87
A class average of 85.9

```

第 3 章 代码注释

电脑必须能够理解你的程序。因为电脑是不会说话的机器，所以必须小心地拼写完全正确的 C 命令，并按照你想要的执行顺序输入。然而，其他人也会读你的程序。另外，你会经常修改程序，并且如果你是为一家公司写程序，公司的需求也会经常改变。因此，必须确保你的程序能被他人和电脑理解。因此，你应该为程序写文档，解释它们做了什么。

3.1 给代码加注释

整个 C 程序从头至尾都应加上注释（comment）。注释是散布于程序中的解释性信息。如果你写程序来计算工资表，那么程序的注释就会解释薪水总额、国税、地税、社保费等所有数字是怎么计算出来的。

说明 如果你写的程序只是你自己使用，就不需要注释了，对吗？不是的。C 语言是一种晦涩的编程语言，即使程序是你自己写的，但是过后你很可能也会看不懂它。

线索 在写程序的同时要加上注释。现在就要养成这个习惯，因为程序员很少在程序写好后回头添加注释。等到必须修改程序时，就会后悔没有早给程序加注释。

在写程序的同时加注释还有另一个好处。写程序时，经常需要查看前面写的语句。你不必重新去理解所写过的 C 代码，可以通过浏览注释很快地找到要看的代码部分。如果没有加注释，那么就不得不译解每一段 C 代码。

程序维护（maintenance）是随时间流逝不断修改程序的过程，它会修正隐藏的错误，并调整程序以适应环境的改变。如果你为某家公司写了一个工资表程序，这家公司可能某一天会改变计算薪水的方式，于是你（或别的程序员）就得修改工资表程序，与公司的新制度保持一致。注释可以方便程序的维护。有了注释，就可以快速浏览程序并找出需要修改的地方。

注释不是 C 的命令。C 语言会忽略程序中的所有注释。注释是给人看的，而注释之间的程序语句是给电脑看的（参见图 3-1）。

考虑下面的 C 语句：

```

return ((s1 < s2) ? s1 : s2);

```



图 3-1 注释给人看，而 C 程序语句给电脑看

你还没掌握 C 语言，然而，即使你已经掌握了，这条语句还是要费些脑筋才能理解。下面这样写不是更好吗？

```
return ((s1 < s2) ? s1 : s2); /* Gets the smaller of 2 values */
```

3.2 节将解释注释的语法，但现在，你明白在/*和*/之间的信息是注释。

注释要口语化且有别于 C 代码。不要只是为了注释而写注释。下面语句的注释是毫无用处的：

```
printf("Payroll"); /* Prints the word "Payroll" */
```

警告 你还没掌握 C 语言，不过你仍然不需要上面的注释行！冗余的注释是在浪费时间，不会给程序带来任何好处。要为可能需要读你的程序的人（包括你自己）加注释，解释程序在做什么。

3.2 注释详述

C 语言的注释以/*开头并以*/结束。注释可以在程序中跨越好几行，并且可以放置在程序的任何地方。下面的每一行都包含了注释：

```
/* This is a comment that happens to span two lines
before coming to an end */

/* This is a single-line comment */

for (i = 0; i < 25; i++) /* Counts from 0 to 24 */
```

说明 注释可以在编程语句之前或之后独立成行。位置的选择取决于注释的长度和注释所描述的代码量。

附录 B 中的 21 点游戏程序包含了所有类型的注释。通过通读程序中的注释，你不需要看 C 代码就能知道程序做了什么。

不要对每一行加注释，通常每隔几行才需要注释。许多程序员喜欢在一段代码前放一段多行的注释，然后在需要的行中插入短一些的注释。下面是一个带有各种注释的完整程序：

```

/* Written by: Perilous Perry, finished on April 9, 1492 */
/* Filename: AVG.C */
/* Computes the average of three class grades */
#include <stdio.h>
main()
{
    float gr1, gr2, gr3; /* Variables to hold grades */
    float avg;           /* Variable to hold average */
    /* Asks for each student's grade */
    printf("What grade did the first student get? ");
    scanf(" %f", &gr1);
    printf("What grade did the second student get? ");
    scanf(" %f", &gr2);
    printf("What grade did the third student get? ");
    scanf(" %f", &gr3);

    avg = (gr1 + gr2 + gr3) / 3.0; /* Computes average */
    printf("\nThe student average is %.2f", avg);
    return 0; /* Goes back to DOS */
}

```

许多公司要求程序员在其所写程序顶部的注释中加入自己的名字。如果将来需要修改程序，可以找最初的程序员来帮忙。在程序的开始处写上它在硬盘上保存的文件名也是一个好习惯，这样当你拿到打印出来的程序时可以在硬盘上找到它。

说明 这本书在有些地方可能注释得太多了，尤其是在开始的几章里。你还不熟悉 C 语言，每个细小的解释都有很大的帮助。

请跳过这里，这有点儿难 测试代码时，你可能会发现用/*和*/注释掉一段代码很有用。这样做能让 C 语言忽略这段代码，于是就可以全神贯注于正在处理的代码。但是，不要注释掉一段已经含有注释的代码，因为注释不能嵌套。C 语言遇到第一个*/就认为是注释已结束，而当 C 语言发现下一个*/前面没有表示开始的/*时，就会报错。

3.3 留白艺术

空白 (white space) 指很多程序中存在的空格和空行。在某种程度上，空白在提高程序的可读性方面比注释更重要。人们在看 C 程序时需要有空白，程序代码不能都挤在一起。看一看下面的程序：

```

#include <stdio.h>
main(){float s,t;printf("How much do you make? ");scanf(" %f",
&s);t=.33*s;printf("You owe %.2f in taxes.",t);return 0;}

```

对编译器来说，这是一个相当好的 C 程序，但对正在看程序的人来说就不是了，尽管代码很简单并且不难理解。下面的程序即使没有注释，也要容易理解得多：

```

#include <stdio.h>
main()
{
    float s, t;

    printf("How much do you make? ");
    scanf(" %f", &s);

    t = .33 * s;
    printf("You owe %.2f in taxes.", t);
    return 0;
}

```

这个代码清单与前一个是一样的，只是它包含了空白，而前一个没有。程序的实际长度不能决定其可读性，空白的数量对程序的可读性来说才是决定性的（当然，加上一些注释也能改善程序，但是这个练习是想说明有没有空白会有多么大的区别）。

说明 你可能想知道为什么在挤在一起的那个程序的第一行，即带有#include的行，在右尖括号之后没有写代码。毕竟，如果在它后面再加上代码，程序的可读性就更差。我差点儿想那么干！许多 C 编译器不允许在#include（或者任何其他以#开头的语句）后面加上

代码。某些 C 编译器甚至不允许在这样的行尾加注释，尽管现在的 C 编译器一般都允许在那里加注释了。

3.4 另一种注释方式

现在许多 C 编译器支持一种原本为 C++ 程序设计的注释，即以两个斜杠 (//) 开头且只在行尾结束。

下面是新的注释方式：

```
// Short program! ,
#include <stdio.h>
main()
{
    printf("Looking good!"); // A message
    return 0;
}
```

不过，本书采用的是 /* 和 */ 形式的注释方式。

奖励

- | 编程的 3 条规则是注释、注释、再注释，尽可能多地使用注释。
- | 使用注释时，以 /* 开始，以 */ 结束。
- | 新的注释方式是以 // 开始注释行的。但是，这种注释还没有被 ANSI C 认可。

陷阱

- | 不要使用冗余的注释。没有价值的注释，只会浪费你宝贵的编程时间。
- | 不要嵌套注释。如果你要注释掉一段程序，必须确保这段代码不含有注释。
- | 不要写几乎没有空白的程序。需要时尽可能多地在程序中使用缩进和空行，对代码行进行分组。当你学习了 C 语言的更多知识之后，你将学会在哪里加空白能提高程序的可读性。

3.5 小结

你必须为程序加注释，它们不是给电脑看的，是给人看的。尽管 C 程序有点晦涩，但注释可以为你消除许多混乱。注释用来说明 C 代码正在做什么。在 /* 和 */ 之间的任何东西都是 C 语言的注释。C 语言编译执行时会忽略所有的注释，因为它知道注释是给人看的。

除了注释，在程序中加入许多空白能使程序更具可读性。如果程序都挤在一起而没有空行和有用的缩进，当你日后回来研究和修改代码时，就会觉得像在读一本不分段落的书。日后一旦要修改程序，注释可以使程序更易于维护，而足够多的空白能帮助你节省时间和精力。

1. 代码范例

下面是两行没有注释的代码：

```
scanf(" %d", &a);
yrs = (a >= 21) ? 0 : 21 - a;
```

下面给这两行代码加上了注释：

```
scanf(" %d", &a); /* Gets the user's age */
yrs = (a >= 21) ? 0 : 21 - a; /* Calculates the number of */
                             /* years until adulthood */
```

2. 代码分析

从这几行可以看出，C 程序中的代码要做什么并不总是很明显。注释用简易的口头语解释了代码的行为。C 程序中并非每一行都需要注释，但是多加些注释确实能更清楚地阐明代码的行为。

如果任何人（包括你自己）都看不到程序的输出，那么程序就没有什么用了。所以最后，你必须能够查看程序的输出结果。C 语言用于输出的主要方法是使用 printf() 函数。没有其他的命令执行输出，但 printf() 函数是每个 C 编译器的一部分，并且是该语言中最常用的特性。

第4章 输出

4.1 printf()做了什么

简而言之，`printf()`在屏幕上产生输出。如图4-1所示，`printf()`把字符、数字和单词发送到屏幕上。`printf()`有很多选项，但是你不必精通所有这些选项（很少有程序员能做到这一点）才能使用 `printf()`在屏幕上输出。

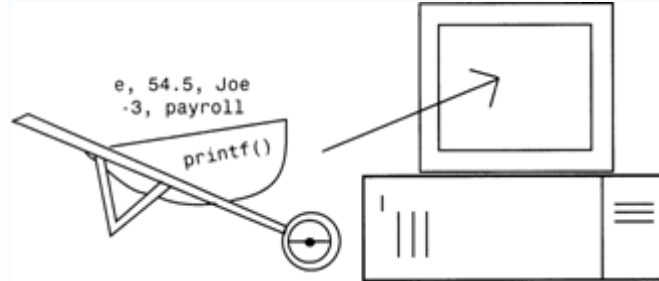


图 4-1 `printf()`把字符、数字和单词发送到屏幕上

4.2 printf()的格式

`printf()`有许多种形式，但是一旦你习惯了它的格式，`printf()`就很容易使用。下面是 `printf()`的一般格式：

```
printf(controlString [, data]);
```

本书常常在你第一次见到某命令和某函数时给出它们的格式。格式是指语句一般的形式。如果格式中有一部分出现在方括号中，如上面的 `data`，那么这部分就是可选的。你不需要输入方括号。如果命令中需要方括号，则会在格式后面的文本中说明清楚。`printf()`需要一个 `controlString`，但其后面的 `data` 是可选的。

警告 `printf()`实际上没有把输出发送到屏幕，而是发送到了电脑的标准输出设备（standard output device）。大多数操作系统，包括 MS-DOS 在内，都将标准输出发送到屏幕，除非你足够了解 MS-DOS，能让它把输出发送到其他地方。大多数时候你可以忽略这个标准输出设备，因为你几乎总是想输出到屏幕上。后面要学习的其他 C 函数可以把输出发送到打印机和硬盘驱动器。

警告 你可能想知道为什么格式中的一些单词用斜体。这是由于它们是占位符（placeholder）。占位符是你提供的名称、符号或公式。在函数和命令的格式中占位符是斜体的，提醒你这个位置要用一些东西来替代。

下面是一个 `printf()`的例子：

```
printf("I am %d", 16); /* Prints I am 16 */
```

因为 C 语言中的每个字符串必须出现在引号中（在第 2 章中提到），所以 `controlString` 必须在引号中。`controlString` 后面的任何东西都是可选的，并且是根据想要打印的值取舍的。

说明 每个命令和函数后面都需要分号（`;`），C 语言据此判断一行是否结束。括号和函数的第一行不需要分号，因为这些行中没有可执行的内容。所有 `printf()`语句都要以分号结束。但是，不需要在 `main()`后面加分号，因为你不会显式地让 C 语言去执行 `main()`。但你需要显式地让 C 语言去执行 `printf()`和其他许多函数。等你学了更多 C 语言知识，你将会更了解分号的放置。

4.3 打印字符串

字符串消息是用 `printf()`打印的最简便的数据类型。你只需要把字符串放在引号中。下面的 `printf()`语句在屏幕上打印一条消息：

```
printf("Read a lot");
```

电脑执行这条语句时，屏幕上会出现 Read a lot。

说明 字符串 `Read a lot` 就是这个 `printf()` 中的 `controlString`。这里几乎没有什么控制 (control)，仅仅是输出。

下面两行 `printf()` 语句

```
printf("Read a lot");
printf("Keep learning");
```

可能不会产生你所期望的输出。实际产生的输出如下：

```
Read a lotKeep learning
```

线索 当 `printf()` 执行时，C 语言不会自动把光标移到下一行。如果想让 C 语言在 `printf()` 之后转到下一行，你必须在 `controlString` 中插入一个转义序列。

4.4 转义序列

C 语言包含许多转义序列，几乎在你写的每个程序中都要用到一些。表 4-1 列出了使用较多的转义序列。

表 4-1 转义序列

代 码	描 述
<code>\n</code>	换行
<code>\a</code>	警报（电脑响铃）
<code>\t</code>	制表符
<code>\\</code>	反斜杠
<code>\"</code>	引号

说明 转义序列其实没有听起来那么难学。一个转义字符在 C 语言中存储为一个字符，并且它产生表 4-1 中描述的效果。例如，当 C 发送 `\a` 到屏幕时，电脑的铃声会响起，而不是真把字符 `\` 和 `a` 打印出来。

你将在 `printf()` 函数中看到很多转义序列。打印多行文本时，如果想要移到下一行，就必须键入 `\n`。C 程序会产生新行 (newline)，它会把闪烁的光标移到屏幕的下一行。在下面的两行 `printf()` 语句中，由于在第一个语句后面有 `\n`，它们会分两行打印消息。

```
printf("Read a lot\n");
printf("Keep learning");
```

请跳过这里，这有点儿难 `\n` 可以放在第 2 行的开始处，输出结果相同。因为转义序列对 C 语言来说是字符，所以必须把它们放在引号中，这样 C 语言就知道转义序列是要被打印的字符串的一部分。下面的代码也会产生两行输出：

```
printf("Read a lot\nKeep learning");
```

因为引号标志字符串的结束，而反斜杠标志转义序列的开始，所以它们都有自己的转义序列。`\a` 让电脑响铃，`\t` 让输出后移一些空格。查看编译器手册，找出你的 C 语言所支持的其他转义序列。尽管表 4-1 中的转义序列几乎是最普遍的（且是与 ANSI C 兼容的），但不是所有的 ANSI C 编译器在所有的编译模式下都支持这些转义序列。例如，Visual C++ 在 QuickWin 模式下就不允许使用 `\a`。

下面的 `printf()` 语句将输出注释里的内容：

```
printf("Ready\tSet\tGo!\n");      /* Ready    Set    Go! */
printf("Ring my charm!\a\n");    /* Ring my charm! <BEEP> */
printf("I said, \"No way.\"\n"); /* I said, "No way." */
printf("\\ means escape\n");     /* \ means escape */
```

线索 不同的编译器可能为 `\t` 转义字符产生不一样多的制表符空格。

4.5 转化字符

打印数字和字符时，必须准确告诉 C 程序如何打印它们。用转化字符 (conversion character) 来表明数字的格式。表 4-2 列举了 C 语言中最常用的转化字符。

表 4-2 转化字符

转化字符	描 述
%d	整数
%f	浮点数
%c	字符
%s	字符串

当你在字符串中打印一个值时，要在 `controlString` 中插入合适的转化字符，然后在 `controlString` 的右边，列出想要打印的值。图 4-2 是一个例子，它说明了 `printf()` 如何打印 3 个数字——一个整数、一个浮点数和另一个整数。

输出：

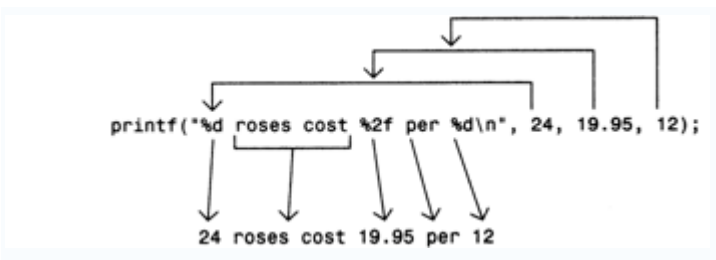


图 4-2 `printf()`转化字符决定了怎样打印以及在哪里打印数字

字符串和字符也有自己的转化字符。尽管只打印字符串时不需要 `%s`，但是当打印与其他数据组合的字符串时，就可能需要用 `%s`。下面的 `printf()` 用各种转化字符打印几种不同类型的数据：

```
printf("%s %d %f %c\n", "Sam", 14, -8.76, 'X');
```

上述代码的输出如下：

```
Sam 14 -8.760000 X
```

说明 字符串 `Sam` 需要加上引号，所有的字符串都需要这样。而字符 `X` 需要加上单引号，所有的字符都是这样。

警告 当 C 语言遇到浮点数就会变得疯狂。尽管 `-8.76` 只有两个小数位，但 C 语言却坚持打印出 6 个小数位。

可以通过在浮点转化字符的 `%` 和 `f` 之间加上一个。来控制 C 程序如何打印浮点数值。图 4-2 使用了这种小数控制。尽管 `Printf()` 语句中给定的浮点数相同，不过运行输出的 4 个数却不相同：

```
printf("%f %.3f %.2f %.1f", 4.5678, 4.5678, 4.5678, 4.5678);
```

C 程序把浮点数四舍五入到在 `%f` 转化字符中指定的小数位，并产生如下输出：

```
4.567800 4.568 4.57 4.6
```

线索 等你在下一章中学习变量时，转化字符会变得更有用。

最后一个要点：`printf()` 的 `controlString` 严格控制着输出结果。之所以在数字之间出现两个空格，是因为 `controlString` 中各个 `%f` 之间有两个空格。

奖励

- l 如果想在屏幕上打印数据就用 `printf()`。
- l 每个 `printf()` 需要一个控制字符串，用来决定数据打印的结果。
- l 打印字符串很容易。它们不需要特殊的格式代码或转化字符。
- l 用转义序列实现换行、制表符、引号、反斜杠和电脑响铃。
- l 用转化字符来控制数字打印的结果。

陷阱

- l 不要指望 C 语言知道如何自动格式化数据。你必须自己使用转化字符。
- l 不要忘记用 `%f` 的小数控制，除非你想让 C 程序把所有的浮点数都打印出 6 个小数位。

4.6 小结

`printf()` 把数据输出到屏幕上。不能输出到屏幕的程序几乎没有用。你写的程序必须能够与键盘边的用户交流。

`printf()` 需要一个 `controlString`，它描述了其后面的数据格式。通过使用 `controlString`，能够准确指定数字和字符数据如何打印。你也可以打印转义序列，这是一种有时要用到的特殊的输出控制，比如要在输出的行尾换行。

1. 代码范例

看一看下面的部分代码清单：

```
printf("%c %s %d %f %.2f\n", 'Q', "Hello!", 14, 64.21, 64.21);
printf("%c\n", 'Q');
printf("%s\n", "Hello!");
printf("%d\n", 14);
printf("%f\n", 64.21);
printf("%.2f", 64.21);
```

2. 代码分析

第一个 `printf()` 打印出 5 个数据值——一个字符、一个字符串、一个整数、两个浮点数。接下来的 5 行把这些值一次打印一个。最后一个浮点值的小数位数在最后一个 `printf()` 中指定，用来限制打印出来的小数位。下面是代码的输出：

```
Q Hello! 14 64.210000 64.21
Q
Hello!
14
64.210000
64.21
```

众所周知电脑可以处理数据。不管怎样，你必须有办法存储这些数据。与大多数编程语言一样，C 语言把数据存储在变量（`variable`）中。变量仅仅是电脑内存中的盒子，用来保存数字或字符。第 2 章讲解了不同的数据类型：字符、字符串、整数和浮点数。本章讲解如何在程序中存储这些类型的数据。

第 5 章 变量

5.1 变量类型

由于有几种不同类型的数据，所以 C 语言中也有几种不同类型的变量。任何一种变量都不能保存几种数据。只有整型变量才能保存整型数据，只有浮点型变量才能保存浮点数据，如此类推。

说明 在本章我们将电脑中的变量看成是邮局中的邮箱。邮箱的大小各不相同，且有彼此不同的编号。程序中变量的大小也不同，这取决于它保存哪种数据类型。此外，每个变量都有不同的名字与其他变量相区别。

第 2 章中所学的数据称为字面量数据（`literal data`）或常量数据（`constant data`）。特定的数字和字母不会改变。数字 2 和字符 'x' 始终是 2 和字符 'x'。不过，要处理的大多数数据都是要改变的。如年龄、工资和重量等数据都是变化的。如果想编写一个工资表程序，就需要一个方式来保存变化的数据。变量可以帮你实现。变量就是内存中用来存放数据的小盒子，里面的数值可以随时间改变。

有许多变量类型。表 5-1 列出了较常用的变量类型。注意，许多变量的数据类型（字符、整型和浮点型）与字面量数据很类似。毕竟，你必须有一个地方来存放整数，即使用整型变量。

表 5-1 几种变量类型

名 称	描 述
<code>char</code>	存放字符数据，如 'x' 和 '*'
<code>int</code>	存放整型数据，如 1、32 和 -459。存放的数据在 -32768 和 32767 之间

<code>long int</code>	存放大于 32767 和小于-32768 的整型数据
<code>float</code>	存放浮点数，如 0.0003、-121.34 和 43323.4
<code>double</code>	存放极大和极小的浮点数。 <code>float</code> 只能存放从 -3.4×10^{38} 到 3.4×10^{38} 之间的数

警告 你可能注意到了，尽管有字符串字面量类型，但却没有字符串变量。`C` 是少数几种没有字符串变量的一种编程语言，但是第 6 章会介绍如何在变量中存储字符串。

表 5-1 中的名称一列给出了编程时创建变量要用的关键字。也就是说，如果想要一个整型变量，就要用 `int` 关键字。最后你还需要知道如何给变量命名。

5.2 变量的命名

所有的变量都有名字，而且由于是你负责对它们命名，你必须掌握命名规则。所有的变量名必须不同。在同一个程序中不能有两个相同名字的变量。

变量名可以包含 1 到 32 个字符。变量名必须以字母开头，后面可以是字母、数字或下划线的组合。下面所列的都是合法的变量名：

```
myData  pay94  age_limit  amount  QtlyIncome
```

线索 `C` 语言允许变量名以下划线开头，但是你不应该这么做。因为 `C` 语言的一些内置变量以下划线开头，如果你的变量也以下划线开头，就有可能与某个内置变量重名。

下面这些变量名是不合法的：

```
94Pay  my Age  rate*pay
```

你应当明白为什么这些变量名不合法。第 1 个 `94pay`，以数字开头；第 2 个变量名 `my Age`，含有一个空格；而第 3 个变量名 `rate*pay`，含有一个特殊字符（*）。

警告 不要用与函数或命令相同的名字给变量命名。如果你给了一个变量和某命令相同的名字，那么程序将无法运行；如果你给了一个变量和某函数相同的名字，那么在后面的程序中将无法使用相同的函数名，并且不会报错。

5.3 定义变量

变量使用之前必须先定义（define）。变量定义（有时称为变量声明，`variable declaration`）是为了让 `C` 程序知道你需要一些变量空间，于是 `C` 程序会为你保留。定义一个变量，只需声明它的类型，后面跟上变量名。下面是某程序的第一行，这里定义了一些变量：

```
main()
{
    char initial;
    int age;
    float amount;
    /* Rest of program would follow */
}
```

上面的范例代码中有 3 个变量——`initial`、`age` 和 `amount`。它们可以存放 3 种不同类型的数据——字符数据、整型数据和浮点型数据。如果程序没有定义这些变量，就不能在变量中存放数据。

在同一行上可以定义多个相同类型的变量。例如，如果想定义两个字符变量而不是一个，那么你可以这样做：

```
main()
{
    char initial1, initial2;  /* Defines 2 characters */
    int age;
    float amount;
    /* Rest of program would follow */
}
```

或者这样做：


```
main()
{
    char initial1;
    char initial2; /* Defines a second */
    int age;
    float amount;
    /* Rest of program would follow */
}
```

请跳过这里，这有点儿难。大多数 C 语言变量被定义在左大括号之后，如跟在函数名后面的左大括号。这些变量称为局部变量（**local variable**）。C 语言也允许通过在函数名之前定义变量来创建全局（**global**）变量，如在 `main()` 之前。局部变量几乎总是比全局变量更可取。第 30 章会讲解局部变量和全局变量的区别，但目前，所有的程序都将用局部变量。

5.4 在变量中存储数据

赋值运算符（**assignment operator**）在变量中存放值。它用起来比听起来简单得多。赋值运算符只是一个等号（`=`）。给变量赋值的格式如下：

```
variable = data;
```

其中，*variable* 是想存放数据的变量名。你必须已经在前面定义了这个变量（像前一节所说的那样）。*Data* 可以是数字、字符或产生数字的数学表达式。下面是 3 条赋值语句的范例，它们给前一节定义的变量赋值：

```
initial = 'G'; /* Assigns values to three variables */
age = 31;
amount = 2983.43;
```

你也可以在变量中存放表达式的值：

```
sales = 4432.67 / 1.20; /* Divides to get value */
```

甚至可以在表达式中使用其他变量：

```
newSales = sales + 2167.65; /* Uses value from
                             another variable */
```

图 5-1 说明了当你使用赋值语句时，C 语言做了什么。

线索 等号会告诉 C 程序把等号右边的任何东西粘到左边的变量中。等号的作用就像往左指的箭头，表示“往这儿走”。哦，绝对不要在数字中使用逗号，无论数字有多大！

假设一个小企业支付你很多钱让你帮他们编写一个薪水计算程序。你就要定义一些浮点数来存放每小时支付的费用、工作的小时数以及税率等。附录 B 中的 21 点程序必须记录很多东西，因此用了很多变量。大多数程序会在函数的开始处定义变量。

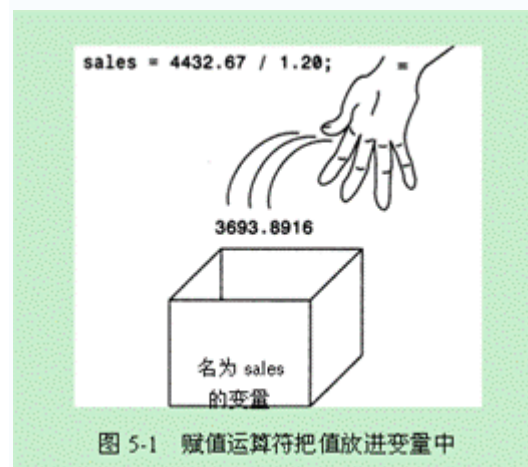


图 5-1 赋值运算符把值放进变量中

线索 你可以在定义变量的同时给它赋初值。下面的代码定义了 3 个变量，并给其中两个（`val` 和 `numSold`）赋了初值：

```
int numSold = 25, numBought;
float val = 436.54;
```


奖励

- | 学会如何命名变量，因为你将会在程序中用到它们。
- | 总是在使用之前定义变量。
- | 在同一行上可以定义多个变量。
- | 等号称为赋值运算符。赋值运算符用来在变量中存储值。

陷阱

| 不要混淆数据类型和变量类型。避免把一种数据类型的值存储到另一种数据类型的变量中。

| 尽管可以，但最好不要在函数的前面定义变量（这样的变量称为全局变量，如果不小心使用，容易导致错误），而应该在函数的左大括号之后定义变量。目前对你来说这些局部变量使用起来更安全。

- | 不要在数字中用逗号。输入值 3 万应该是 30000，而不是 30,000。

5.5 小结

本章介绍了 C 语言中不同的变量类型。因为存在不同类型的数据，所以必须有不同类型的变量来存放它们。数据类型的选择取决于程序员。要小心选择变量的数据类型，确保此类型与要存放在变量中的值相匹配。

当较小的数据类型足够存放数据时，不要使用大的数据类型。当 int 足够用时使用 long int 不仅会影响效率，而且还会导致程序庞大且运行缓慢。

1. 代码范例

```
main()
{
    /* Defines a different variable for each data type */
    char priceCode = 'J';
    int quantity = 100;

    long int wholeSaleQuant = 45000;
    float price = 13.54;
    double yrlySales = 9845543.23;
```

2. 代码分析

上面程序段中的语句既保留了变量的空间又给它们赋了初值。不过，并非总要给变量赋初值。很多时候你并不知道变量将存放什么值，因为这个值可能来自用户的键盘输入、磁盘文件，或者计算结果。无论是否知道初值，你都必须在使用它们之前为变量保留存储空间。

尽管 C 语言没有字符串变量，仍然有办法存储字符串数据。本章将讲解它的实现方法。你已经知道字符串数据必须放在引号中。即使是一个字符，如果放在引号中也是字符串。你也知道如何用 printf() 打印字符串。

剩下的唯一任务就是如何用一种特殊的字符变量类型来存放字符串数据，这样你的程序就可以输入、处理并输出字符串了。

第 6 章 字符串

6.1 字符串结束符

C 语言对字符串做了最奇怪的事：在每个字符串的末端加上了一个零。字符串末端的零有几种名字。下面列出了一些：

- | Null 零
- | 二进制零
- | 字符串结束符
- | ASCII 零
- | \0

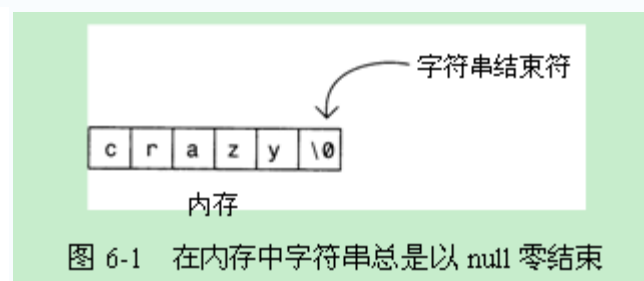
警告 你唯独不能把字符串末端的零叫做零（zero）。C 程序员给字符串末端的零起了特殊的名字，这样你就知道在字符串末端用的既不是普通的数值零也不是字符'0'。只有 **null** 零出现在字符串末端。

C 语言用字符串末端的零标识所有字符串的结束。当你输入文字 *My name is Julie* 时，不需要做什么特别的事情。C 程序会自动加上 **null** 零。你永远看不到 **null** 零，但它确实存在。在内存中，只有当 C 程序遇到 **null** 零时才知道字符串结束了。

说明 查看附录 C 时，你会找到 **ASCII** 表（在第 2 章中讨论）。第 1 个条目就是 **null**，它的 **ASCII** 数是 0。往下找到 **ASCII 48**，会看到一个 0。**ASCII 48** 是字符'0'，而第 1 个 **ASCII** 值是 **null** 零。C 语言把 **null** 零放在字符串的末端。即使字符串 *"I am 20"* 也以 **ASCII 0** 结束，它就在 20 的字符 0 后面。

线索 字符串结束符有时被称为 **\0**（反斜杠零），这是因为可以把 **\0** 放在单引号中表示 **null** 零。因此，'0'是字符零，而**\0**是字符串结束符。

图 6-1 展示了字符串 *"Crazy"* 在内存中是如何存储的。如你所见，尽管字符串只有 5 个字母，但它占用了 6 个字节（一个字节是一个存储单元）。作为字符串 *"Crazy"* 的一部分，**null** 零也占用了一个字节。



6.2 字符串的长度

字符串的长度（length）是指最大的字符数，但不包括 **null** 零。有时候你需要找出字符串的长度。计算字符串长度时，**null** 零不能算在内。尽管字符串必须以 **null** 零结束（这样 C 语言知道字符串在哪里结束），但是 **null** 零并不是字符串长度的一部分。

根据定义，下面的字符串长度都是 9 个字符：

```
August 10
和
Batter up
```

警告 第 1 个字符串的长度不在 10 中的 0 处结束，因为 10 中的 0 不是 **null** 零，而是字符零。

线索 数据的所有单个字符的长度都是 1。因此，'x'和"x"的长度都是 1，但"x"占用了两个字符的内存，因为它带有 **null** 零。无论何时，每次看到引号中的字符串字面量时，就要想到在内存中字符串末端的 **null** 零。

6.3 字符数组：字符的列表

字符数组（character array）在内存中存放字符串。数组（array）是一种特殊的变量类型，在后面的章节中你会了解更多。所有的数据类型——**int**、**float**、**char** 等，都有相应的数组类型。数组只不过是具有相同数据类型的许多变量的一个列表。

在使用字符数组存放字符串之前，你必须告诉 C 语言你需要一个字符数组，也是在声明其他类型的变量的地方声明。在数组名后面使用方括号[和]，还要用一个数字来表明将要存放的字符串的最大字符数。

趣闻轶事
可以用 C
语言创建
自己的计
算机语言。

例子总是胜过千言万语。如果需要存放月份名字，你可以像下面这样定义一个名为 month 的字符数组：

```
char month[10]; /* Defines a character array */
```

线索 数组的定义很简单。去掉 10 和方括号，就是一个普通的字符变量。加上带有 10 的方括号就是告诉 C 语言你需要 10 个字符变量，它们都排列在名为 month 的列表中。

在定义该数组时用 10 的原因是最长的月份名 September 有 9 个字符。第 10 个字符是为——你已经猜到了——null 零准备的。

线索 你总是得预留足够的字符数组空间容纳最长的字符串，还要算上字符串结束符。你可以定义比需要的更多的数组字符，但是不能比所需要的少。

如果你愿意，可以在定义数组的同时存储字符串：

```
char month[10] = "January"; /* Defines a character array */
```

图 6-2 展示了数组在内存中是如何存储的。因为数组的最后两位没有存放东西（January 占用 7 个字符加上第 8 个字符 null 零），你不知道最后两位里放的是什么。但是，有些编译器会将未使用的元素用 0 填充。

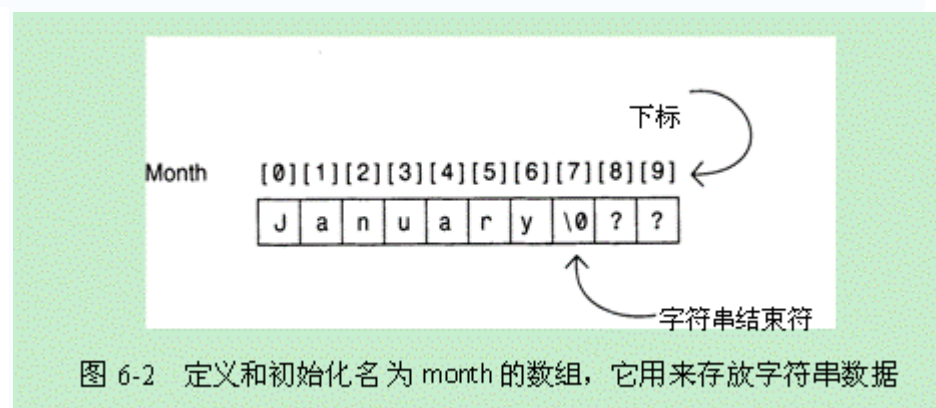


图 6-2 定义和初始化名为 month 的数组，它用来存放字符串数据

线索 数组的每个单元被称为元素（element）。month 数组有 10 个元素。你可以用下标（subscript）来区分它们。下标是你在方括号中指定的数字，可以访问到每个数组元素。

所有的数组下标以 0 开始。如图 6-2 所示，month 数组中的第一个元素叫做 month[0]。最后一个元素叫做 month[9]，因为一共有 10 个元素，当从 0 开始时，最后一个就是 9。

字符数组的每个元素都是一个字符。字符的组合——数组或字符列表——存放整个字符串。如果你愿意，可以一次一个元素地把数组的内容从 January 改为 March，像这样：

```
month[0] = 'M';  
month[1] = 'a';  
month[2] = 'r';  
month[3] = 'c';  
month[4] = 'h';  
month[5] = '\0'; /* Very important */
```

在字符串的末尾加上 null 零是至关重要的。否则，month 数组仍然在 month[7] 处有一个 null 零。打印字符串时，你将得到：

```
Marchry
```

线索 打数组中的字符串很简单。可以使用 %s 转化字符：

```
printf("The month is %s", month);
```

请跳过这里，这有点儿难。如果你定义一个数组并同时对其进行了初始化，你就可以不必在方括号里加上数字。下面的两句话做了相同的事：

```
char month[8] = "January";
```

和

```
char month[] = "January";
```

在第二个例子中，C 语言计算 **January** 的字符数，并为最后一个 **null** 零加上 1。但是，你就不能存放 8 个以上字符的字符串了。如果你想定义一个字符串的字符数组，并对其进行初始化，但又要为将来存放更长的字符串留下空间，你可这样做：

```
char month[25] = "January"; /* Leaves room for  
longer strings */
```

6.4 初始化字符串

你一定不希望像前一节那样一次一个字符地初始化字符串。但是，与普通的非数组变量不同，你不能像下面这样把一个字符串赋给一个数组：

```
month = "April"; /* NOT allowed */
```

你只能在定义字符串的同时用等号给字符串赋值。如果在后面的程序中你想给数组一个新的字符串，就必须一次一个字符地赋值，或者用 C 编译器带的 **strcpy()** 函数（字符串复制）来实现。下面的语句把一个新的字符串赋给 **month**：

```
strcpy(month, "April"); /* Puts new string in month array */
```

线索 在使用了 **strcpy()** 的程序中，必须在 **#include <stdio.h>** 后面加上这一行：

```
#include <string.h>
```

警告 不用担心。**strcpy()** 会自动在字符串末尾加上 **null** 零。

说明 附录 B 中的 21 点游戏使用字符数组来存放玩家的名。看看你是否能找到使用字符数组的函数。可不要告诉我是 **main()** 函数。

奖励

- | 在字符数组中存储字符串。
- | 预留足够的数组元素容纳最长的字符串。
- | 可以在定义字符数组时对其进行初始化，即一次给一个元素赋值，或者使用 **strcpy()** 函数实现。
- | 如果使用 **strcpy()**，要确保加上了 **#include <String.h>**。

陷阱

- | 只有当字符数组包含足够多的元素用于存放字符串时，才能把字符串放到字符数组中。
- | 不要忘了数组的下标从 0 开始，而不是 1（有些编程语言是从 1 开始）。
- | 不要计算错了！如果你没有为结尾的零预留足够的元素，C 语言将不能正确处理字符数组。

6.5 小结

除了第 5 章中介绍的存储数值型数据，C 语言还提供了存储字符串的方法。与单个字符的变量不同，字符串可以保存许多字符，如单词、地址和文本段。

C 语言不支持字符串变量类型。尽管通过不支持字符串变量获得了一定的效率，程序员也不能放弃存储字符串数据的能力。因此，C 语言提供了字符数组来以类似字符串的形式存放包含多个字符的数据。

1. 代码范例

```

/* Stores the days of the week in seven
different character arrays */
char day1[7] = "Sunday";
char day2[7] = "Monday";
char day3[8] = "Tuesday";
char day4[10] = "Wednesday";
char day5[] = "Thursday";
char day6[] = "Friday";
char day7[] = "Saturday";
char myName[6];
strcpy(myName, "Julie");

```

2. 代码分析

当你想存储字符串数据时，要像上面这样定义字符数组。你必须在变量名后面加上方括号，否则 C 语言会认为你在定义单个字符变量。记住，一定要为字符串末尾的 `null` 零预留足够的空间。如果在定义数组时给它赋了一个字符串，那么你就不必计算字符数并为 `null` 零又加 1 了。最后的 3 个 `day` 变量没有写上字符总数，因为 C 语言能够计算出要保存的数据所需的数目。

要注意上面的 7 个数组名正好都包含了数字，但是这些数字与下标没有关系。`day4[]` 有 10 个元素，因为初始定义的下标为 10。`day4` 中的 4 只是为了与其他数组相区别。

如果你想定义没有赋值的数组，就像 `myName` 数组那样，那么必须在定义数组时包含最大的元素个数。然后可以用 `strcpy()` 给数组赋上字符串值。

在很多 C 程序中有两种代码行不是 C 命令。它们是预处理器指令（`preprocessor directive`）。预处理器指令总是以符号 `#` 开始，它们不会在运行时（即运行程序时）产生任何行为。它们只在编译程序时发生作用。

最常用的预处理器指令是：

```

| #include;
| #define。

```

在前面的章节中，你已经看到使用 `#include` 的例子。本章将最终揭开预处理指令的神秘面纱。

第 7 章 `#include` 和 `#define`

7.1 包含文件

`#include` 有两种格式，二者几乎等价：

```

#include <filename>
和
#include "filename"

```

图 7-1 展示了 `#include` 做了些什么。它只不过是一个文件合并（`file merge`）命令。就在程序被编译前，`#include` 语句被 `#include` 后的文件的内容替换了。文件名可以用大写或小写字母表示，前提是你的操作系统允许这么做。例如，MS-DOS 不区分文件名中的大写和小写字母，但 UNIX 却区分。如果你的文件名是 `myFile.txt`，MS-DOS 允许你使用下面任意一个 `#include` 指令：

```

#include <MYFILE.TXT>
#include <myfile.txt>
#include <myFile.txt>

```

但 UNIX 只允许你使用这个：

```

#include <myFile.txt>

```

线索 如果你用过文字处理器，你就很可能用过 `#include` 类型的命令。你可能曾将存储在硬盘上的文件合并到正在编辑的文件中。

下面是你写的：
你的源文件：

之后

之前

下面是编译器看到的：

名为 `addr.h` 的文件：

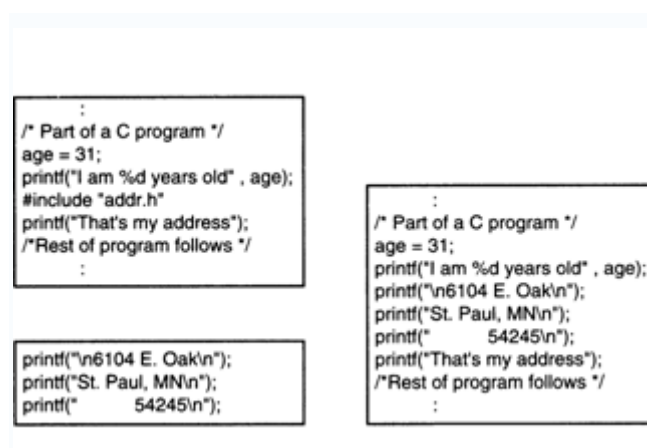


图 7-1 `#include` 把磁盘文件插入到另一个文件中

当你安装编译器时，安装程序会在硬盘上创建一个独立的位置（在一个目录中）存放编译器附带的各种`#include`文件。当你想要使用某个内置`#include`文件时，要使用带尖括号`<`和`>`的`#include`格式。

警告 你会问：“我怎么知道什么时候要用内置的`#include`文件？”问得好！所有的内置函数，如 `printf()`，都有相应的`#include`文件。本书在描述内置函数时，也会告诉你要包含哪个文件。

线索 编译器手册中有一个完整列表，上面列出了哪个函数包含在哪个文件中。

说明 你已经看到两个内置函数——`printf()`和 `strcpy()`。`main()`不是内置函数，`main()`是一个你必须提供的函数。`printf()`对应的`#include`文件是 `STDIO.H`（它代表了标准输入/输出），而 `strcpy()`对应的`#include`文件是 `STRING.H`。

线索 大多数 C 编译器提供上下文相关的帮助（`context-sensitive help`）。如果你的编译器有此功能，你可以将鼠标指针移到内置函数名（如 `strcpy()`）上面，并选择上下文相关的帮助（一般可以按 `Alt+F1` 组合键）。编译器会在帮助消息中告诉你要使用这个函数必须包含哪个头文件。

本书中几乎所有的代码清单都包含如下的预处理器指令：

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

这是因为书中几乎每一个程序都使用了 `printf()`。第 6 章会包含 `STRING.H`，因为要讨论 `strcpy()`函数。

线索 你所包含的文件称为头文件（header file）。这就是大多数被包含的文件以扩展名.H 结束的原因。

如果你写了自己的头文件，就要用第二种形式的预处理器指令——带引号的那种。如果你使用引号，C 语言会首先在程序所在的磁盘目录中搜索文件，再到内置的#include 目录中搜索。由于这种搜索顺序，你可以编写自己的头文件，然后给它们与 C 语言内置的头文件相同的名字，这样你的文件就能代替 C 语言的文件使用了。

警告 如果你写了自己的头文件，不要把它们放在 C 语言内置的#include 文件目录中。请保持 C 语言提供的头文件完好无损。几乎没有理由覆盖 C 语言的头文件，但你可以增加一些自己的头文件。

说明 当你有要在很多程序中经常使用的程序语句时，你或许可以编写自己的头文件。可把这些语句放在程序目录下的一个文件中，当你要用到它们时就用#include 包含这个文件，这样就不用在每个程序中都输入它们了。

7.2 在哪里放置#include 指令

你用#include 包含的头文件只是一些包含 C 代码的文本文件。后面你将学到更多关于头文件的知识，但现在只需理解头文件做两件事情。内置的头文件帮助 C 语言正确地执行内置函数。自己写的头文件通常包含你想要写到多个文件中的代码。

说明 附录 B 中的 21 点游戏程序包含了很多头文件，因为它使用了很多内置函数。注意#include 的位置：它们出现在 main() 之前。

7.3 定义常量

#define 预处理器指令用来定义常量（constant）。C 常量其实就是字面量。在第 2 章中学习了字面量就是不会改变的值，就像数字 4 或字符串"C programming"。#define 预处理器指令允许给字面量命名。当你给字面量命名时，命名后的字面量在 C 语言术语中称为命名常量（named constant）或定义常量（defined constant）。

警告 在第 5 章中你学过如何通过指定数据类型、变量名称和初始值来定义变量。用#define 定义的常量不是变量，尽管它们在使用时有点像变量。

下面是#define 指令的格式：

```
#define CONSTANT constantDefinition
```

如同 C 语言中大多数事物一样，使用定义常量实际上要比看起来容易。下面是一些#define 指令的例子：

```
#define AGELIMIT 21
#define MYNAME "Paula Holt"
#define PI 3.14159
```

线索 简而言之，#define 告诉 C：把程序中 CONSTANT 出现的每个地方都用 constantDefinition 替换。

上面的第一个#define 语句指示 C 语言找出单词 AGELIMIT 出现的每个地方，并将其替换为 21。因此，如果下面这条语句出现在#define 后面的程序中：

```
if (employeeAge < AGELIMIT)
```

编译器会认为你输入了：

```
if (employeeAge < 21)
```

尽管你没有。

请跳过这里，这有点儿难。定义常量名要用大写字母。这是 C 语言中的一个少见的例外，而且推荐使用大写字母。因为定义常量不是变量，大写能使你在快速浏览程序时分辨出哪些是变量，哪些是常量。

线索 假设你前面定义过常量 PI，大写字母能帮助你避免在程序中做这样的事情：

```
PI = 544.34; /* Not allowed */
```

只要你坚持用大写字母定义常量，你就能知道不要去改变它们，因为它们是常量。

趣闻轶事
UNIX 操作
系统 (DOS
的竞争者)
的大部分
是用 C 语
言写的。

有些值可能在程序运行期间要被修改，定义常量对于给这样的值命名是有好处的。例如，如果你不为 AGELIMIT 使用定义常量，而在程序中使用实际的年龄限制值如 21，那么寻找并修改每个 21 是很困难的。如果你已经在程序顶部用了定义常量，当要修改年龄限制值时，你只需要像下面这样修改 `#define` 语句：

```
#define AGELIMIT 18
```

警告 `#define` 指令不是 C 语言命令。与 `#include` 一样，C 语言在编译程序之前处理 `#define` 语句。因此，如果你定义了 PI 为 3.14159，并且在需要用到数学上的 pi (p) 的程序中使用了 PI，那么当你在程序中实际上输入 PI 时，C 语言会认为你输入了 3.14159。PI 更容易记住（帮助消除输入错误）且更清晰地表明为一个常量。

只要你在 `main()` 出现之前用 `#define` 定义了常量，则整个程序都能认得这个常量。因此，如果你在 `main()` 之前把 PI 定义为值 3.14159，你就可以在 `main()` 和后面任何其他你写的函数中使用 PI，并且编译器会在编译你的程序之前把 PI 替换为 3.14159。

奖励

- l 使用内置函数时需要使用正确的头文件。
- l 当包含编译器提供的头文件时，用尖括号 <和> 把文件名括起来。
- l 当包含位于源代码目录下自己写的头文件时，用引号 "和" 把文件名引起来。
- l 在所有的定义常量中使用大写字母，这样你就能把它们与普通变量名区分开来了。

陷阱

l 不要把为内置函数写的 `#include` 语句放在 `main()` 后面。在 `main()` 之前用 `#include` 包含头文件。你可以在任何时候用 `#include` 包含自己写的头文件。

l 不要把定义常量当成变量看待。与变量不同，一旦常量被定义之后，你就不能在常量中存储数据了。

7.4 小结

C 语言的预处理器指令使 C 语言能够看到你没有实际输入的代码。例如，当你需要另一个文件（如 `STDIO.H`）的内容时（此文件帮助你正确地产生输入和输出），你不必键入文件的内容。你只需通过 `#include` 指令指示 C 语言包含这个文件即可。

如果程序中有一个常量值，如销售奖金限制，你应该在程序的顶部用 `#define` 定义这个常量。在程序中就只需键入常量的名字，而不用键入实际数字了。如果这个限制值发生了改变，你只需要修改 `#define` 这一行代码。

`#include` 和 `#define` 不是与程序的其他部分一起运行的 C 语句。它们只对源代码上起作用，即在 C 语言编译程序之前包含文本或者把定义名改为实际的值。

1. 代码范例

```
#include <stdio.h>
#include "mycode.h"
#define MINORDER 50
#define COMPNAME "Amalgamated Co."
```

2. 代码分析

这段代码只包含几个预处理器指令。第一列中的#是预处理指令的标志。第一行指示 C 程序把 `STDIO.H` 文件复制到当前程序中。由于使用了尖括号，编译器会查找安装时创建的普通的包含目录。第二行用了引号，这会指示 C 程序首先在源代码目录下查找该文件。

随后定义了两个定义常量，`MINORDER` 和 `COMPNAME`。当后面的程序需要测试或打印最小订单数量或公司名字时，就用定义常量名代替常量本身。