

C 控制语句：循环

本章介绍以下内容：

- 关键字：for、while、do while
- 运算符：<、>、>=、<=、!=、==、+=、*=、-=、/=、%=?
- 函数：fabs()
- C 语言有 3 种循环：for、while、do while
- 使用关系运算符构建控制循环的表达式
- 其他运算符
- 循环常用的数组
- 编写有返回值的函数

大多数人都希望自己是体格强健、天资聪颖、多才多艺的能人。虽然有时事与愿违，但至少我们用 C 能写出这样的程序。诀窍是控制程序流。对于计算机科学（是研究计算机，不是用计算机做研究）而言，一门语言应该提供以下 3 种形式的程序流：

- 执行语句序列；
- 如果满足某些条件就重复执行语句序列（循环）；
- 通过测试选择执行哪一个语句序列（分支）。

读者对第一种形式应该很熟悉，前面学过的程序中大部分都是由语句序列组成。while 循环属于第二种形式。本章将详细讲解 while 循环和其他两种循环：for 和 do while。第三种形式用于在不同的执行方案之间进行选择，让程序更“智能”，且极大地提高了计算机的用途。不过，要等到下一章才介绍这部分的内容。本章还将介绍数组，可以把新学的知识应用在数组上。另外，本章还将继续介绍函数的相关内容。首先，我们从 while 循环开始学习。

6.1 再探 while 循环

经过上一章的学习，读者已经熟悉了 while 循环。这里，我们用一个程序来回顾一下，程序清单 6.1 根据用户从键盘输入的整数进行求和。程序利用了 scanf() 的返回值来结束循环。

程序清单 6.1 summing.c 程序

```
/* summing.c -- 根据用户键入的整数求和 */
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;           /* 把 sum 初始化为 0 */
    /* ... */
```

```

int status;

printf("Please enter an integer to be summed ");
printf("(q to quit): ");
status = scanf("%ld", &num);
while (status == 1) /* == 的意思是“等于” */
{
    sum = sum + num;
    printf("Please enter next integer (q to quit): ");
    status = scanf("%ld", &num);
}
printf("Those integers sum to %ld.\n", sum);

return 0;
}

```

该程序使用 long 类型以储存更大的整数。尽管 C 编译器会把 0 自动转换为合适的类型，但是为了保持程序的一致性，我们把 sum 初始化为 0L (long 类型的 0)，而不是 0 (int 类型的 0)。

该程序的运行示例如下：

```

Please enter an integer to be summed (q to quit): 44
Please enter next integer (q to quit): 33
Please enter next integer (q to quit): 88
Please enter next integer (q to quit): 121
Please enter next integer (q to quit): q
Those integers sum to 286.

```

6.1.1 程序注释

先看 while 循环，该循环的测试条件是如下表达式：

```
status == 1
```

$\mathbf{==}$ 运算符是 C 的相等运算符 (*equality operator*)，该表达式判断 status 是否等于 1。不要把 status == 1 与 status = 1 混淆，后者是把 1 赋给 status。根据测试条件 status == 1，只要 status 等于 1，循环就会重复。每次循环，num 的当前值都被加到 sum 上，这样 sum 的值始终是当前整数之和。当 status 的值不为 1 时，循环结束。然后程序打印 sum 的最终值。

要让程序正常运行，每次循环都要获取 num 的一个新值，并重置 status。程序利用 scanf() 的两个不同的特性来完成。首先，使用 scanf() 读取 num 的一个新值；然后，检查 scanf() 的返回值判断是否成功获取值。第 4 章中介绍过，scanf() 返回成功读取项的数量。如果 scanf() 成功读取一个整数，就把该数存入 num 并返回 1，随后返回值将被赋给 status (注意，用户输入的值储存在 num 中，不是 status 中)。这样做同时更新了 num 和 status 的值，while 循环进入下一次迭代。如果用户输入的不是数字 (如，q)，scanf() 会读取失败并返回 0。此时，status 的值就是 0，循环结束。因为输入的字符 q 不是数字，所以它会被放回输入队列中 (实际上，不仅仅是 q，任何非数值的数据都会导致循环终止，但是提示用户输入 q 退出程序比提示用户输入一个非数字字符要简单)。

如果 scanf() 在转换值之前出了问题 (例如，检测到文件结尾或遇到硬件问题)，会返回一个特殊值 EOF (其值通常被定义为 -1)。这个值也会引起循环终止。

如何告诉循环何时停止？该程序利用 scanf() 的双重特性避免了在循环中交互输入时的这个棘手的问题。例如，假设 scanf() 没有返回值，那么每次循环只会改变 num 的值。虽然可以使用 num 的值来结束循环，比如把 num > 0 (num 大于 0) 或 num != 0 (num 不等于 0) 作为测试条件，但是这样用户就

不能输入某些值，如 -3 或 0。也可以在循环中添加代码，例如每次循环时询问用户“是否继续循环？<y/n>”，然后判断用户是否输入 y。这个方法有些笨拙，而且还减慢了输入的速度。使用 `scanf()` 的返回值，轻松地避免了这些问题。

现在，我们来看看该程序的结构。总结如下：

把 `sum` 初始化为 0

提示用户输入数据

读取用户输入的数据

当输入的数据为整数时，

 输入添加给 `sum`，

 提示用户进行输入，

 然后读取下一个输入

输入完成后，打印 `sum` 的值

顺带一提，这叫作伪代码 (*pseudocode*)，是一种用简单的句子表示程序思路的方法，它与计算机语言的形式相对应。伪代码有助于设计程序的逻辑。确定程序的逻辑无误之后，再把伪代码翻译成实际的编程代码。使用伪代码的好处之一是，可以把注意力集中在程序的组织和逻辑上，不用在设计程序时还要分心如何用编程语言来表达自己的想法。例如，可以用缩进来代表一块代码，不用考虑 C 的语法要用花括号把这部分代码括起来。

总之，因为 `while` 循环是入口条件循环，程序在进入循环体之前必须获取输入的数据并检查 `status` 的值，所以在 `while` 前面要有一个 `scanf()`。要让循环继续执行，在循环内需要一个读取数据的语句，这样程序才能获取下一个 `status` 的值，所以在 `while` 循环末尾还要有一个 `scanf()`，它为下一次迭代做好了准备。可以把下面的伪代码作为 `while` 循环的标准格式：

获得第 1 个用于测试的值

当测试为真时

 处理值

 获取下一个值

6.1.2 C 风格读取循环

根据伪代码的设计思路，程序清单 6.1 可以用 Pascal、BASIC 或 FORTRAN 来编写。但是 C 更为简洁，下面的代码：

```
status = scanf("%ld", &num);
while (status == 1)
{
    /* 循环行为 */
    status = scanf("%ld", &num);
}
```

可以用这些代码替换：

```
while (scanf("%ld", &num) == 1)
{
    /* 循环行为 */
}
```

第二种形式同时使用 `scanf()` 的两种不同的特性。首先，如果函数调用成功，`scanf()` 会把一个值存入 `num`。然后，利用 `scanf()` 的返回值（0 或 1，不是 `num` 的值）控制 `while` 循环。因为每次迭代都会判断循环的条件，所以每次迭代都要调用 `scanf()` 读取新的 `num` 值来做判断。换句话说，C 的语法特性让你可以用下面的精简版本替换标准版本：

当获取值和判断值都成功

处理该值

接下来，我们正式地学习 `while` 语句。

6.2 while 语句

`while` 循环的通用形式如下：

```
while ( expression )
    statement
```

`statement` 部分可以是以分号结尾的简单语句，也可以是用花括号括起来的复合语句。

到目前为止，程序示例中的 `expression` 部分都使用关系表达式。也就是说，`expression` 是值之间的比较，可以使用任何表达式。如果 `expression` 为真（或者更一般地说，非零），执行 `statement` 部分一次，然后再次判断 `expression`。在 `expression` 为假（0）之前，循环的判断和执行一直重复进行。每次循环都被称为一次迭代（iteration），如图 6.1 所示。

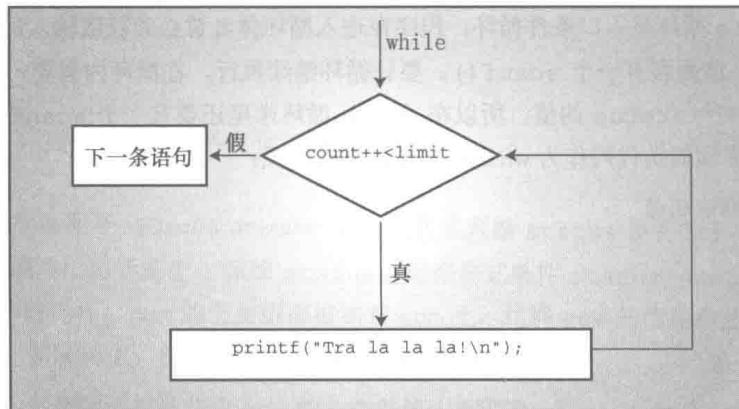


图 6.1 `while` 循环的结构

6.2.1 终止 `while` 循环

`while` 循环有一点非常重要：在构建 `while` 循环时，必须让测试表达式的值有变化，表达式最终要为假。否则，循环就不会终止（实际上，可以使用 `break` 和 `if` 语句来终止循环，但是你尚未学到）。考虑下面的例子：

```
index = 1;
while (index < 5)
    printf("Good morning!\\n");
```

上面的程序段将打印无数次 `Good morning!`。为什么？因为循环中 `index` 的值一直都是原来的值 1，不曾变过。现在，考虑下面的程序段：

```
index = 1;
while (--index < 5)
    printf("Good morning!\\n");
```

这段程序也好不到哪里去。虽然改变了 `index` 的值，但是改错了！不过，这个版本至少在 `index` 减

少到其类型到可容纳的最小负值并变成最大正值时会终止循环(第3章3.4.2节中的`toobig.c`程序解释过，最大正值加1一般会得到一个负值；类似地，最小负值减1一般会得到最大正值)。

6.2.2 何时终止循环

要明确一点：只有在对测试条件求值时，才决定是终止还是继续循环。例如，考虑程序清单6.2中的程序。

程序清单 6.2 when.c 程序

```
// when.c -- 何时退出循环
#include <stdio.h>
int main(void)
{
    int n = 5;

    while (n < 7)           // 第 7 行
    {
        printf("n = %d\n", n);
        n++;                 // 第 10 行
        printf("Now n = %d\n", n); // 第 11 行
    }
    printf("The loop has finished.\n");

    return 0;
}
```

运行程序清单6.2，输出如下：

```
n = 5
Now n = 6
n = 6
Now n = 7
The loop has finished.
```

在第2次循环时，变量n在第10行首次获得值7。但是，此时程序并未退出，它结束本次循环（第11行），并在对第7行的测试条件求值时才退出循环（变量n在第1次判断时为5，第2次判断时为6）。

6.2.3 while：入口条件循环

while循环是使用入口条件的有条件循环。所谓“有条件”指的是语句部分的执行取决于测试表达式描述的条件，如(`index < 5`)。该表达式是一个入口条件(*entry condition*)，因为必须满足条件才能进入循环体。在下面的情况下，就不会进入循环体，因为条件一开始就为假：

```
index = 10;
while (index++ < 5)
    printf("Have a fair day or better.\n");
```

把第1行改为：

```
index = 3;
```

就可以运行这个循环了。

6.2.4 语法要点

使用while时，要牢记一点：只有在测试条件后面的单独语句（简单语句或复合语句）才是循环部分。程序清单6.3演示了忽略这点的后果。缩进是为了让读者阅读方便，不是计算机的要求。

程序清单 6.3 while1.c 程序

```
/* while1.c -- 注意花括号的使用 */
/* 糟糕的代码创建了一个无限循环 */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n < 3)
        printf("n is %d\n", n);
        n++;
    printf("That's all this program does\n");

    return 0;
}
```

该程序的输出如下：

```
n is 0
...
...
```

屏幕上会一直输出以上内容，除非强行关闭这个程序。

虽然程序中缩进了 `n++;` 这条语句，但是并未把它和上一条语句括在花括号内。因此，只有直接跟在测试条件后面的一条语句是循环的一部分。变量 `n` 的值不会改变，条件 `n < 3` 一直为真。该循环会一直打印 `n is 0`，除非强行关闭程序。这是一个无限循环 (*infinite loop*) 的例子，没有外部干涉就不会退出。

记住，即使 `while` 语句本身使用复合语句，在语句构成上，它也是一条单独的语句。该语句从 `while` 开始执行，到第 1 个分号结束。在使用了复合语句的情况下，到右花括号结束。

要注意放置分号的位置。例如，考虑程序清单 6.4。

程序清单 6.4 while2.c 程序

```
/* while2.c -- 注意分号的位置 */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n++ < 3); /* 第 7 行 */
        printf("n is %d\n", n); /* 第 8 行 */
    printf("That's all this program does.\n");

    return 0;
}
```

该程序的输出如下：

```
n is 4
That's all this program does.
```

如前所述，循环在执行完测试条件后面的第 1 条语句（简单语句或复合语句）后进入下一轮迭代，直到测试条件为假才会结束。该程序中第 7 行的测试条件后面直接跟着一个分号，循环在此进入下一轮迭代，因为单独一个分号被视为一条语句。虽然 n 的值在每次循环时都递增 1，但是第 8 行的语句不是循环的一部分，因此只会打印一次循环结束后的 n 值。

在该例中，测试条件后面的单独分号是空语句 (*null statement*)，它什么也不做。在 C 语言中，单独的分号表示空语句。有时，程序员会故意使用带空语句的 while 语句，因为所有的任务都在测试条件中完成了，不需要在循环体中做什么。例如，假设你想跳过输入到第 1 个非空白字符或数字，可以这样写：

```
while (scanf("%d", &num) == 1)
    /* 跳过整数输入 */
```

只要 scanf() 读取一个整数，就会返回 1，循环继续执行。注意，为了提高代码的可读性，应该让这个分号独占一行，不要直接把它放在测试表达式同行。这样做一方面让读者更容易看到空语句，一方面也提醒自己和读者空语句是有意而为之。处理这种情况更好的方法是使用下一章介绍的 continue 语句。

6.3 用关系运算符和表达式比较大小

while 循环经常依赖测试表达式作比较，这样的表达式被称为关系表达式 (*relational expression*)，出现在关系表达式中间的运算符叫做关系运算符 (*relational operator*)。前面的示例中已经用过一些关系运算符，表 6.1 列出了 C 语言的所有关系运算符。该表也涵盖了所有的数值关系（数字之间的关系再复杂也没有人与人之间的关系复杂）。

表 6.1 关系运算符

运算符	含义
<	小于
<=	小于或等于
==	等于
>=	大于或等于
>	大于
!=	不等于

关系运算符常用于构造 while 语句和其他 C 语句（稍后讨论）中用到的关系表达式。这些语句都会检查关系表达式为真还是为假。下面有 3 个互不相关的 while 语句，其中都包含关系表达式。

```
while (number < 6)
{
    printf("Your number is too small.\n");
    scanf("%d", &number);
}

while (ch != '$')
{
    count++;
    scanf("%c", &ch);
}

while (scanf("%f", &num) == 1)
    sum = sum + num;
```

注意，第2个while语句的关系表达式还可用于比较字符。比较时使用的是机器字符码(假定为ASCII)。但是，不能用关系运算符比较字符串。第11章将介绍如何比较字符串。

虽然关系运算符也可用来比较浮点数，但是要注意：比较浮点数时，尽量只使用<和>。因为浮点数的舍入误差会导致在逻辑上应该相等的两数却不相等。例如，3乘以 $1/3$ 的积是1.0。如果用把 $1/3$ 表示成小数点后面6位数字，乘积则是.999999，不等于1。使用fabs()函数(声明在math.h头文件中)可以方便地比较浮点数，该函数返回一个浮点值的绝对值(即，没有代数符号的值)。例如，可以用类似程序清单6.5的方法来判断一个数是否接近预期结果。

程序清单 6.5 cmpflt.c 程序

```
// cmpflt.c -- 浮点数比较
#include <math.h>
#include <stdio.h>
int main(void)
{
    const double ANSWER = 3.14159;
    double response;

    printf("What is the value of pi?\n");
    scanf("%lf", &response);
    while (fabs(response - ANSWER) > 0.0001)
    {
        printf("Try again!\n");
        scanf("%lf", &response);
    }
    printf("Close enough!\n");

    return 0;
}
```

循环会一直提示用户继续输入，除非用户输入的值与正确值之间相差0.0001：

```
What is the value of pi?
3.14
Try again!
3.1416
Close enough!
```

6.3.1 什么是真

这是一个古老的问题，但是对C而言还不算难。在C中，表达式一定有一个值，关系表达式也不例外。程序清单6.6中的程序用于打印两个关系表达式的值，一个为真，一个为假。

程序清单 6.6 t_and_f.c 程序

```
/* t_and_f.c -- C 中的真和假的值 */
#include <stdio.h>
int main(void)
{
    int true_val, false_val;

    true_val = (10 > 2);           // 关系为真的值
    false_val = (10 == 2); // 关系为假的值
    printf("true = %d; false = %d \n", true_val, false_val);
```

```

    return 0;
}

```

程序清单 6.6 把两个关系表达式的值分别赋给两个变量，即把表达式为真的值赋给 `true_val`，表达式为假的值赋给 `false_val`。运行该程序后输出如下：

```
true = 1; false = 0
```

原来如此！对 C 而言，表达式为真的值是 1，表达式为假的值是 0。一些 C 程序使用下面的循环结构，由于 1 为真，所以循环会一直进行。

```

while (1)
{
    ...
}
```

6.3.2 其他真值

既然 1 或 0 可以作为 `while` 语句的测试表达式，是否还可以使用其他数字？如果可以，会发生什么？我们用程序清单 6.7 来做个实验。

程序清单 6.7 `truth.c` 程序

```

// truth.c -- 哪些值为真
#include <stdio.h>
int main(void)
{
    int n = 3;

    while (n)
        printf("%2d is true\n", n--);
    printf("%2d is false\n", n);

    n = -3;
    while (n)
        printf("%2d is true\n", n++);
    printf("%2d is false\n", n);

    return 0;
}

```

该程序的输出如下：

```

3 is true
2 is true
1 is true
0 is false
-3 is true
-2 is true
-1 is true
0 is false

```

执行第 1 个循环时，`n` 分别是 3、2、1，当 `n` 等于 0 时，第 1 个循环结束。与此类似，执行第 2 个循环时，`n` 分别是 -3、-2 和 -1，当 `n` 等于 0 时，第 2 个循环结束。一般而言，所有的非零值都视为真，只有 0 被视为假。在 C 中，真的概念还真宽！

也可以说，只要测试条件的值为非零，就会执行 `while` 循环。这是从数值方面而不是从真/假方面来

看测试条件。要牢记：关系表达式为真，求值得 1；关系表达式为假，求值得 0。因此，这些表达式实际上相当于数值。

许多 C 程序员都会很好地利用测试条件的这一特性。例如，用 `while (goats)` 替换 `while (goats != 0)`，因为表达式 `goats != 0` 和 `goats` 都只有在 `goats` 的值为 0 时才为 0 或假。第 1 种形式 (`while (goats != 0)`) 对初学者而言可能比较清楚，但是第 2 种形式 (`while (goats)`) 才是 C 程序员最常用的。要想成为一名 C 程序员，应该多熟悉 `while (goats)` 这种形式。

6.3.3 真值的问题

C 对真的概念约束太少会带来一些麻烦。例如，我们稍微修改一下程序清单 6.1，修改后的程序如程序清单 6.8 所示。

程序清单 6.8 trouble.c 程序

```
// trouble.c -- 误用=会导致无限循环

#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status = 1)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

运行该程序，其输出如下：

```
Please enter an integer to be summed (q to quit): 20
Please enter next integer (q to quit): 5
Please enter next integer (q to quit): 30
Please enter next integer (q to quit): q
Please enter next integer (q to quit):
```

(……屏幕上会一直显示最后的提示内容，除非强行关闭程序。也许你根本不想运行这个示例。)

这个麻烦的程序示例改动了 `while` 循环的测试条件，把 `status == 1` 替换成 `status = 1`。后者是一个赋值表达式语句，所以 `status` 的值为 1。而且，整个赋值表达式的值就是赋值运算符左侧的值，所以 `status = 1` 的值也是 1。这里，`while (status = 1)` 实际上相当于 `while (1)`，也就是说，循环不会退出。虽然用户输入 `q`，`status` 被设置为 0，但是循环的测试条件把 `status` 又重置为 1，进入了

下一次迭代。

读者可能不太理解，程序的循环一直运行着，用户在输入 q 后完全没机会继续输入。如果 `scanf()` 读取指定形式的输入失败，就把无法读取的输入留在输入队列中，供下次读取。当 `scanf()` 把 q 作为整数读取时失败了，它把 q 留下。在下次循环时，`scanf()` 从上次读取失败的地方 (q) 开始读取，`scanf()` 把 q 作为整数读取，又失败了。因此，这样修改后不仅创建了一个无限循环，还创建了一个无限失败的循环，真让人沮丧。好在计算机觉察不出来。对计算机而言，无限地执行这些愚蠢的指令比成功预测未来 10 年的股市行情没什么两样。

不要在本应使用`==`的地方使用`=`。一些计算机语言（如，BASIC）用相同的符号表示赋值运算符和关系相等运算符，但是这两个运算符完全不同（见图 6.2）。赋值运算符把一个值赋给它左侧的变量；而关系相等运算符检查它左侧和右侧的值是否相等，不会改变左侧变量的值（如果左侧是一个变量）。



图 6.2 关系运算符`==`和赋值运算符`=`

示例如下：

<code>canoes = 5</code>	← 把 5 赋给 canoes
<code>canoes == 5</code>	← 检查 canoes 的值是否为 5

要注意使用正确的运算符。编译器不会检查出你使用了错误的形式，得出也不是预期的结果（误用`=`的人实在太多了，以至于现在大多数编译器都会给出警告，提醒用户是否要这样做）。如果待比较的一个值是常量，可以把该常量放在左侧有助于编译器捕获错误：

<code>5 = canoes</code>	← 语法错误
<code>5 == canoes</code>	← 检查 canoes 的值是否为 5

可以这样做是因为 C 语言不允许给常量赋值，编译器会把赋值运算符的这种用法作为语法错误标记出来。许多经验丰富的程序员在构建比较是否相等的表达式时，都习惯把常量放在左侧。

总之，关系运算符用于构成关系表达式。关系表达式为真时值为 1，为假时值为 0。通常用关系表达式作为测试条件的语句（如 `while` 和 `if`）可以使用任何表达式作为测试条件，非零为真，零为假。

6.3.4 新的`_Bool`类型

在 C 语言中，一直用 `int` 类型的变量表示真/假值。C99 专门针对这种类型的变量新增了`_Bool` 类型。该类型是以英国数学家 George Boole 的名字命名的，他开发了用代数表示逻辑和解决逻辑问题。在编程中，表示真或假的变量被称为布尔变量 (*Boolean variable*)，所以`_Bool` 是 C 语言中布尔变量的类型名。`_Bool` 类型的变量只能储存 1 (真) 或 0 (假)。如果把其他非零数值赋给`_Bool` 类型的变量，该变量会被设置为 1。这反映了 C 把所有的非零值都视为真。

程序清单 6.9 修改了程序清单 6.8 中的测试条件，把 int 类型的变量 status 替换为 _Bool 类型的变量 input_is_good。给布尔变量取一个能表示真或假值的变量名是一种常见的做法。

程序清单 6.9 boolean.c 程序

```
// boolean.c -- 使用 _Bool 类型的变量 variable
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    _Bool input_is_good;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    input_is_good = (scanf("%ld", &num) == 1);
    while (input_is_good)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        input_is_good = (scanf("%ld", &num) == 1);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

注意程序中把比较的结果赋值给 _Bool 类型的变量 input_is_good:

```
input_is_good = (scanf("%ld", &num) == 1);
```

这样做没问题，因为==运算符返回的值不是 1 就是 0。顺带一提，从优先级方面考虑的话，并不需要用圆括号把 scanf("%ld", &num) == 1 括起来。但是，这样做可以提高代码可读性。还要注意，如何为变量命名才能让 while 循环的测试简单易懂：

```
while (input_is_good)
```

C99 提供了 stdbool.h 头文件，该头文件让 bool 成为 _Bool 的别名，而且还把 true 和 false 分别定义为 1 和 0 的符号常量。包含该头文件后，写出的代码可以与 C++ 兼容，因为 C++ 把 bool、true 和 false 定义为关键字。

如果系统不支持 _Bool 类型，导致无法运行该程序，可以把 _Bool 替换成 int 即可。

6.3.5 优先级和关系运算符

关系运算符的优先级比算术运算符（包括+和-）低，比赋值运算符高。这意味着 $x > y + 2$ 和 $x > (y + 2)$ 相同， $x = y > 2$ 和 $x = (y > 2)$ 相同。换言之，如果 y 大于 2，则给 x 赋值 1，否则赋值 0。y 的值不会赋给 x。

关系运算符比赋值运算符的优先级高，因此， $x_bigger = x > y$; 相当于 $x_bigger = (x > y);$ 。

关系运算符之间有两种不同的优先级。

高优先级组: <<= >>=

低优先级组: == !=

与其他大多数运算符一样，关系运算符的结合律也是从左往右。因此：

`ex != wye == zee` 与 `(ex != wye) == zee` 相同

首先, C 判断 `ex` 与 `wye` 是否相等; 然后, 用得出的值 1 或 0 (真或假) 再与 `zee` 比较。我们并不推荐这样写, 但是在这里有必要说明一下。

表 6.2 列出了目前我们学过的运算符的性质。附录 B 的参考资料 II “C 运算符” 中列出了全部运算符的完整优先级表。

表 6.2 运算符优先级

运算符 (优先级从高至低)	结合律
<code>()</code>	从左往右
<code>- + ++ -- sizeof</code>	从右往左
<code>* / %</code>	从左往右
<code>+ -</code>	从左往右
<code>< > <= >=</code>	从左往右
<code>== !=</code>	从左往右
<code>=</code>	从右往左

小结: while 语句

关键字: `while`

一般注解:

`while` 语句创建了一个循环, 重复执行直到测试表达式为假或 0。`while` 语句是一种入口条件循环, 也就是说, 在执行多次循环之前已决定是否执行循环。因此, 循环有可能不被执行。循环体可以是简单语句, 也可以是复合语句。

形式:

```
while ( expression )
    statement
```

在 `expression` 部分为假或 0 之前, 重复执行 `statement` 部分。

示例:

```
while (n++ < 100)
    printf("%d %d\n", n, 2 * n + 1); // 简单语句
while (fargo < 1000)
{ // 复合语句
    fargo = fargo + step;
    step = 2 * step;
}
```

小结: 关系运算符和表达式

关系运算符:

每个关系运算符都把它左侧的值和右侧的值进行比较。

`<` 小于

`<=` 小于或等于

<code>==</code>	等于
<code>>=</code>	大于或等于
<code>></code>	大于
<code>!=</code>	不等于

关系表达式：

简单的关系表达式由关系运算符及其运算对象组成。如果关系为真，关系表达式的值为 1；如果关系为假，关系表达式的值为 0。

示例：

`5 > 2` 为真，关系表达式的值为 1

`(2 + a) == a` 为假，关系表达式的值为 0

6.4 不确定循环和计数循环

一些 `while` 循环是不确定循环 (*indefinite loop*)。所谓不确定循环，指在测试表达式为假之前，预先不知道要执行多少次循环。例如，程序清单 6.1 通过与用户交互获得数据来计算整数之和。我们事先并不知道用户会输入什么整数。另外，还有一类是计数循环 (*counting loop*)。这类循环在执行循环之前就知道要重复执行多少次。程序清单 6.10 就是一个简单的计数循环。

程序清单 6.10 sweetiel.c 程序

```
// sweetiel.c -- 一个计数循环
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count = 1; // 初始化

    while (count <= NUMBER) // 测试
    {
        printf("Be my Valentine!\n"); // 行为
        count++; // 更新计数
    }

    return 0;
}
```

虽然程序清单 6.10 运行情况良好，但是定义循环的行为并未组织在一起，程序的编排并不是很理想。我们来仔细分析一下。

在创建一个重复执行固定次数的循环中涉及了 3 个行为：

1. 必须初始化计数器；
2. 计数器与有限的值作比较；
3. 每次循环时递增计数器。

`while` 循环的测试条件执行比较，递增运算符执行递增。程序清单 6.10 中，递增发生在循环的末尾，这可以防止不小心漏掉递增。因此，这样做比将测试和更新组合放在一起（即使用 `count++ <= NUMBER`）要好，但是计数器的初始化放在循环外，就有可能忘记初始化。实践告诉我们可能会发生的事情终究会发生，所以我们来学习另一种控制语句，可以避免这些问题。

6.5 for 循环

for 循环把上述 3 个行为（初始化、测试和更新）组合在一处。程序清单 6.11 使用 for 循环修改了程序清单 6.10 的程序。

程序清单 6.11 sweetie2.c 程序

```
// sweetie2.c -- 使用 for 循环的计数循环
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count;

    for (count = 1; count <= NUMBER; count++)
        printf("Be my Valentine!\n");

    return 0;
}
```

关键字 for 后面的圆括号中有 3 个表达式，分别用两个分号隔开。第 1 个表达式是初始化，只会在 for 循环开始时执行一次。第 2 个表达式是测试条件，在执行循环之前对表达式求值。如果表达式为假（本例中，count 大于 NUMBER 时），循环结束。第 3 个表达式执行更新，在每次循环结束时求值。程序清单 6.10 用这个表达式递增 count 的值，更新计数。完整的 for 语句还包括后面的简单语句或复合语句。for 圆括号中的表达式也叫做控制表达式，它们都是完整表达式，所以每个表达式的副作用（如，递增变量）都发生在对下一个表达式求值之前。图 6.3 演示了 for 循环的结构。

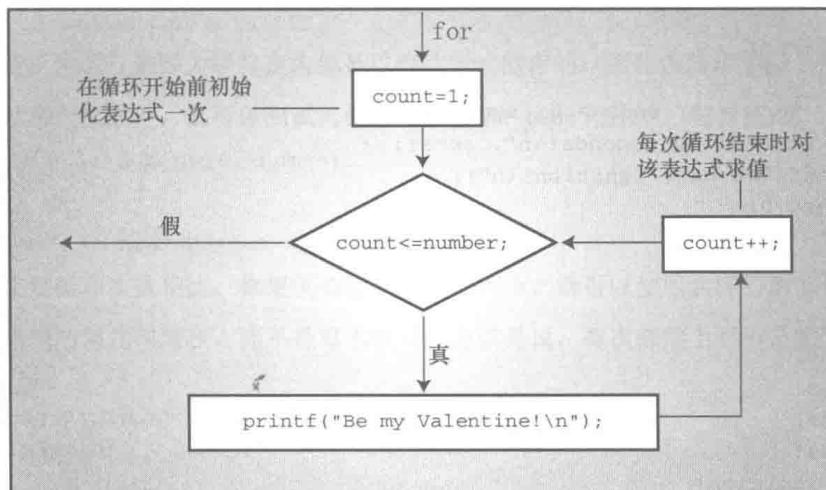


图 6.3 for 循环的结构

程序清单 6.12 for_cube.c 程序

```
/* for_cube.c -- 使用 for 循环创建一个立方表 */
#include <stdio.h>
int main(void)
{
    int num;

    printf("    n    n cubed\n");
}
```

```

for (num = 1; num <= 6; num++)
    printf("%5d %5d\n", num, num*num*num);

return 0;
}

```

程序清单 6.12 打印整数 1~6 及其对应的立方，该程序的输出如下：

n	n cubed
1	1
2	8
3	27
4	64
5	125
6	216

for 循环的第 1 行包含了循环所需的所有信息：num 的初值¹和每次循环 num 的增量。

6.5.1 利用 for 的灵活性

虽然 for 循环看上去和 FORTRAN 的 DO 循环、Pascal 的 FOR 循环、BASIC 的 FOR...NEXT 循环类似，但是 for 循环比这些循环灵活。这些灵活性源于如何使用 for 循环中的 3 个表达式。以前面程序示例中的 for 循环为例，第 1 个表达式给计数器赋初值，第 2 个表达式表示计数器的范围，第 3 个表达式递增计数器。这样使用 for 循环确实很像其他语言的循环。除此之外，for 循环还有其他 9 种用法。

- 可以使用递减运算符来递减计数器：

```

/* for_down.c */
#include <stdio.h>
int main(void)
{
    int secs;

    for (secs = 5; secs > 0; secs--)
        printf("%d seconds!\n", secs);
    printf("We have ignition!\n");
    return 0;
}

```

该程序输出如下：

```

5 seconds!
4 seconds!
3 seconds!
2 seconds!
1 seconds!
We have ignition!

```

- 可以让计数器递增 2、10 等：

```

/* for_13s.c */
#include <stdio.h>
int main(void)
{
    int n; // 从 2 开始，每次递增 13

```

¹ 其实 num 的最终值不是 6，而是 7。虽然最后一次循环打印的 num 值是 6，但随后 num++使 num 的值为 7，然后 num <= 6 为假，for 循环结束。——译者注

```

    for (n = 2; n < 60; n = n + 13)
        printf("%d \n", n);
    return 0;
}

```

每次循环 n 递增 13，程序的输出如下：

```

2
15
28
41
54

```

- 可以用字符代替数字计数：

```

/* for_char.c */
#include <stdio.h>
int main(void)
{
    char ch;

    for (ch = 'a'; ch <= 'z'; ch++)
        printf("The ASCII value for %c is %d.\n", ch, ch);
    return 0;
}

```

该程序假定系统用 ASCII 码表示字符。由于篇幅有限，省略了大部分输出：

```

The ASCII value for a is 97.
The ASCII value for b is 98.

...
The ASCII value for x is 120.
The ASCII value for y is 121.
The ASCII value for z is 122.

```

该程序能正常运行是因为字符在内部是以整数形式储存的，因此该循环实际上仍是用整数来计数。

- 除了测试迭代次数外，还可以测试其他条件。在 for_cube 程序中，可以把：

```
for (num = 1; num <= 6; num++)
```

替换成：

```
for (num = 1; num*num*num <= 216; num++)
```

如果与控制循环次数相比，你更关心限制立方的大小，就可以使用这样的测试条件。

- 可以让递增的量几何增长，而不是算术增长。也就是说，每次都乘上而不是加上一个固定的量：

```

/* for_geo.c */
#include <stdio.h>
int main(void)
{
    double debt;
    for (debt = 100.0; debt < 150.0; debt = debt * 1.1)
        printf("Your debt is now $%.2f.\n", debt);
    return 0;
}

```

该程序中，每次循环都把 debt 乘以 1.1，即 debt 的值每次都增加 10%，其输出如下：

```

Your debt is now $100.00.
Your debt is now $110.00.
Your debt is now $121.00.
Your debt is now $133.10.

```

Your debt is now \$146.41.

- 第3个表达式可以使用任意合法的表达式。无论是什么表达式，每次迭代都会更新该表达式的值。

```
/* for_wild.c */
#include <stdio.h>
int main(void)
{
    int x;
    int y = 55;

    for (x = 1; y <= 75; y = (++x * 5) + 50)
        printf("%10d %10d\n", x, y);
    return 0;
}
```

该循环打印 x 的值和表达式 $++x * 5 + 50$ 的值，程序的输出如下：

```
1      55
2      60
3      65
4      70
5      75
```

注意，测试涉及 y，而不是 x。for 循环中的3个表达式可以是不同的变量（注意，虽然该例可以正常运行，但是编程风格不太好。如果不在更新部分加入代数计算，程序会更加清楚）。

- 可以省略一个或多个表达式（但是不能省略分号），只要在循环中包含能结束循环的语句即可。

```
/* for_none.c */
#include <stdio.h>
int main(void)
{
    int ans, n;
    ans = 2;
    for (n = 3; ans <= 25;)
        ans = ans * n;
    printf("n = %d; ans = %d.\n", n, ans);
    return 0;
}
```

该程序的输出如下：

```
n = 3; ans = 54.
```

该循环保持 n 的值为 3。变量 ans 开始的值为 2，然后递增到 6 和 18，最终是 54（18 比 25 小，所以 for 循环进入下一次迭代，18 乘以 3 得 54）。顺带一提，省略第2个表达式被视为真，所以下面的循环会一直运行：

```
for ( ; ; )
    printf("I want some action\n");
```

- 第1个表达式不一定是给变量赋初值，也可以使用 printf()。记住，在执行循环的其他部分之前，只对第1个表达式求值一次或执行一次。

```
/* for_show.c */
#include <stdio.h>
int main(void)
{
    int num = 0;

    for (printf("Keep entering numbers!\n"); num != 6;)
        scanf("%d", &num);
```

```

    printf("That's the one I want!\n");
    return 0;
}

```

该程序打印第 1 行的句子一次，在用户输入 6 之前不断接受数字：

```
Keep entering numbers!
```

```
3
```

```
5
```

```
8
```

```
6
```

```
That's the one I want!
```

- 循环体中的行为可以改变循环头中的表达式。例如，假设创建了下面的循环：

```
for (n = 1; n < 10000; n = n + delta)
```

如果程序经过几次迭代后发现 `delta` 太小或太大，循环中的 `if` 语句（详见第 7 章）可以改变 `delta` 的大小。在交互式程序中，用户可以在循环运行时才改变 `delta` 的值。这样做也有危险的一面，例如，把 `delta` 设置为 0 就没用了。

总而言之，可以自己决定如何使用 `for` 循环头中的表达式，这使得在执行固定次数的循环外，还可以做更多的事情。接下来，我们将简要讨论一些运算符，使 `for` 循环更加有用。

小结：for 语句

关键字：for

一般注解：

`for` 语句使用 3 个表达式控制循环过程，分别用分号隔开。`initialize` 表达式在执行 `for` 语句之前只执行一次；然后对 `test` 表达式求值，如果表达式为真（或非零），执行循环一次；接着对 `update` 表达式求值，并再次检查 `test` 表达式。`for` 语句是一种入口条件循环，即在执行循环之前就决定了是否执行循环。因此，`for` 循环可能一次都不执行。`statement` 部分可以是一条简单语句或复合语句。

形式：

```
for ( initialize; test; update )
    statement
```

在 `test` 为假或 0 之前，重复执行 `statement` 部分。

示例：

```
for (n = 0; n < 10 ; n++)
    printf("%d %d\n", n, 2 * n + 1);
```

6.6 其他赋值运算符：+=、-=、*=、/=、%≡

C 有许多赋值运算符。最基本、最常用的是`=`，它把右侧表达式的值赋给左侧的变量。其他赋值运算符都用于更新变量，其用法都是左侧是一个变量名，右侧是一个表达式。赋给变量的新值是根据右侧表达式的值调整后的值。确切的调整方案取决于具体的运算符。例如：

<code>scores += 20</code>	与	<code>scores = scores + 20</code>	相同
<code>dimes -= 2</code>	与	<code>dimes = dimes - 2</code>	相同
<code>bunnies *= 2</code>	与	<code>bunnies = bunnies * 2</code>	相同
<code>time /= 2.73</code>	与	<code>time = time / 2.73</code>	相同
<code>reduce %= 3</code>	与	<code>reduce = reduce % 3</code>	相同

上述所列的运算符右侧都使用了简单的数，还可以使用更复杂的表达式，例如：

`x *= 3 * y + 12` 与 `x = x * (3 * y + 12)` 相同

以上提到的赋值运算符与=的优先级相同，即比+或*优先级低。上面最后一个例子也反映了赋值运算符的优先级，`3 * y` 先与 12 相加，再把计算结果与 `x` 相乘，最后再把乘积赋给 `x`。

并非一定要使用这些组合形式的赋值运算符。但是，它们让代码更紧凑，而且与一般形式相比，组合形式的赋值运算符生成的机器代码更高效。当需要在 `for` 循环中塞进一些复杂的表达式时，这些组合的赋值运算符特别有用。

6.7 逗号运算符

逗号运算符扩展了 `for` 循环的灵活性，以便在循环头中包含更多的表达式。例如，程序清单 6.13 演示了一个打印一类邮件资费 (*first-class postage rate*) 的程序（在撰写本书时，邮资为首重 40 美分/盎司，续重 20 美分/盎司，可以在互联网上查看当前邮资）。

程序清单 6.13 `postage.c` 程序

```
// postage.c -- 一类邮资
#include <stdio.h>
int main(void)
{
    const int FIRST_OZ = 46;      // 2013 邮资
    const int NEXT_OZ = 20;       // 2013 邮资
    int ounces, cost;

    printf(" ounces  cost\n");
    for (ounces = 1, cost = FIRST_OZ; ounces <= 16; ounces++, cost += NEXT_OZ)
        printf("%5d  $%4.2f\n", ounces, cost / 100.0);

    return 0;
}
```

该程序的前 5 行输出如下：

ounces	cost
1	\$0.46
2	\$0.66
3	\$0.86
4	\$1.06

该程序在初始化表达式和更新表达式中使用了逗号运算符。初始化表达式中的逗号使 `ounces` 和 `cost` 都进行了初始化，更新表达式中的逗号使每次迭代 `ounces` 递增 1、`cost` 递增 20 (`NEXT_OZ` 的值是 20)。绝大多数计算都在 `for` 循环头中进行（见图 6.4）。

逗号运算符并不局限于在 `for` 循环中使用，但是这是它最常用的地方。逗号运算符有两个其他性质。首先，它保证了被它分隔的表达式从左往右求值（换言之，逗号是一个序列点，所以逗号左侧项的所有副作用都在程序执行逗号右侧项之前发生）。因此，`ounces` 在 `cost` 之前被初始化。在该例中，顺序并不重要，但是如果 `cost` 的表达式中包含了 `ounces` 时，顺序就很重要。例如，假设有下面的表达式：

`ounces++, cost = ounces * FIRST_OZ`

在该表达式中，先递增 `ounce`，然后在第 2 个子表达式中使用 `ounce` 的新值。作为序列点的逗号保证了左侧子表达式的副作用在对右侧子表达式求值之前发生。

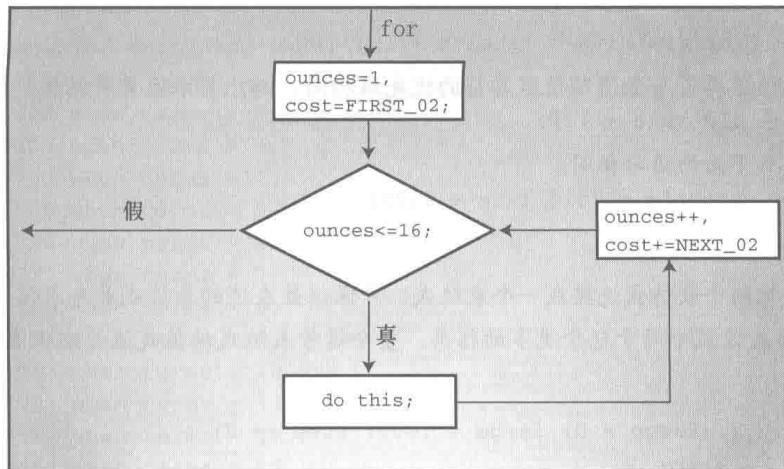


图 6.4 逗号运算符和 for 循环

其次，整个逗号表达式的值是右侧项的值。例如，下面语句

```
x = (y = 3, (z = ++y + 2) + 5);
```

的效果是：先把 3 赋给 y，递增 y 为 4，然后把 4 加 2 之和 (6) 赋给 z，接着加上 5，最后把结果 11 赋给 x。至于为什么有人编写这样的代码，在此不做评价。另一方面，假设在写数字时不小心输入了逗号：

```
houseprice = 249,500;
```

这不是语法错误，C 编译器会将其解释为一个逗号表达式，即 houseprice = 249 是逗号左侧的子表达式，500 是右侧的子表达式。因此，整个逗号表达式的值是逗号右侧表达式的值，而且左侧的赋值表达式把 249 赋给变量 houseprice。因此，这与下面代码的效果相同：

```
houseprice = 249;  
500;
```

记住，任何表达式后面加上一个分号就成了表达式语句。所以，500; 也是一条语句，但是什么也不做。

另外，下面的语句

```
houseprice = (249,500);
```

赋给 houseprice 的值是逗号右侧子表达式的值，即 500。

逗号也可用作分隔符。在下面语句中的逗号都是分隔符，不是逗号运算符：

```
char ch, date;  
printf("%d %d\n", chimps, chumps);
```

小结：新的运算符

赋值运算符：

下面的运算符用右侧的值，根据指定的操作更新左侧的变量：

- $+=$ 把右侧的值加到左侧的变量上
- $-=$ 从左侧的变量中减去右侧的值
- $*=$ 把左侧的变量乘以右侧的值
- $/=$ 把左侧的变量除以右侧的值
- $\%=$ 左侧变量除以右侧值得到的余数

示例：

```
rabbits *= 1.6; 与 rabbits = rabbits * 1.6; 相同
```

这些组合赋值运算符与普通赋值运算符的优先级相同，都比算术运算符的优先级低。因此，
contents *= old_rate + 1.2;

最终的效果与下面的语句相同：

```
contents = contents * (old_rate + 1.2);
```

逗号运算符：

逗号运算符把两个表达式连接成一个表达式，并保证最左边的表达式最先求值。逗号运算符通常在 for 循环头的表达式中用于包含更多的信息。整个逗号表达式的值是逗号右侧表达式的值。

示例：

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
    fargo += step;
```

6.7.1 当 Zeno 遇到 for 循环

接下来，我们看看 for 循环和逗号运算符如何解决古老的悖论。希腊哲学家 Zeno 曾经提出箭永远不会达到它的目标。首先，他认为箭要到达目标距离的一半，然后再达到剩余距离的一半，然后继续到达剩余距离的一半，这样就无穷无尽。Zeno 认为箭的飞行过程有无数个部分，所以要花费无数时间才能结束这一过程。不过，我们怀疑 Zeno 是自愿甘做靶子才会得出这样的结论。

我们采用一种定量的方法，假设箭用 1 秒钟走完一半的路程，然后用 $1/2$ 秒走完剩余距离的一半，然后用 $1/4$ 秒再走完剩余距离的一半，等等。可以用下面的无限序列来表示总时间：

$1 + 1/2 + 1/4 + 1/8 + 1/16 + \dots$

程序清单 6.14 中的程序求出了序列前几项的和。变量 power_of_two 的值分别是 1.0、2.0、4.0、8.0 等。

程序清单 6.14 zeno.c 程序

```
/* zeno.c -- 求序列的和 */
#include <stdio.h>

int main(void)
{
    int t_ct;          // 项计数
    double time, power_of_2;
    int limit;

    printf("Enter the number of terms you want: ");
    scanf("%d", &limit);
    for (time = 0, power_of_2 = 1, t_ct = 1; t_ct <= limit;
         t_ct++, power_of_2 *= 2.0)
    {
        time += 1.0 / power_of_2;
        printf("time = %f when terms = %d.\n", time, t_ct);
    }

    return 0;
}
```

下面是序列前 15 项的和:

```
Enter the number of terms you want: 15
time = 1.000000 when terms = 1.
time = 1.500000 when terms = 2.
time = 1.750000 when terms = 3.
time = 1.875000 when terms = 4.
time = 1.937500 when terms = 5.
time = 1.968750 when terms = 6.
time = 1.984375 when terms = 7.
time = 1.992188 when terms = 8.
time = 1.996094 when terms = 9.
time = 1.998047 when terms = 10.
time = 1.999023 when terms = 11.
time = 1.999512 when terms = 12.
time = 1.999756 when terms = 13.
time = 1.999878 when terms = 14.
time = 1.999939 when terms = 15.
```

不难看出, 尽管不断添加新的项, 但是总和看起来变化不大。就像程序输出显示的那样, 数学家的确证明了当项的数目接近无穷时, 总和无限接近 2.0。假设 s 表示总和, 下面我们用数学的方法来证明一下:

$$S = 1 + 1/2 + 1/4 + 1/8 + \dots$$

这里的省略号表示“等等”。把 S 除以 2 得:

$$S/2 = 1/2 + 1/4 + 1/8 + 1/16 + \dots$$

第 1 个式子减去第 2 个式子得:

$$S - S/2 = 1 + 1/2 - 1/2 + 1/4 - 1/4 + \dots$$

除了第 1 个值为 1, 其他的值都是一正一负地成对出现, 所以这些项都可以消去。只留下:

$$S/2 = 1$$

然后, 两侧同乘以 2, 得:

$$S = 2$$

从这个示例中得到的启示是, 在进行复杂的计算之前, 先看看数学上是否有简单的方法可用。

程序本身是否有需要注意的地方? 该程序演示了在表达式中可以使用多个逗号运算符, 在 for 循环中, 初始化了 time、power_of_2 和 count。构建完循环条件之后, 程序本身就很简短了。

6.8 出口条件循环: do while

while 循环和 for 循环都是入口条件循环, 即在循环的每次迭代之前检查测试条件, 所以有可能根本不执行循环体中的内容。C 语言还有出口条件循环 (*exit-condition loop*), 即在循环的每次迭代之后检查测试条件, 这保证了至少执行循环体中的内容一次。这种循环被称为 do while 循环。程序清单 6.15 演示了一个示例。

程序清单 6.15 do_while.c 程序

```
/* do_while.c -- 出口条件循环 */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    do
```

```

{
    printf("To enter the triskaidekaphobia therapy club,\n");
    printf("please enter the secret code number: ");
    scanf("%d", &code_entered);
} while (code_entered != secret_code);
printf("Congratulations! You are cured!\n");

return 0;
}

```

程序清单 6.15 在用户输入 13 之前不断提示用户输入数字。下面是一个运行示例：

```

To enter the triskaidekaphobia therapy club,
please enter the secret code number: 12
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 14
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 13
Congratulations! You are cured!

```

使用 while 循环也能写出等价的程序，但是长一些，如程序清单 6.16 所示。

程序清单 6.16 entry.c 程序

```

/* entry.c -- 出口条件循环 */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    printf("To enter the triskaidekaphobia therapy club,\n");
    printf("please enter the secret code number: ");
    scanf("%d", &code_entered);
    while (code_entered != secret_code)
    {
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    }
    printf("Congratulations! You are cured!\n");

    return 0;
}

```

下面是 do while 循环的通用形式：

```

do
    statement
while ( expression );

```

statement 可以是一条简单语句或复合语句。注意，do while 循环以分号结尾，其结构见图 6.5。

do while 循环在执行完循环体后才执行测试条件，所以至少执行循环体一次；而 for 循环或 while 循环都是在执行循环体之前先执行测试条件。do while 循环适用于那些至少要迭代一次的循环。例如，下面是一个包含 do while 循环的密码程序伪代码：

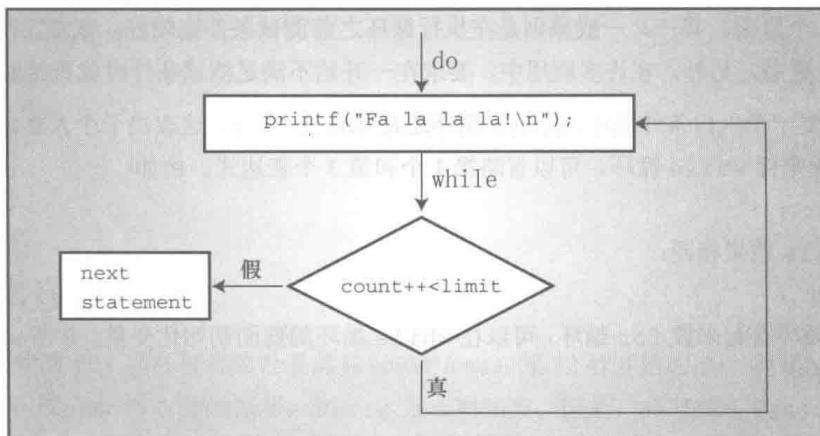


图 6.5 do while 循环的结构

```

do
{
    提示用户输入密码
    读取用户输入的密码
} while (用户输入的密码不等于密码);
避免使用这种形式的 do while 结构:
do
{
    询问用户是否继续
    其他行为
} while (回答是 yes);
  
```

这样的结构导致用户在回答“no”之后，仍然执行“其他行为”部分，因为测试条件执行晚了。

小结: do while 语句

关键字: do while

一般注解:

do while 语句创建一个循环，在 expression 为假或 0 之前重复执行循环体中的内容。do while 语句是一种出口条件循环，即在执行完循环体后才根据测试条件决定是否再次执行循环。因此，该循环至少必须执行一次。statement 部分可是一条简单语句或复合语句。

形式:

```

do
    statement
while ( expression );
  
```

在 test 为假或 0 之前，重复执行 statement 部分。

示例:

```

do
    scanf("%d", &number);
while (number != 20);
  
```

6.9 如何选择循环

如何选择使用哪一种循环？首先，确定是需要入口条件循环还是出口条件循环。通常，入口条件循环

用得比较多，有几个原因。其一，一般原则是在执行循环之前测试条件比较好。其二，测试放在循环的开头，程序的可读性更高。另外，在许多应用中，要求在一开始不满足测试条件时就直接跳过整个循环。

那么，假设需要一个入口条件循环，用 `for` 循环还是 `while` 循环？这取决于个人喜好，因为二者皆可。要让 `for` 循环看起来像 `while` 循环，可以省略第 1 个和第 3 个表达式。例如：

```
for ( ; test ; )
```

与下面的 `while` 效果相同：

```
while ( test )
```

要让 `while` 循环看起来像 `for` 循环，可以在 `while` 循环的前面初始化变量，并在 `while` 循环体中包含更新语句。例如：

```
初始化;
```

```
while ( 测试 )
```

```
{
```

```
    其他语句
```

```
    更新语句
```

```
}
```

与下面的 `for` 循环效果相同：

```
for ( 初始化 ; 测试 ; 更新 )
```

```
    其他语句
```

一般而言，当循环涉及初始化和更新变量时，用 `for` 循环比较合适，而在其他情况下用 `while` 循环更好。对于下面这种条件，用 `while` 循环就很合适：

```
while (scanf("%ld", &num) == 1)
```

对于涉及索引计数的循环，用 `for` 循环更适合。例如：

```
for (count = 1; count <= 100; count++)
```

6.10 嵌套循环

嵌套循环（*nested loop*）指在一个循环内包含另一个循环。嵌套循环常用于按行和列显示数据，也就是说，一个循环处理一行中的所有列，另一个循环处理所有的行。程序清单 6.17 演示了一个简单的示例。

程序清单 6.17 rows1.c 程序

```
/* rows1.c -- 使用嵌套循环 */
#include <stdio.h>
#define ROWS 6
#define CHARS 10
int main(void)
{
    int row;
    char ch;

    for (row = 0; row < ROWS; row++) /* 第 10 行 */
    {
        for (ch = 'A'; ch < ('A' + CHARS); ch++) /* 第 12 行 */
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

运行该程序后，输出如下：

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

6.10.1 程序分析

第 10 行开始的 `for` 循环被称为外层循环(*outer loop*)，第 12 行开始的 `for` 循环被称为内层循环(*inner loop*)。外层循环从 `row` 为 0 开始循环，到 `row` 为 6 时结束。因此，外层循环要执行 6 次，`row` 的值从 0 变为 5。每次迭代要执行的第 1 条语句是内层的 `for` 循环，该循环要执行 10 次，在同一行打印字符 A~J；第 2 条语句是外层循环的 `printf("\n");`，该语句的效果是另起一行，这样在下一次运行内层循环时，将在下一行打印的字符。

注意，嵌套循环中的内层循环在每次外层循环迭代时都执行完所有的循环。在程序清单 6.17 中，内层循环一行打印 10 个字符，外层循环创建 6 行。

6.10.2 嵌套变式

上一个实例中，内层循环和外层循环所做的事情相同。可以通过外层循环控制内层循环，在每次外层循环迭代时内层循环完成不同的任务。把程序清单 6.17 稍微修改后，如程序清单 6.18 所示。内层循环开始打印的字符取决于外层循环的迭代次数。该程序的第 1 行使用了新的注释风格，而且用 `const` 关键字代替 `#define`，有助于读者熟悉这两种方法。

程序清单 6.18 rows2.c 程序

```
// rows2.c -- 依赖外部循环的嵌套循环
#include <stdio.h>
int main(void)
{
    const int ROWS = 6;
    const int CHARS = 6;
    int row;
    char ch;

    for (row = 0; row < ROWS; row++)
    {
        for (ch = ('A' + row); ch < ('A' + CHARS); ch++)
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

该程序的输出如下：

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

因为每次迭代都要把 `row` 的值与 '`A`' 相加，所以 `ch` 在每一行都被初始化为不同的字符。然而，测试条件并没有改变，所以每行依然以 `F` 结尾，这使得每一行打印的字符都比上一行少一个。

6.11 数组简介

在许多程序中，数组很重要。数组可以作为一种储存多个相关项的便利方式。我们在第 10 章中将详细介绍数组，但是由于循环经常用到数组，所以在这里先简要地介绍一下。

数组 (*array*) 是按顺序储存的一系列类型相同的值，如 10 个 `char` 类型的字符或 15 个 `int` 类型的值。整个数组有一个数组名，通过整数下标访问数组中单独的项或元素 (*element*)。例如，以下声明：

```
float debts[20];
```

声明 `debts` 是一个内含 20 个元素的数组，每个元素都可以储存 `float` 类型的值。数组的第一个元素是 `debts[0]`，第二个元素是 `debts[1]`，以此类推，直到 `debts[19]`。注意，数组元素的编号从 0 开始，不是从 1 开始。可以给每个元素赋 `float` 类型的值。例如，可以这样写：

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

实际上，使用数组元素和使用同类型的变量一样。例如，可以这样把值读入指定的元素中：

```
scanf("%f", &debts[4]); // 把一个值读入数组的第 5 个元素
```

这里要注意一个潜在的陷阱：考虑到影响执行的速度，C 编译器不会检查数组的下标是否正确。下面的代码，都不正确：

```
debts[20] = 88.32; // 该数组元素不存在
debts[33] = 828.12; // 该数组元素不存在
```

编译器不会查找这样的错误。当运行程序时，这会导致数据被放置在已被其他数据占用的地方，可能会破坏程序的结果甚至导致程序异常中断。

数组的类型可以是任意数据类型。

```
int nannies[22]; /* 可储存 22 个 int 类型整数的数组 */
char actors[26]; /* 可储存 26 个字符的数组 */
long big[500]; /* 可储存 500 个 long 类型整数的数组 */
```

我们在第 4 章中讨论过字符串，可以把字符串储存在 `char` 类型的数组中（一般而言，`char` 类型数组的所有元素都储存 `char` 类型的值）。如果 `char` 类型的数组末尾包含一个表示字符串末尾的空字符 `\0`，则该数组中的内容就构成了一个字符串（见图 6.6）。

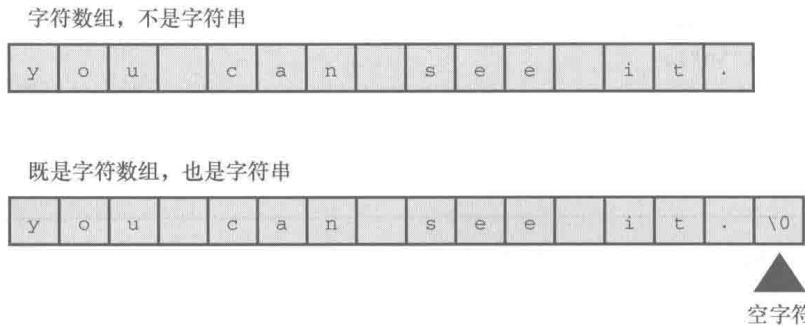


图 6.6 字符数组和字符串

用于识别数组元素的数字被称为下标 (*subscript*)、索引 (*indice*) 或偏移量 (*offset*)。下标必须是整数，

而且要从 0 开始计数。数组的元素被依次储存在内存中相邻的位置，如图 6.7 所示。

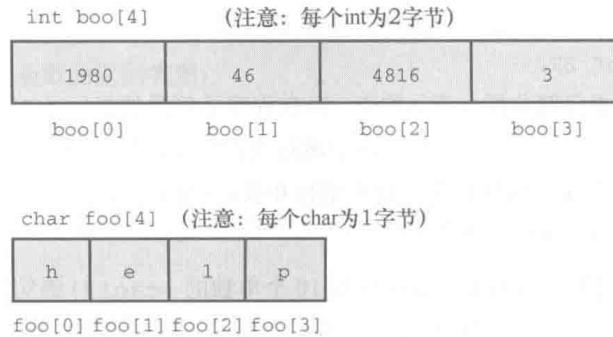


图 6.7 内存中的 `char` 和 `int` 类型的数组

6.11.1 在 `for` 循环中使用数组

程序中有许多地方要用到数组，程序清单 6.19 是一个较为简单的例子。该程序读取 10 个高尔夫分数，稍后进行处理。使用数组，就不用创建 10 个不同的变量来储存 10 个高尔夫分数。而且，还可以用 `for` 循环来读取数据。程序打印总分、平均分、差点 (*handicap*，它是平均分与标准分的差值)。

程序清单 6.19 `scores_in.c` 程序

```
// scores_in.c -- 使用循环处理数组
#include <stdio.h>
#define SIZE 10
#define PAR 72
int main(void)
{
    int index, score[SIZE];
    int sum = 0;
    float average;

    printf("Enter %d golf scores:\n", SIZE);
    for (index = 0; index < SIZE; index++)
        scanf("%d", &score[index]); // 读取 10 个分数
    printf("The scores read in are as follows:\n");
    for (index = 0; index < SIZE; index++)
        printf("%5d", score[index]); // 验证输入
    printf("\n");
    for (index = 0; index < SIZE; index++)
        sum += score[index]; // 求总分数
    average = (float) sum / SIZE; // 求平均分
    printf("Sum of scores = %d, average = %.2f\n", sum, average);
    printf("That's a handicap of %.0f.\n", average - PAR);

    return 0;
}
```

先看看程序清单 6.19 是否能正常工作，接下来再做一些解释。下面是程序的输出：

```
Enter 10 golf scores:
99 95 109 105 100
```

```

96 98 93 99 97 98
The scores read in are as follows:
99 95 109 105 100 96 98 93 99 97
Sum of scores = 991, average = 99.10
That's a handicap of 27.

```

程序运行没问题，我们来仔细分析一下。首先，注意程序示例虽然打印了 11 个数字，但是只读入了 10 个数字，因为循环只读了 10 个值。由于 `scanf()` 会跳过空白字符，所以可以在一行输入 10 个数字，也可以每行只输入一个数字，或者像本例这样混合使用空格和换行符隔开每个数字（因为输入是缓冲的，只有当用户键入 `Enter` 键后数字才会被发送给程序）。

然后，程序使用数组和循环处理数据，这比使用 10 个单独的 `scanf()` 语句和 10 个单独的 `printf()` 语句读取 10 个分数方便得多。`for` 循环提供了一个简单直接的方法来使用数组下标。注意，`int` 类型数组元素的用法与 `int` 类型变量的用法类似。要读取 `int` 类型变量 `fue`，应这样写：`scanf("%d", &fue)`。程序清单 6.19 中要读取 `int` 类型的元素 `score[index]`，所以这样写 `scanf("%d", &score[index])`。

该程序示例演示了一些较好的编程风格。第一，用 `#define` 指令创建的明示常量（`SIZE`）来指定数组的大小。这样就可以在定义数组和设置循环边界时使用该明示常量。如果以后要扩展程序处理 20 个分数，只需简单地把 `SIZE` 重新定义为 20 即可，不用逐一修改程序中使用了数组大小的每一处。

第二，下面的代码可以很方便地处理一个大小为 `SIZE` 的数组：

```
for (index = 0; index < SIZE; index++)
```

设置正确的数组边界很重要。第 1 个元素的下标是 0，因此循环开始时把 `index` 设置为 0。因为从 0 开始编号，所以数组中最后一个元素的下标是 `SIZE - 1`。也就是说，第 10 个元素是 `score[9]`。通过测试条件 `index < SIZE` 来控制循环中使用的最后一个 `index` 的值是 `SIZE - 1`。

第三，程序能重复显示刚读入的数据。这是很好的编程习惯，有助于确保程序处理的数据与期望相符。

最后，注意该程序使用了 3 个独立的 `for` 循环。这是否必要？是否可以将其合并成一个循环？当然可以，读者可以动手试试，合并后的程序显得更加紧凑。但是，调整时要注意遵循模块化（modularity）的原则。模块化隐含的思想是：应该把程序划分为一些独立的单元，每个单元执行一个任务。这样做提高了程序的可读性。也许更重要的是，模块化使程序的不同部分彼此独立，方便后续更新或修改程序。在掌握如何使用函数后，可以把每个执行任务的单元放进函数中，提高程序的模块化。

6.12 使用函数返回值的循环示例

本章最后一个程序示例要用一个函数计算数的整数次幂（`math.h` 库提供了一个更强大幂函数 `pow()`，可以使用浮点指数）。该示例有 3 个主要任务：设计算法、在函数中表示算法并返回计算结果、提供一个测试函数的便利方法。

首先分析算法。为简化函数，我们规定该函数只处理正整数的幂。这样，把 `n` 与 `n` 相乘 `p` 次便可计算 `n` 的 `p` 次幂。这里自然会用到循环。先把变量 `pow` 设置为 1，然后将其反复乘以 `n`：

```
for(i = 1; i <= p; i++)
    pow *= n;
```

回忆一下，`*=` 运算符把左侧的项乘以右侧的项，再把乘积赋给左侧的项。第 1 次循环后，`pow` 的值是 1 乘以 `n`，即 `n`；第 2 次循环后，`pow` 的值是上一次的值（`n`）乘以 `n`，即 `n` 的平方；以此类推。这种情况使用 `for` 循环很合适，因为在执行循环之前已预先知道了迭代的次数（已知 `p`）。

现在算法已确定，接下来要决定使用何种数据类型。指数 `p` 是整数，其类型应该是 `int`。为了扩大 `n` 及其幂的范围，`n` 和 `pow` 的类型都是 `double`。

接下来，考虑如何把以上内容用函数来实现。要使用两个参数（分别是 double 类型和 int 类型）才能把所需的信息传递给函数，并指定求哪个数的多少次幂。而且，函数要返回一个值。如何把函数的返回值返回给主调函数？编写一个有返回值的函数，要完成以下内容：

1. 定义函数时，确定函数的返回类型；
2. 使用关键字 return 表明待返回的值。

例如，可以这样写：

```
double power(double n, int p) // 返回一个 double 类型的值
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow; // 返回 pow 的值
}
```

要声明函数的返回类型，在函数名前写出类型即可，就像声明一个变量那样。关键字 return 表明该函数将把它后面的值返回给主调函数。根据上面的代码，函数返回一个变量的值。返回值也可以是表达式的值，如下所示：

```
return 2 * x + b;
```

函数将计算表达式的值，并返回该值。在主调函数中，可以把返回值赋给另一个变量、作为表达式中的值、作为另一个函数的参数（如，`printf("%f", power(6.28, 3))`），或者忽略它。

现在，我们在一个程序中使用这个函数。要测试一个函数很简单，只需给它提供几个值，看它是如何响应的。这种情况下可以创建一个输入循环，选择 while 循环很合适。可以使用 `scanf()` 函数一次读取两个值。如果成功读取两个值，`scanf()` 则返回 2，所以可以把 `scanf()` 的返回值与 2 作比较来控制循环。还要注意，必须先声明 `power()` 函数（即写出函数原型）才能在程序中使用它，就像先声明变量再使用一样。程序清单 6.20 演示了这个程序。

程序清单 6.20 power.c 程序

```
// power.c -- 计算数的整数幂
#include <stdio.h>
double power(double n, int p); // ANSI 函数原型
int main(void)
{
    double x, xpow;
    int exp;

    printf("Enter a number and the positive integer power");
    printf(" to which\nthe number will be raised. Enter q");
    printf(" to quit.\n");
    while (scanf("%lf%d", &x, &exp) == 2)
    {
        xpow = power(x, exp); // 函数调用
        printf("%.3g to the power %d is %.5g\n", x, exp, xpow);
        printf("Enter next pair of numbers or q to quit.\n");
    }
    printf("Hope you enjoyed this power trip -- bye!\n");
```

```

    return 0;
}

double power(double n, int p) // 函数定义
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow; // 返回 pow 的值
}

```

运行该程序后，输出示例如下：

```

Enter a number and the positive integer power to which
the number will be raised. Enter q to quit.
1.2 12
1.2 to the power 12 is 8.9161
Enter next pair of numbers or q to quit.
2
16
2 to the power 16 is 65536
Enter next pair of numbers or q to quit.
q
Hope you enjoyed this power trip -- bye!

```

6.12.1 程序分析

该程序示例中的 main() 是一个驱动程序 (driver)，即被设计用来测试函数的小程序。

该例的 while 循环是前面讨论过的一般形式。输入 1.2 12，scanf() 成功读取两值，并返回 2，循环继续。因为 scanf() 跳过空白，所以可以像输出示例那样，分多行输入。但是输入 q 会使 scanf() 的返回值为 0，因为 q 与 scanf() 中的转换说明 %1f 不匹配。scanf() 将返回 0，循环结束。类似地，输入 2.8 q 会使 scanf() 的返回值为 1，循环也会结束。

现在分析一下与函数相关的内容。power() 函数在程序中出现了 3 次。首次出现是：

```
double power(double n, int p); // ANSI 函数原型
```

这是 power() 函数的原型，它声明程序将使用一个名为 power() 的函数。开头的关键字 double 表明 power() 函数返回一个 double 类型的值。编译器要知道 power() 函数返回值的类型，才能知道有多少字节的数据，以及如何解释它们。这就是为什么必须声明函数的原因。圆括号中的 double n, int p 表示 power() 函数的两个参数。第 1 个参数应该是 double 类型的值，第 2 个参数应该是 int 类型的值。

第 2 次出现是：

```
xpow = power(x, exp); // 函数调用
```

程序调用 power()，把两个值传递给它。该函数计算 x 的 exp 次幂，并把计算结果返回给主调函数。在主调函数中，返回值将被赋给变量 xpow。

第 3 次出现是：

```
double power(double n, int p) // 函数定义
```

这里，power() 有两个形参，一个是 double 类型，一个是 int 类型，分别由变量 n 和变量 p 表示。

注意，函数定义的末尾没有分号，而函数原型的末尾有分号。在函数头后面花括号中的内容，就是 power() 完成任务的代码。

power() 函数用 for 循环计算 n 的 p 次幂，并把计算结果赋给 pow，然后返回 pow 的值，如下所示：

```
return pow; // 返回 pow 的值
```

6.12.2 使用带返回值的函数

声明函数、调用函数、定义函数、使用关键字 return，都是定义和使用带返回值函数的基本要素。

这里，读者可能有一些问题。例如，既然在使用函数返回值之前要声明函数，那么为什么在使用 scanf() 的返回值之前没有声明 scanf()？为什么在定义中说明了 power() 的返回类型为 double，还要单独声明这个函数？

我们先回答第 2 个问题。编译器在程序中首次遇到 power() 时，需要知道 power() 的返回类型。此时，编译器尚未执行到 power() 的定义，并不知道函数定义中的返回类型是 double。因此，必须通过前置声明（forward declaration）预先说明函数的返回类型。前置声明告诉编译器，power() 定义在别处，其返回类型为 double。如果把 power() 函数的定义置于 main() 的文件顶部，就可以省略前置声明，因为编译器在执行到 main() 之前已经知道 power() 的所有信息。但是，这不是 C 的标准风格。因为 main() 通常只提供整个程序的框架，最好把 main() 放在所有函数定义的前面。另外，通常把函数放在其他文件中，所以前置声明必不可少。

接下来，为什么不用声明 scanf() 函数就可以使用它？其实，你已经声明了。stdio.h 头文件中包含了 scanf()、printf() 和其他 I/O 函数的原型。scanf() 函数的原型表明，它返回的类型是 int。

6.13 关键概念

循环是一个强大的编程工具。在创建循环时，要特别注意以下 3 个方面：

- 注意循环的测试条件要能使循环结束；
- 确保循环测试中的值在首次使用之前已初始化；
- 确保循环在每次迭代都更新测试的值。

C 通过求值来处理测试条件，结果为 0 表示假，非 0 表示真。带关系运算符的表达式常用于循环测试，它们有些特殊。如果关系表达式为真，其值为 1；如果为假，其值为 0。这与新类型 _Bool 的值保持一致。

数组由相邻的内存位置组成，只储存相同类型的数据。记住，数组元素的编号从 0 开始，所有数组最后一个元素的下标一定比元素数目少 1。C 编译器不会检查数组下标值是否有效，自己要多留心。

使用函数涉及 3 个步骤：

- 通过函数原型声明函数；
- 在程序中通过函数调用使用函数；
- 定义函数。

函数原型是为了方便编译器查看程序中使用的函数是否正确，函数定义描述了函数如何工作。现代的编程习惯是把程序要素分为接口部分和实现部分，例如函数原型和函数定义。接口部分描述了如何使用一个特性，也就是函数原型所做的；实现部分描述了具体的行为，这正是函数定义所做的。

6.14 本章小结

本章的主题是程序控制。C 语言为实现结构化的程序提供了许多工具。`while` 语句和 `for` 语句提供了入口条件循环。`for` 语句特别适用于需要初始化和更新的循环。使用逗号运算符可以在 `for` 循环中初始化和更新多个变量。有些场合也需要使用出口条件循环，C 为此提供了 `do while` 语句。

典型的 `while` 循环设计的伪代码如下：

```
获得初值
while (值满足测试条件)
{
    处理该值
    获取下一个值
}
```

`for` 循环也可以完成相同的任务：

```
for (获得初值; 值满足测试条件; 获得下一个值)
    处理该值
```

这些循环都使用测试条件来判断是否继续执行下一次迭代。一般而言，如果对测试表达式求值为非 0，则继续执行循环；否则，结束循环。通常，测试条件都是关系表达式（由关系运算符和表达式构成）。表达式的关系为真，则表达式的值为 1；如果关系为假，则表达式的值为 0。C99 新增了 `_Bool` 类型，该类型的变量只能储存 1 或 0，分别表示真或假。

除了关系运算符，本章还介绍了其他的组合赋值运算符，如 `+=` 或 `*=`。这些运算符通过对其左侧运算对象执行算术运算来修改它的值。

接下来还简单地介绍了数组。声明数组时，方括号中的值指明了该数组的元素个数。数组的第一个元素编号为 0，第二个元素编号为 1，以此类推。例如，以下声明：

```
double hippos[20];
```

创建了一个有 20 个元素的数组 `hippos`，其元素从 `hippos[0] ~ hippos[19]`。利用循环可以很方便地操控数组的下标。

最后，本章演示了如何编写和使用带返回值的函数。

6.15 复习题

复习题的参考答案在附录 A 中。

- 写出执行完下列各行后 `quack` 的值是多少。后 5 行中使用的是第 1 行 `quack` 的值。

```
int quack = 2;
quack += 5;
quack *= 10;
quack -= 6;
quack /= 8;
quack %= 3;
```

- 假设 `value` 是 `int` 类型，下面循环的输出是什么？

```
for (value = 36; value > 0; value /= 2)
    printf("%3d", value);
```

如果 `value` 是 `double` 类型，会出现什么问题？

3. 用代码表示以下测试条件:

- a. x大于5
- b. scanf()读取一个名为x的double类型值且失败
- c. x 的值等于5

4. 用代码表示以下测试条件:

- a. scanf()成功读入一个整数
- b. x 不等于5
- c. x 大于或等于20

5. 下面的程序有点问题, 请找出问题所在。

```
#include <stdio.h>
int main(void)
{
    int i, j, list(10); /* 第 3 行 */
    /* 第 4 行 */

    for (i = 1, i <= 10, i++) /* 第 6 行 */
    {
        list[i] = 2*i + 3; /* 第 8 行 */
        for (j = 1, j >= i, j++) /* 第 9 行 */
            printf(" %d", list[j]); /* 第 10 行 */
        printf("\n"); /* 第 11 行 */
    } /* 第 12 行 */
```

6. 编写一个程序打印下面的图案, 要求使用嵌套循环:

```
$$$$$$$
$$$$$$
$$$$$$
$$$$$$
```

7. 下面的程序各打印什么内容?

a.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    while (++i < 4)
        printf("Hi! ");
    do
        printf("Bye! ");
    while (i++ < 8);
    return 0;
}
```

b.

```
#include <stdio.h>
int main(void)
{
    int i;
    char ch;
```

```

    for (i = 0, ch = 'A'; i < 4; i++, ch += 2 * i)
        printf("%c", ch);
    return 0;
}

```

8. 假设用户输入的是 Go west, young man!，下面各程序的输出是什么？（在 ASCII 码中，!紧跟在空格字符后面）

a.

```

#include <stdio.h>
int main(void)
{
    char ch;

    scanf("%c", &ch);
    while (ch != 'g')
    {
        printf("%c", ch);
        scanf("%c", &ch);
    }
    return 0;
}

```

b.

```

#include <stdio.h>

int main(void)
{
    char ch;

    scanf("%c", &ch);
    while (ch != 'g')
    {
        printf("%c", ++ch);
        scanf("%c", &ch);
    }
    return 0;
}

```

c.

```

#include <stdio.h>

int main(void)
{
    char ch;

    do {
        scanf("%c", &ch);
        printf("%c", ch);
    } while (ch != 'g');
    return 0;
}

```

d.

```

#include <stdio.h>
int main(void)
{
    char ch;

```

```

scanf("%c", &ch);
for (ch = '$'; ch != 'g'; scanf("%c", &ch))
    printf("%c", ch);
return 0;
}

```

9. 下面的程序打印什么内容?

```

#include <stdio.h>
int main(void)
{
    int n, m;

    n = 30;
    while (++n <= 33)
        printf("%d|", n);

    n = 30;
    do
        printf("%d|", n);
    while (++n <= 33);

    printf("\n***\n");

    for (n = 1; n*n < 200; n += 4)
        printf("%d\n", n);

    printf("\n***\n");

    for (n = 2, m = 6; n < m; n *= 2, m += 2)
        printf("%d %d\n", n, m);

    printf("\n***\n");

    for (n = 5; n > 0; n--)
    {
        for (m = 0; m <= n; m++)
            printf("=");
        printf("\n");
    }
    return 0;
}

```

10. 考虑下面的声明:

```
double mint[10];
```

- 数组名是什么?
- 该数组有多少个元素?
- 每个元素可以储存什么类型的值?
- 下面的哪一个 `scanf()` 的用法正确?
 - `scanf("%lf", mint[2])`
 - `scanf("%lf", &mint[2])`
 - `scanf("%lf", &mint)`

11. Noah 先生喜欢以 2 计数, 所以编写了下面的程序, 创建了一个储存 2、4、6、8 等数字的数组。

这个程序是否有错误之处？如果有，请指出。

```
#include <stdio.h>
#define SIZE 8
int main(void)
{
    int by_twos[SIZE];
    int index;

    for (index = 1; index <= SIZE; index++)
        by_twos[index] = 2 * index;
    for (index = 1; index <= SIZE; index++)
        printf("%d ", by_twos);
    printf("\n");
    return 0;
}
```

12. 假设要编写一个返回 long 类型值的函数，函数定义中应包含什么？
13. 定义一个函数，接受一个 int 类型的参数，并以 long 类型返回参数的平方值。
14. 下面的程序打印什么内容？

```
#include <stdio.h>
int main(void)
{
    int k;
    for (k = 1, printf("Hi!\n", k); printf("k = %d\n", k),
         k*k < 26; k += 2, printf("Now k is %d\n", k))
        printf("k is %d in the loop\n", k);
    return 0;
}
```

6.16 编程练习

1. 编写一个程序，创建一个包含 26 个元素的数组，并在其中储存 26 个小写字母。然后打印数组的所有内容。
2. 使用嵌套循环，按下面的格式打印字符：

```
$  
$$  
$$$  
$$$$  
$$$$$
```

3. 使用嵌套循环，按下面的格式打印字母：

```
F  
FE  
FED  
FEDC  
FEDCB  
FEDCBA
```

注意：如果你的系统不使用 ASCII 或其他以数字顺序编码的代码，可以把字符数组初始化为字母表中的字母：

```
char lets[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

然后用数组下标选择单独的字母，例如 lets[0] 是 'A'，等等。

4. 使用嵌套循环，按下面的格式打印字母：

```
A
BC
DEF
GHIJ
KLMNO
PQRSTU
```

如果你的系统不使用以数字顺序编码的代码，请参照练习 3 的方案解决。

5. 编写一个程序，提示用户输入大写字母。使用嵌套循环以下面金字塔型的格式打印字母：

```
A
ABA
ABCBA
ABCDcba
ABCDEDCBA
```

打印这样的图形，要根据用户输入的字母来决定。例如，上面的图形是在用户输入 E 后的打印结果。

提示：用外层循环处理行，每行使用 3 个内层循环，分别处理空格、以升序打印字母、以降序打印字母。如果系统不使用 ASCII 或其他以数字顺序编码的代码，请参照练习 3 的解决方案。

6. 编写一个程序打印一个表格，每一行打印一个整数、该数的平方、该数的立方。要求用户输入表格的上下限。使用一个 `for` 循环。
7. 编写一个程序把一个单词读入一个字符数组中，然后倒序打印这个单词。提示：`strlen()` 函数（第 4 章介绍过）可用于计算数组最后一个字符的下标。
8. 编写一个程序，要求用户输入两个浮点数，并打印两数之差除以两数乘积的结果。在用户输入非数字之前，程序应循环处理用户输入的每对值。
9. 修改练习 8，使用一个函数返回计算的结果。
10. 编写一个程序，要求用户输入一个上限整数和一个下限整数，计算从上限到下限范围内所有整数的平方和，并显示计算结果。然后程序继续提示用户输入上限和下限整数，并显示结果，直到用户输入的上限整数小于下限整数为止。程序的运行示例如下：

```
Enter lower and upper integer limits: 5 9
The sums of the squares from 5 to 9 is 255
Enter next set of limits: 3 25
The sums of the squares from 3 to 25 is 5520
Enter next set of limits: 5 5
Done
```

11. 编写一个程序，在数组中读入 8 个整数，然后按倒序打印这 8 个整数。

12. 考虑下面两个无限序列：

```
1.0 + 1.0/2.0 + 1.0/3.0 + 1.0/4.0 + ...
1.0 - 1.0/2.0 + 1.0/3.0 - 1.0/4.0 + ...
```

编写一个程序计算这两个无限序列的总和，直到到达某次数。提示：奇数个 -1 相乘得 -1，偶数个 -1 相乘得 1。让用户交互地输入指定的次数，当用户输入 0 或负值时结束输入。查看运行 100 项、1000 项、10000 项后的总和，是否发现每个序列都收敛于某值？

13. 编写一个程序，创建一个包含 8 个元素的 `int` 类型数组，分别把数组元素设置为 2 的前 8 次幂。使用 `for` 循环设置数组元素的值，使用 `do while` 循环显示数组元素的值。
14. 编写一个程序，创建两个包含 8 个元素的 `double` 类型数组，使用循环提示用户为第一个数组输入 8 个值。第二个数组元素的值设置为第一个数组对应元素的累积之和。例如，第二个数组的第 4

个元素的值是第一个数组前 4 个元素之和，第二个数组的第 5 个元素的值是第一个数组前 5 个元素之和（用嵌套循环可以完成，但是利用第二个数组的第 5 个元素是第二个数组的第 4 个元素与第一个数组的第 5 个元素之和，只用一个循环就能完成任务，不需要使用嵌套循环）。最后，使用循环显示两个数组的内容，第一个数组显示成一行，第二个数组显示在第一个数组的下一行，而且每个元素都与第一个数组各元素相对应。

15. 编写一个程序，读取一行输入，然后把输入的内容倒序打印出来。可以把输入储存在 char 类型的数组中，假设每行字符不超过 255。回忆一下，根据%c 转换说明，scanf() 函数一次只能从输入中读取一个字符，而且在用户按下 Enter 键时 scanf() 函数会生成一个换行字符 (\n)。
16. Daphne 以 10% 的单利息投资了 100 美元（也就是说，每年投资获利相当于原始投资的 10%）。Deirdre 以 5% 的复合利息投资了 100 美元（也就是说，利息是当前余额的 5%，包含之前的利息）。编写一个程序，计算需要多少年 Deirdre 的投资额才会超过 Daphne，并显示那时两人的投资额。
17. Chuckie Lucky 赢得了 100 万美元（税后），他把奖金存入年利率 8% 的账户。在每年的最后一天，Chuckie 取出 10 万美元。编写一个程序，计算多少年后 Chuckie 会取完账户的钱？
18. Rabnud 博士加入了一个社交圈。起初他有 5 个朋友。他注意到他的朋友数量以下面的方式增长。第 1 周少了 1 个朋友，剩下的朋友数量翻倍；第 2 周少了 2 个朋友，剩下的朋友数量翻倍。一般而言，第 N 周少了 N 个朋友，剩下的朋友数量翻倍。编写一个程序，计算并显示 Rabnud 博士每周的朋友数量。该程序一直运行，直到超过邓巴数 (*Dunbar's number*)。邓巴数是粗略估算一个人在社交圈中有稳定关系的成员的最大值，该值大约是 150。

C 控制语句：分支和跳转

本章介绍以下内容：

- 关键字：if、else、switch、continue、break、case、default、goto
- 运算符：&&、||、?:
- 函数：getchar()、putchar()、ctype.h 系列
- 如何使用 if 和 if else 语句，如何嵌套它们
- 在更复杂的测试表达式中用逻辑运算符组合关系表达式
- C 的条件运算符
- switch 语句
- break、continue 和 goto 语句
- 使用 C 的字符 I/O 函数：getchar() 和 putchar()
- ctype.h 头文件提供的字符分析函数系列

随着越来越熟悉 C，可以尝试用 C 程序解决一些更复杂的问题。这时候，需要一些方法来控制和组织程序，为此 C 提供了一些工具。前面已经学过如何在程序中用循环重复执行任务。本章将介绍分支结构（如，if 和 switch），让程序根据测试条件执行相应的行为。另外，还将介绍 C 语言的逻辑运算符，使用逻辑运算符能在 while 或 if 的条件下测试更多关系。此外，本章还将介绍跳转语句，它将程序流转换到程序的其他部分。学完本章后，读者就可以设计按自己期望方式运行的程序。

7.1 if 语句

我们从一个有 if 语句的简单示例开始学习，请看程序清单 7.1。该程序读取一列数据，每个数据都表示每日的最低温度（℃），然后打印统计的总天数和最低温度在 0℃ 以下的天数占总天数的百分比。程序中的循环通过 scanf() 读入温度值。while 循环每迭代一次，就递增计数器增加天数，其中的 if 语句负责判断 0℃ 以下的温度并单独统计相应的天数。

程序清单 7.1 colddays.c 程序

```
// colddays.c -- 找出 0°C 以下的天数占总天数的百分比
#include <stdio.h>
int main(void)
{
    const int FREEZING = 0;
    float temperature;
    int cold_days = 0;
    int all_days = 0;

    printf("Enter the list of daily low temperatures.\n");
    while (scanf("%f", &temperature) != EOF) {
        if (temperature <= FREEZING)
            cold_days++;
        all_days++;
    }
    printf("The percentage of cold days is %.2f\n",
           (cold_days * 100.0 / all_days));
}
```

```

printf("Use Celsius, and enter q to quit.\n");
while (scanf("%f", &temperature) == 1)
{
    all_days++;
    if (temperature < FREEZING)
        cold_days++;
}
if (all_days != 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
if (all_days == 0)
    printf("No data entered!\n");

return 0;
}

```

下面是该程序的输出示例：

```

Enter the list of daily low temperatures.
Use Celsius, and enter q to quit.
12 5 -2.5 0 6 8 -3 -10 5 10 q
10 days total: 30.0% were below freezing.

```

while 循环的测试条件利用 scanf() 的返回值来结束循环，因为 scanf() 在读到非数字字符时会返回 0。temperature 的类型是 float 而不是 int，这样程序既可以接受-2.5 这样的值，也可以接受 8 这样的值。

while 循环中的新语句如下：

```

if (temperature < FREEZING)
    cold_days++;

```

if 语句指示计算机，如果刚读取的值 (temperature) 小于 0，就把 cold_days 递增 1；如果 temperature 不小于 0，就跳过 cold_days++; 语句，while 循环继续读取下一个温度值。

接着，该程序又使用了两次 if 语句控制程序的输出。如果有数据，就打印结果；如果没有数据，就打印一条消息（稍后将介绍一种更好的方法来处理这种情况）。

为避免整数除法，该程序示例把计算后的百分比强制转换为 float 类型。其实，也不必使用强制类型转换，因为在表达式 $100.0 * \text{cold_days} / \text{all_days}$ 中，将首先对表达式 $100.0 * \text{cold_days}$ 求值，由于 C 的自动转换类型规则，乘积会被强制转换成浮点数。但是，使用强制类型转换可以明确表达转换类型的意图，保护程序免受不同版本编译器的影响。if 语句被称为分支语句 (*branching statement*) 或选择语句 (*selection statement*)，因为它相当于一个交叉点，程序要在两条分支中选择一条执行。if 语句的通用形式如下：

```

if ( expression )
    statement

```

如果对 expression 求值为真 (非 0)，则执行 statement；否则，跳过 statement。与 while 循环一样，statement 可以是一条简单语句或复合语句。if 语句的结构和 while 语句很相似，它们的主要区别是：如果满足条件可执行的话，if 语句只能测试和执行一次，而 while 语句可以测试和执行多次。

通常，expression 是关系表达式，即比较两个量的大小 (如，表达式 $x > y$ 或 $c == 6$)。如果 expression 为真 (即 x 大于 y ，或 $c == 6$)，则执行 statement。否则，忽略 statement。概括地说，可以使用任意表达式，表达式的值为 0 则为假。

statement 部分可以是一条简单语句，如本例所示，或者是一条用花括号括起来的复合语句 (或块)：

```

if (score > big)
    printf("Jackpot!\n"); // 简单语句

if (joe > ron)
{
    // 复合语句
    joecash++;
    printf("You lose, Ron.\n");
}

```

注意，即使 if 语句由复合语句构成，整个 if 语句仍被视为一条语句。

7.2 if else 语句

简单形式的 if 语句可以让程序选择执行一条语句，或者跳过这条语句。C 还提供了 if else 形式，可以在两条语句之间作选择。我们用 if else 形式修正程序清单 7.1 中的程序段。

```

if (all_days != 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
if (all_days == 0)
    printf("No data entered!\n");

```

如果程序发现 all_days 不等于 0，那么它应该知道另一种情况一定是 all_days 等于 0。用 if else 形式只需测试一次。重写上面的程序段如下：

```

if (all_days != 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
else
    printf("No data entered!\n");

```

如果 if 语句的测试表达式为真，就打印温度数据；如果为假，就打印警告消息。

注意，if else 语句的通用形式是：

```

if ( expression )
    statement1
else
    statement2

```

如果 expression 为真（非 0），则执行 statement1；如果 expression 为假或 0，则执行 else 后面的 statement2。statement1 和 statement2 可以是一条简单语句或复合语句。C 并不要求一定要缩进，但这是标准风格。缩进让根据测试条件的求值结果来判断执行哪部分语句一目了然。

如果要在 if 和 else 之间执行多条语句，必须用花括号把这些语句括起来成为一个块。下面的代码结构违反了 C 语法，因为在 if 和 else 之间只允许有一条语句（简单语句或复合语句）：

```

if (x > 0)
    printf("Incrementing x:\n");
    x++;
else // 将产生一个错误
    printf("x <= 0\n");

```

编译器把 printf() 语句视为 if 语句的一部分，而把 x++; 看作一条单独的语句，它不是 if 语句的一部分。然后，编译器发现 else 并没有所属的 if，这是错误的。上面的代码应该这样写：

```

if (x > 0)
{
    printf("Incrementing x:\n");
}

```

```

    x++;
}
else
    printf("x <= 0 \n");

```

`if` 语句用于选择是否执行一个行为，而 `else if` 语句用于在两个行为之间选择。图 7.1 比较了这两种语句。

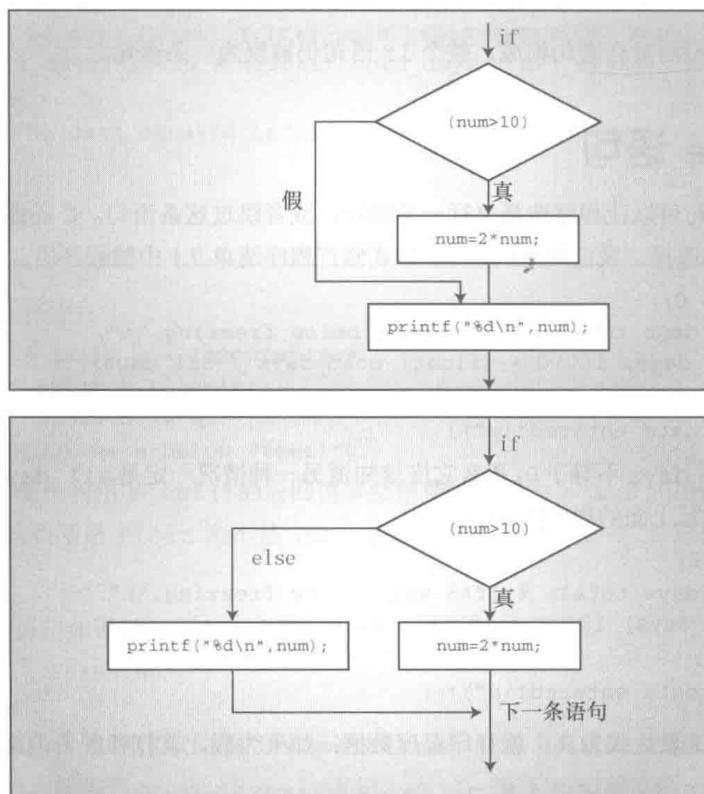


图 7.1 `if` 语句和 `if...else` 语句

7.2.1 另一个示例：介绍 `getchar()` 和 `putchar()`

到目前为止，学过的大多数组程序示例都要求输入数值。接下来，我们看看输入字符的示例。相信读者已经熟悉了如何用 `scanf()` 和 `printf()` 根据`%c` 转换说明读写字符，我们马上要讲解的示例中要用到一对字符输入/输出函数：`getchar()` 和 `putchar()`。

`getchar()` 函数不带任何参数，它从输入队列中返回下一个字符。例如，下面的语句读取下一个字符输入，并把该字符的值赋给变量 `ch`：

```
ch = getchar();
```

该语句与下面的语句效果相同：

```
scanf("%c", &ch);
```

`putchar()` 函数打印它的参数。例如，下面的语句把之前赋给 `ch` 的值作为字符打印出来：

```
putchar(ch);
```

该语句与下面的语句效果相同：

```
printf("%c", ch);
```

由于这些函数只处理字符，所以它们比更通用的 `scanf()` 和 `printf()` 函数更快、更简洁。而且，注意 `getchar()` 和 `putchar()` 不需要转换说明，因为它们只处理字符。这两个函数通常定义在 `stdio.h`

头文件中（而且，它们通常是预处理宏，而不是真正的函数，第 16 章会讨论类似函数的宏）。

接下来，我们编写一个程序来说明这两个函数是如何工作的。该程序把一行输入重新打印出来，但是每个非空格都被替换成原字符在 ASCII 序列中的下一个字符，空格不变。这一过程可描述为“如果字符是空白，原样打印；否则，打印原字符在 ASCII 序列中的下一个字符”。

C 代码看上去和上面的描述很相似，请看程序清单 7.2。

程序清单 7.2 cypher1.c 程序

```
// cypher1.c -- 更改输入，空格不变
#include <stdio.h>
#define SPACE ' '
int main(void)
{
    char ch;

    ch = getchar(); // 读取一个字符
    while (ch != '\n') // 当一行未结束时
    {
        if (ch == SPACE) // 留下空格
            putchar(ch); // 该字符不变
        else
            putchar(ch + 1); // 改变其他字符
        ch = getchar(); // 获得下一个字符
    }
    putchar(ch); // 打印换行符

    return 0;
}
```

（如果编译器警告因转换可能导致数据丢失，不用担心。第 8 章在讲到 EOF 时再解释。）

下面是该程序的输入示例：

```
CALL ME HAL.
DBMM NF IBM/
```

把程序清单 7.1 中的循环和该例中的循环作比较。前者使用 `scanf()` 返回的状态值判断是否结束循环，而后者使用输入项的值来判断是否结束循环。这使得两程序所用的循环结构略有不同：程序清单 7.1 中在循环前面有一条“读取语句”，程序清单 7.2 中在每次迭代的末尾有一条“读取语句”。不过，C 的语法比较灵活，读者也可以模仿程序清单 7.1，把读取和测试合并成一个表达式。也就是说，可以把这种形式的循环：

```
ch = getchar(); /* 读取一个字符 */
while (ch != '\n') /* 当一行未结束时 */
{
    ...
    /* 处理字符 */
    ch = getchar(); /* 获得下一个字符 */
}
```

替换成下面形式的循环：

```
while ((ch = getchar()) != '\n')
{
    ...
    /* 处理字符 */
}
```

关键的一行代码是：

```
while ((ch = getchar()) != '\n')
```

这体现了C特有的编程风格——把两个行为合并成一个表达式。C对代码的格式要求宽松，这样写让其中的每个行为更加清晰：

```
while (
    (ch = getchar())           // 给 ch 赋一个值
    != '\n')      // 把 ch 和\n 作比较
```

以上执行的行为是赋值给ch和把ch的值与换行符作比较。表达式ch = getchar()两侧的圆括号使之成为!=运算符的左侧运算对象。要对该表达式求值，必须先调用getchar()函数，然后把该函数的返回值赋给ch。因为赋值表达式的值是赋值运算符左侧运算对象的值，所以ch = getchar()的值就是ch的新值，因此，读取ch的值后，测试条件相当于ch != '\n'（即，ch不是换行符）。

这种独特的写法在C编程中很常见，应该多熟悉它。还要记住合理使用圆括号组合子表达式。上面例子中的圆括号都必不可少。假设省略ch = getchar()两侧的圆括号：

```
while (ch = getchar() != '\n')
```

!=运算符的优先级比=高，所以先对表达式getchar() != '\n'求值。由于这是关系表达式，所以其值不是1就是0(真或假)。然后，把该值赋给ch。省略圆括号意味着赋给ch的值是0或1，而不是getchar()的返回值。这不是我们的初衷。

下面的语句：

```
putchar(ch + 1); /* 改变其他字符 */
```

再次演示了字符实际上是作为整数储存的。为方便计算，表达式ch + 1中的ch被转换成int类型，然后int类型的计算结果被传递给接受一个int类型参数的putchar()，该函数只根据最后一个字节确定显示哪个字符。

7.2.2 ctype.h系列的字符函数

注意到程序清单7.2的输出中，最后输入的点号(.)被转换成斜杠(/)，这是因为斜杠字符对应的ASCII码比点号的ASCII码多1。如果程序只转换字母，保留所有的非字母字符(不只是空格)会更好。本章稍后讨论的逻辑运算符可用来测试字符是否不是空格、不是逗号等，但是列出所有的可能性太繁琐。C有一系列专门处理字符的函数，ctype.h头文件包含了这些函数的原型。这些函数接受一个字符作为参数，如果该字符属于某特殊的类别，就返回一个非零值(真)；否则，返回0(假)。例如，如果isalpha()函数的参数是一个字母，则返回一个非零值。程序清单7.3在程序清单7.2的基础上使用了这个函数，还使用了刚才精简后的循环。

程序清单7.3 cypher2.c程序

```
// cypher2.c -- 替换输入的字母，非字母字符保持不变
#include <stdio.h>
#include <ctype.h>           // 包含isalpha()的函数原型
int main(void)
{
    char ch;

    while ((ch = getchar()) != '\n')
    {
        if (isalpha(ch))      // 如果是一个字符,
            putchar(ch + 1);  // 显示该字符的下一个字符
    }
}
```

```

    else          // 否则,
        putchar(ch); // 原样显示
    }
    putchar(ch); // 显示换行符

    return 0;
}

```

下面是该程序的一个输出示例，注意大小写字母都被替换了，除了空格和标点符号：

```

Look! It's a programmer!
Mpp! Ju't b qsphsbnnfs!

```

表 7.1 和表 7.2 列出了 ctype.h 头文件中的一些函数。有些函数涉及本地化，指的是为适应特定区域的使用习惯修改或扩展 C 基本用法的工具（例如，许多国家在书写小数点时，用逗号代替点号，于是特殊的本地化可以指定 C 编译器使用逗号以相同的方式输出浮点数，这样 123.45 可以显示为 123,45）。注意，字符映射函数不会修改原始的参数，这些函数只会返回已修改的值。也就是说，下面的语句不改变 ch 的值：

```
tolower(ch); // 不影响 ch 的值
```

这样做才会改变 ch 的值：

```
ch = tolower(ch); // 把 ch 转换成小写字母
```

表 7.1 ctype.h 头文件中的字符测试函数

函数名	如果是下列参数时，返回值为真
isalnum()	字母数字（字母或数字）
isalpha()	字母
isblank()	标准的空白字符（空格、水平制表符或换行符）或任何其他本地化指定为白色的字符
iscntrl()	控制字符，如 Ctrl+B
isdigit()	数字
isgraph()	除空格之外的任意可打印字符
islower()	小写字母
isprint()	可打印字符
ispunct()	标点符号（除空格或字母数字字符以外的任何可打印字符）
isspace()	空白字符（空格、换行符、换页符、回车符、垂直制表符、水平制表符或其他本地化定义的字符）
isupper()	大写字母
isxdigit()	十六进制数字符

表 7.2 ctype.h 头文件中的字符映射函数

函数名	行为
tolower()	如果参数是大写字符，该函数返回小写字符；否则，返回原始参数
toupper()	如果参数是小写字符，该函数返回大写字符；否则，返回原始参数

7.2.3 多重选择 else if

现实生活中我们经常有多种选择。在程序中也可以用 else if 扩展 if else 结构模拟这种情况。来看一个特殊的例子。电力公司通常根据客户的总用电量来决定电费。下面是某电力公司的电费清单，单位是千瓦时 (kWh)：

首 360kWh:	\$0.13230/kWh
续 108kWh:	\$0.15040/kWh
续 252kWh:	\$0.30025/kWh
超过 720kWh:	\$0.34025/kWh

如果对用电管理感兴趣，可以编写一个计算电费的程序。程序清单 7.4 是完成这一任务的第一步。

程序清单 7.4 electric.c 程序

```
// electric.c -- 计算电费
#include <stdio.h>
#define RATE1 0.13230          // 首次使用 360 kWh 的费率
#define RATE2 0.15040          // 接着再使用 108 kWh 的费率
#define RATE3 0.30025          // 接着再使用 252 kWh 的费率
#define RATE4 0.34025          // 使用超过 720kwh 的费率
#define BREAK1 360.0            // 费率的第一个分界点
#define BREAK2 468.0            // 费率的第二个分界点
#define BREAK3 720.0            // 费率的第三个分界点
#define BASE1 (RATE1 * BREAK1)  // 使用 360kwh 的费用
#define BASE2 (BASE1 + (RATE2 * (BREAK2 - BREAK1)))  // 使用 468kwh 的费用
#define BASE3 (BASE1 + BASE2 + (RATE3 * (BREAK3 - BREAK2)))  // 使用 720kwh 的费用
int main(void)
{
    double kwh;                // 使用的千瓦时
    double bill;                // 电费

    printf("Please enter the kwh used.\n");
    scanf("%lf", &kwh);          // %lf 对应 double 类型
    if (kwh <= BREAK1)
        bill = RATE1 * kwh;
    else if (kwh <= BREAK2)      // 360~468 kWh
        bill = BASE1 + (RATE2 * (kwh - BREAK1));
    else if (kwh <= BREAK3)      // 468~720 kWh
        bill = BASE2 + (RATE3 * (kwh - BREAK2));
    else                          // 超过 720 kWh
        bill = BASE3 + (RATE4 * (kwh - BREAK3));
    printf("The charge for %.1f kWh is $%.2f.\n", kwh, bill);

    return 0;
}
```

该程序的输出示例如下：

```

Please enter the kwh used.
580
The charge for 580.0 kwh is $97.50.

```

程序清单 7.4 用符号常量表示不同的费率和费率分界点，以便把常量统一放在一处。这样，电力公司在更改费率以及费率分界点时，更新数据非常方便。BASE1 和 BASE2 根据费率和费率分界点来表示。一旦费率或分界点发生了变化，它们也会自动更新。预处理器是不进行计算的。程序中出现 BASE1 的地方都会被替换成 $0.13230 * 360.0$ 。不用担心，编译器会对该表达式求值得到一个数值（47.628），以便最终的程序代码使用的是 47.628 而不是一个计算式。

程序流简单明了。该程序根据 kwh 的值在 3 个公式之间选择一个。特别要注意的是，如果 kwh 大于或等于 360，程序只会到达第 1 个 else。因此，`else if (kwh <= BREAK2)` 这行相当于要求 kwh 在 360~482 之间，如程序注释所示。类似地，只有当 kwh 的值超过 720 时，才会执行最后的 else。最后，注意 BASE1、BASE2 和 BASE3 分别代表 360、468 和 720 千瓦时的总费用。因此，当电量超过这些值时，只需要加上额外的费用即可。

实际上，`else if` 是已学过的 `if else` 语句的变式。例如，该程序的核心部分只不过是下面代码的另一种写法：

```

if (kwh <= BREAK1)
    bill = RATE1 * kwh;
else
    if (kwh <= BREAK2)          // 360~468 kwh
        bill = BASE1 + (RATE2 * (kwh - BREAK1));
    else
        if (kwh <= BREAK3)      // 468~720 kwh
            bill = BASE2 + (RATE3 * (kwh - BREAK2));
        else                      // 超过 720 kwh
            bill = BASE3 + (RATE4 * (kwh - BREAK3));

```

也就是说，该程序由一个 `if else` 语句组成，`else` 部分包含另一个 `if else` 语句，该 `if else` 语句的 `else` 部分又包含另一个 `if else` 语句。第 2 个 `if else` 语句嵌套在第 1 个 `if else` 语句中，第 3 个 `if else` 语句嵌套在第 2 个 `if else` 语句中。回忆一下，整个 `if else` 语句被视为一条语句，因此不必把嵌套的 `if else` 语句用花括号括起来。当然，花括号可以更清楚地表明这种特殊格式的含义。

这两种形式完全等价。唯一不同的是使用空格和换行的位置不同，不过编译器会忽略这些。尽管如此，第 1 种形式还是好些，因为这种形式更清楚地显示了有 4 种选择。在浏览程序时，这种形式让读者更容易看清楚各项选择。在需要时要缩进嵌套的部分，例如，必须测试两个单独的量时。本例中，仅在夏季对用电量超过 720 kWh 的用户加收 10% 的电费，就属于这种情况。

可以把多个 `else if` 语句连成一串使用，如下所示（当然，要在编译器的限制范围内）：

```

if (score < 1000)
    bonus = 0;
else if (score < 1500)
    bonus = 1;
else if (score < 2000)
    bonus = 2;
else if (score < 2500)
    bonus = 4;
else
    bonus = 6;

```

（这可能是一个游戏程序的一部分，`bonus` 表示下一局游戏获得的光子炸弹或补给。）

对于编译器的限制范围，C99 标准要求编译器最少支持 127 层嵌套。

7.2.4 else 与 if 配对

如果程序中有许多 if 和 else，编译器如何知道哪个 if 对应哪个 else？例如，考虑下面的程序段：

```
if (number > 6)
    if (number < 12)
        printf("You're close!\n");
else
    printf("Sorry, you lose a turn!\n");
```

何时打印 Sorry, you lose a turn!？当 number 小于或等于 6 时，还是 number 大于 12 时？换言之，else 与第 1 个 if 还是第 2 个 if 匹配？答案是，else 与第 2 个 if 匹配。也就是说，输入的数字和匹配的响应如下：

数字	响应
5	None
10	You're close!
15	Sorry, you lose a turn!

规则是，如果没有花括号，else 与离它最近的 if 匹配，除非最近的 if 被花括号括起来（见图 7.2）。

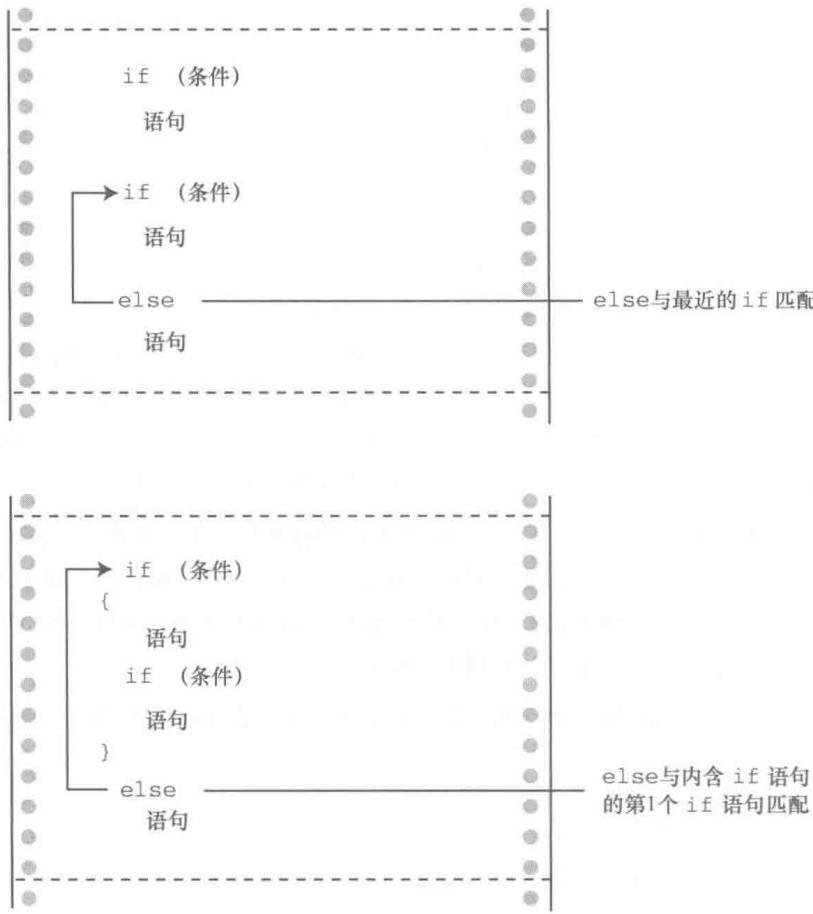


图 7.2 if else 匹配的规则

注意：要缩进“语句”，“语句”可以是一条简单语句或复合语句。

第 1 个例子的缩进使得 else 看上去与第 1 个 if 相匹配，但是记住，编译器是忽略缩进的。如果希望 else 与第 1 个 if 匹配，应该这样写：

```

if (number > 6)
{
    if (number < 12)
        printf("You're close!\n");
}
else
    printf("Sorry, you lose a turn!\n");

```

这样改动后，响应如下：

数字	响应
5	Sorry, you lose a turn!
10	You're close!
15	None

7.2.5 多层嵌套的 if 语句

前面介绍的 `if...else if...else` 序列是嵌套 `if` 的一种形式，从一系列选项中选择一个执行。有时，选择一个特定选项后又引出其他选择，这种情况可以使用另一种嵌套 `if`。例如，程序可以使用 `if else` 选择男女，`if else` 的每个分支里又包含另一个 `if else` 来区分不同收入的群体。

我们把这种形式的嵌套 `if` 应用在下面的程序中。给定一个整数，显示所有能整除它的约数。如果没有约数，则报告该数是一个素数。

在编写程序的代码之前要先规划好。首先，要总体设计一下程序。为方便起见，程序应该使用一个循环让用户能连续输入待测试的数。这样，测试一个新的数字时不必每次都要重新运行程序。下面是我们为这种循环开发的一个模型（伪代码）：

提示用户输入数字

当 `scanf()` 返回值为 1

分析该数并报告结果

提示用户继续输入

回忆一下在测试条件中使用 `scanf()`，把读取数字和判断测试条件确定是否结束循环合并在一起。

下一步，设计如何找出约数。也许最直接的方法是：

```

for (div = 2; div < num; div++)
    if (num % div == 0)
        printf("%d is divisible by %d\n", num, div);

```

该循环检查 2~`num` 之间的所有数字，测试它们是否能被 `num` 整除。但是，这个方法有点浪费时间。我们可以改进一下。例如，考虑如果 $144 \% 2$ 得 0，说明 2 是 144 的约数；如果 144 除以 2 得 72，那么 72 也是 144 的一个约数。所以，`num % div` 测试成功可以获得两个约数。为了弄清其中的原理，我们分析一下循环中得到的成对约数：2 和 72、2 和 48、4 和 36、6 和 24、8 和 18、9 和 16、12 和 12、16 和 9、18 和 8，等等。在得到 12 和 12 这对约数后，又开始得到已找到的相同约数（次序相反）。因此，不用循环到 143，在达到 12 以后就可以停止循环。这大大地节省了循环时间！

分析后发现，必须测试的数只要到 `num` 的平方根就可以了，不用到 `num`。对于 9 这样的数字，不会节约很多时间，但是对于 10000 这样的数，使用哪一种方法求约数差别很大。不过，我们不用在程序中计算平方根，可以这样编写测试条件：

```

for (div = 2; (div * div) <= num; div++)
    if (num % div == 0)
        printf("%d is divisible by %d and %d.\n", num, div, num / div);

```

如果 num 是 144，当 div = 12 时停止循环。如果 num 是 145，当 div = 13 时停止循环。

不使用平方根而用这样的测试条件，有两个原因。其一，整数乘法比求平方根快。其二，我们还没有正式介绍平方根函数。

还要解决两个问题才能准备编程。第 1 个问题，如果待测试的数是一个完全平方数怎么办？报告 144 可以被 12 和 12 整除显得有点傻。可以使用嵌套 if 语句测试 div 是否等于 num / div。如果是，程序只打印一个约数：

```
for (div = 2; (div * div) <= num; div++)
{
    if (num % div == 0)
    {
        if (div * div != num)
            printf("%d is divisible by %d and %d.\n", num, div, num / div);
        else
            printf("%d is divisible by %d.\n", num, div);
    }
}
```

注意

从技术角度看，if else 语句作为一条单独的语句，不必使用花括号。外层 if 也是一条单独的语句，也不必使用花括号。但是，当语句太长时，使用花括号能提高代码的可读性，而且还可防止今后在 if 循环中添加其他语句时忘记加花括号。

第 2 个问题，如何知道一个数字是素数？如果 num 是素数，程序流不会进入 if 语句。要解决这个问题，可以在外层循环把一个变量设置为某个值（如，1），然后在 if 语句中把该变量重新设置为 0。循环完成后，检查该变量是否是 1，如果是，说明没有进入 if 语句，那么该数就是素数。这样的变量通常称为标记 (flag)。

一直以来，C 都习惯用 int 作为标记的类型，其实新增的 _Bool 类型更合适。另外，如果在程序中包含了 stdbool.h 头文件，便可用 bool 代替 _Bool 类型，用 true 和 false 分别代替 1 和 0。

程序清单 7.5 体现了以上分析的思路。为扩大该程序的应用范围，程序用 long 类型而不是 int 类型（如果系统不支持 _Bool 类型，可以把 isPrime 的类型改为 int，并用 1 和 0 分别替换程序中的 true 和 false）。

程序清单 7.5 divisors.c 程序

```
// divisors.c -- 使用嵌套 if 语句显示一个数的约数
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    unsigned long num;           // 待测试的数
    unsigned long div;           // 可能的约数
    bool isPrime;                // 素数标记

    printf("Please enter an integer for analysis: ");
    printf("Enter q to quit.\n");
    while (scanf("%lu", &num) == 1)
    {
```

```

for (div = 2, isPrime = true; (div * div) <= num; div++)
{
    if (num % div == 0)
    {
        if ((div * div) != num)
printf("%lu is divisible by %lu and %lu.\n",
        num, div, num / div);
        else
printf("%lu is divisible by %lu.\n",
        num, div);
        isPrime = false; // 该数不是素数
    }
}
if (isPrime)
    printf("%lu is prime.\n", num);
printf("Please enter another integer for analysis; ");
printf("Enter q to quit.\n");
}
printf("Bye.\n");

return 0;
}

```

注意，该程序在 for 循环的测试表达式中使用了逗号运算符，这样每次输入新值时都可以把 isPrime 设置为 true。

下面是该程序的一个输出示例：

```

Please enter an integer for analysis; Enter q to quit.
123456789
123456789 is divisible by 3 and 41152263.
123456789 is divisible by 9 and 13717421.
123456789 is divisible by 3607 and 34227.
123456789 is divisible by 3803 and 32463.
123456789 is divisible by 10821 and 11409.
Please enter another integer for analysis; Enter q to quit.
149
149 is prime.
Please enter another integer for analysis; Enter q to quit.
2013
2013 is divisible by 3 and 671.
2013 is divisible by 11 and 183.
2013 is divisible by 33 and 61.
Please enter another integer for analysis; Enter q to quit.
q
Bye.

```

该程序会把 1 认为是素数，其实它不是。下一节将要介绍的逻辑运算符可以排除这种特殊的情况。

小结：用 if 语句进行选择

关键字：if、else

一般注解：

下面各形式中，statement 可以是一条简单语句或复合语句。表达式为真说明其值是非零值。

形式 1:

```
if (expression)
    statement
```

如果 *expression* 为真，则执行 *statement* 部分。

形式 2:

```
if (expression)
    statement1
else
    statement2
```

如果 *expression* 为真，执行 *statement1* 部分；否则，执行 *statement2* 部分。

形式 3:

```
if (expression1)
    statement1
else if (expression2)
    statement2
else
    statement3
```

如果 *expression1* 为真，执行 *statement1* 部分；如果 *expression2* 为真，执行 *statement2* 部分；否则，执行 *statement3* 部分。

示例:

```
if (legs == 4)
    printf("It might be a horse.\n");
else if (legs > 4)
    printf("It is not a horse.\n");
else /* 如果 legs < 4 */
{
    legs++;
    printf("Now it has one more leg.\n");
}
```

7.3 逻辑运算符

读者已经很熟悉了，*if* 语句和 *while* 语句通常使用关系表达式作为测试条件。有时，把多个关系表达式组合起来会很有用。例如，要编写一个程序，计算输入的一行句子中除单引号和双引号以外其他字符的数量。这种情况下可以使用逻辑运算符，并使用句点(.) 标识句子的末尾。程序清单 7.6 用一个简短的程序进行演示。

程序清单 7.6 chcount.c 程序

```
// chcount.c -- 使用逻辑与运算符
#include <stdio.h>
#define PERIOD '.'
int main(void)
{
    char ch;
    int charcount = 0;

    while ((ch = getchar()) != PERIOD)
```

```

{
    if (ch != '"' && ch != '\'')
        charcount++;
}
printf("There are %d non-quote characters.\n", charcount);

return 0;
}

```

下面是该程序的一个输出示例：

```
I didn't read the "I'm a Programming Fool" best seller.
There are 50 non-quote characters.
```

程序首先读入一个字符，并检查它是否是一个句点，因为句点标志一个句子的结束。接下来，`if` 语句的测试条件中使用了逻辑与运算符`&&`。该 `if` 语句翻译成文字是“如果待测试的字符不是双引号，并且它也不是单引号，那么 `charcount` 递增 1”。

逻辑运算符两侧的条件必须都为真，整个表达式才为真。逻辑运算符的优先级比关系运算符低，所以不必在子表达式两侧加圆括号。

C 有 3 种逻辑运算符，见表 7.3。

表 7.3 3 种逻辑运算符

逻辑运算符	含义
<code>&&</code>	与
<code> </code>	或
<code>!</code>	非

假设 `exp1` 和 `exp2` 是两个简单的关系表达式（如 `car > rat` 或 `debt == 1000`），那么：

- 当且仅当 `exp1` 和 `exp2` 都为真时，`exp1 && exp2` 才为真；
- 如果 `exp1` 或 `exp2` 为真，则 `exp1 || exp2` 为真；
- 如果 `exp1` 为假，则 `!exp1` 为真；如果 `exp1` 为真，则 `!exp1` 为假。

下面是一些具体的例子：

```
5 > 2 && 4 > 7 为假，因为只有一个子表达式为真；
5 > 2 || 4 > 7 为真，因为有一个子表达式为真；
!(4 > 7) 为真，因为 4 不大于 7。
```

顺带一提，最后一个表达式与下面的表达式等价：

```
4 <= 7
```

如果不熟悉逻辑运算符或者觉得很别扭，请记住：(练习`&&`时间) == 完美。

7.3.1 备选拼写：`iso646.h` 头文件

C 是在美国用标准美式键盘开发的语言。但是在世界各地，并非所有的键盘都有和美式键盘一样的符号。因此，C99 标准新增了可代替逻辑运算符的拼写，它们被定义在 `iso646.h` 头文件中。如果在程序中包含该头文件，便可用 `and` 代替`&&`、`or` 代替`||`、`not` 代替`!`。例如，可以把下面的代码：

```
if (ch != '"' && ch != '\'')
    charcount++;
```

改写为：

```
if (ch != '\"' && ch != '\\')
    charcount++;
```

表7.4列出了逻辑运算符对应的拼写，很容易记。读者也许很好奇，为何C不直接使用and、or和not？因为C一直坚持尽量保持较少的关键字。参考资料V“新增C99和C11的标准ANSI C库”列出了一些运算符的备选拼写，有些我们还没见过。

表7.4 逻辑运算符的备选拼写

传统写法	iso646.h
&&	and
	or
!	not

7.3.2 优先级

!运算符的优先级很高，比乘法运算符还高，与递增运算符的优先级相同，只比圆括号的优先级低。&&运算符的优先级比||运算符高，但是两者的优先级都比关系运算符低，比赋值运算符高。因此，表达式a>b && b > c || b > d相当于((a > b) && (b > c)) || (b > d)。

也就是说，b介于a和c之间，或者b大于d。

尽管对于该例没必要使用圆括号，但是许多程序员更喜欢使用带圆括号的第2种写法。这样做即使不记得逻辑运算符的优先级，表达式的含义也很清楚。

7.3.3 求值顺序

除了两个运算符共享一个运算对象的情况外，C通常不保证先对复杂表达式中哪部分求值。例如，下面的语句，可能先对表达式5+3求值，也可能先对表达式9+6求值：

```
apples = (5 + 3) * (9 + 6);
```

C把先计算哪部分的决定权留给编译器的设计者，以便针对特定系统优化设计。但是，对于逻辑运算符是个例外，C保证逻辑表达式的求值顺序是从左往右。&&和||运算符都是序列点，所以程序在从一个运算对象执行到下一个运算对象之前，所有的副作用都会生效。而且，C保证一旦发现某个元素让整个表达式无效，便立即停止求值。正是由于有这些规定，才能写出这样结构的代码：

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

如上代码所示，读取字符直至遇到第一个空格或换行符。第一个子表达式把读取的值赋给c，后面的子表达式会用到c的值。如果没有求值循序的保证，编译器可能在给c赋值之前先对后面的表达式求值。

这里还有一个例子：

```
if (number != 0 && 12/number == 2)
    printf("The number is 5 or 6.\n");
```

如果number的值是0，那么第一个子表达式为假，且不再对关系表达式求值。这样避免了把0作为除数。许多语言都没有这种特性，知道number为0后，仍继续检查后面的条件。

最后，考虑这个例子：

```
while (x++ < 10 && x + y < 20)
```

实际上，&&是一个序列点，这保证了在对&&右侧的表达式求值之前，已经递增了x。

小结：逻辑运算符和表达式

逻辑运算符：

逻辑运算符的运算对象通常是关系表达式。!运算符只需要一个运算对象，其他两个逻辑运算符都需要两个运算对象，左侧一个，右侧一个。

逻辑运算符	含义
&&	与
	或
!	非

逻辑表达式：

当且仅当 expression1 和 expression2 都为真，expression1 && expression2 才为真。如果 expression1 或 expression2 为真，expression1 || expression2 为真。如果 expression 为假，!expression 则为真，反之亦然。

求值顺序：

逻辑表达式的求值顺序是从左往右。一旦发现有使整个表达式为假的因素，立即停止求值。

示例：

6 > 2 && 3 == 3 真

!(6 > 2 && 3 == 3) 假

x != 0 && (20 / x) < 5 只有当 x 不等于 0 时，才会对第 2 个表达式求值

7.3.4 范围

&& 运算符可用于测试范围。例如，要测试 score 是否在 90~100 的范围内，可以这样写：

```
if (range >= 90 && range <= 100)
    printf("Good show!\n");
```

千万不要模仿数学上的写法：

```
if (90 <= range <= 100) // 千万不要这样写!
    printf("Good show!\n");
```

这样写的问题是代码有语义错误，而不是语法错误，所以编译器不会捕获这样的问题（虽然可能会给出警告）。由于<=运算符的求值顺序是从左往右，所以编译器把测试表达式解释为：

(90 <= range) <= 100

子表达式 90 <= range 的值要么是 1 (为真)，要么是 0 (为假)。这两个值都小于 100，所以不管 range 的值是多少，整个表达式都恒为真。因此，在范围测试中要使用&&。

许多代码都用范围测试来确定一个字符是否是小写字母。例如，假设 ch 是 char 类型的变量：

```
if (ch >= 'a' && ch <= 'z')
    printf("That's a lowercase character.\n");
```

该方法仅对于像 ASCII 这样的字符编码有效，这些编码中相邻字母与相邻数字一一对应。但是，对于像 EBCDIC 这样的代码就没用了。相应的移植方法是，用 ctype.h 系列中的 islower() 函数（参见表 7.1）：

```
if (islower(ch))
    printf("That's a lowercase character.\n");
```

无论使用哪种特定的字符编码，`islower()` 函数都能正常运行（不过，一些早期的编译器没有 `ctype.h` 系列）。

7.4 一个统计单词的程序

现在，我们可以编写一个统计单词数量的程序（即，该程序读取并报告单词的数量）。该程序还可以计算字符数和行数。先来看看编写这样的程序要涉及那些内容。

首先，该程序要逐个字符读取输入，知道何时停止读取。然后，该程序能识别并计算这些内容：字符、行数和单词。据此我们编写的伪代码如下：

读取一个字符

当有更多输入时

 递增字符计数

 如果读完一行，递增行数计数

 如果读完一个单词，递增单词计数

 读取下一个字符

前面有一个输入循环的模型：

```
while ((ch = getchar()) != STOP)
{
    ...
}
```

这里，`STOP` 表示能标识输入末尾的某个值。以前我们用过换行符和句点标记输入的末尾，但是对于一个通用的统计单词程序，它们都不合适。我们暂时选用一个文本中不常用的字符（如，`|`）作为输入的末尾标记。第 8 章中会介绍更好的方法，以便程序既能处理文本文件，又能处理键盘输入。

现在，我们考虑循环体。因为该程序使用 `getchar()` 进行输入，所以每次迭代都要通过递增计数器来计数。为了统计行数，程序要能检查换行字符。如果输入的字符是一个换行符，该程序应该递增行数计数器。这里要注意 `STOP` 字符位于一行的中间的情况。是否递增行数计数？我们可以作为特殊行计数，即没有换行符的一行字符。可以通过记录之前读取的字符识别这种情况，即如果读取时发现 `STOP` 字符的上一个字符不是换行符，那么这行就是特殊行。

最棘手的部分是识别单词。首先，必须定义什么是该程序识别的单词。我们用一个相对简单的方法，把一个单词定义为一个不含空白（即，没有空格、制表符或换行符）的字符序列。因此，“g1ymxck”和“r2d2”都算是一个单词。程序读取的第一个非空白字符即是一个单词的开始，当读到空白字符时结束。判断非空白字符最直接的测试表达式是：

```
c != ' ' && c != '\n' && c != '\t' /* 如果 c 不是空白字符，该表达式为真*/
```

检测空白字符最直接的测试表达式是：

```
c == ' ' || c == '\n' || c == '\t' /* 如果 c 是空白字符，该表达式为真*/
```

然而，使用 `ctype.h` 头文件中的函数 `isspace()` 更简单，如果该函数的参数是空白字符，则返回真。所以，如果 `c` 是空白字符，`isspace(c)` 为真；如果 `c` 不是空白字符，`!isspace(c)` 为真。

要查找一个单词里是否有某个字符，可以在程序读入单词的首字符时把一个标记（名为 `inword`）设置为 1。也可以在此时递增单词计数。然后，只要 `inword` 为 1（或 `true`），后续的非空白字符都不记为单词的开始。下一个空白字符，必须重置标记为 0（或 `false`），然后程序就准备好读取下一个单词。我们

把以上分析写成伪代码：

如果 c 不是空白字符，且 inword 为假

 设置 inword 为真，并给单词计数

如果 c 是空白字符，且 inword 为真

 设置 inword 为假

这种方法在读到每个单词的开头时把 inword 设置为 1 (真)，在读到每个单词的末尾时把 inword 设置为 0 (假)。只有在标记从 0 设置为 1 时，递增单词计数。如果能使用 _Bool 类型，可以在程序中包含 stdbool.h 头文件，把 inword 的类型设置为 bool，其值用 true 和 false 表示。如果编译器不支持这种用法，就把 inword 的类型设置为 int，其值用 1 和 0 表示。

如果使用布尔类型的变量，通常习惯把变量自身作为测试条件。如下所示：

用 if (inword) 代替 if (inword == true)

用 if (!inword) 代替 if (inword == false)

可以这样做的原因是，如果 inword 为 true，则表达式 inword == true 为 true；如果 inword 为 false，则表达式 inword == true 为 false。所以，还不如直接用 inword 作为测试条件。类似地，!inword 的值与表达式 inword == false 的值相同（非真即 false，非假即 true）。

程序清单 7.7 把上述思路（识别行、识别不完整的行和识别单词）翻译成了 C 代码。

程序清单 7.7 wordcnt.c 程序

```
// wordcnt.c -- 统计字符数、单词数、行数
#include <stdio.h>
#include <ctype.h>           // 为 isspace() 函数提供原型
#include <stdbool.h>         // 为 bool、true、false 提供定义
#define STOP '|'
int main(void)
{
    char c;                  // 读入字符
    char prev;                // 读入的前一个字符
    long n_chars = 0L; // 字符数
    int n_lines = 0;          // 行数
    int n_words = 0;          // 单词数
    int p_lines = 0;          // 不完整的行数
    bool inword = false; // 如果 c 在单词中，inword 等于 true

    printf("Enter text to be analyzed (| to terminate):\n");
    prev = '\n';              // 用于识别完整的行
    while ((c = getchar()) != STOP)
    {
        n_chars++;            // 统计字符
        if (c == '\n')
            n_lines++;          // 统计行
        if (!isspace(c) && !inword)
        {
            inword = true; // 开始一个新的单词
            n_words++;        // 统计单词
        }
    }
}
```

```

        if (isspace(c) && inword)
            inword = false;      // 打到单词的末尾
        prev = c;             // 保存字符的值
    }

    if (prev != '\n')
        p_lines = 1;
    printf("characters = %ld, words = %d, lines = %d, ",
           n_chars, n_words, n_lines);
    printf("partial lines = %d\n", p_lines);

    return 0;
}

```

下面是运行该程序后的一个输出示例：

```

Enter text to be analyzed (l to terminate):
Reason is a
powerful servant but
an inadequate master.
|
characters = 55, words = 9, lines = 3, partial lines = 0

```

该程序使用逻辑运算符把伪代码翻译成 C 代码。例如，把下面的伪代码：

如果 c 不是空白字符，且 inword 为假

翻译成如下 C 代码：

```
if (!isspace(c) && !inword)
```

再次提醒读者注意，!inword 与 inword == false 等价。上面的整个测试条件比单独判断每个空白字符的可读性高：

```
if (c != ' ' && c != '\n' && c != '\t' && !inword)
```

上面的两种形式都表示“如果 c 不是空白字符，且如果 c 不在单词里”。如果两个条件都满足，则一定是一个新单词的开头，所以要递增 n_words。如果位于单词中，满足第 1 个条件，但是 inword 为 true，就不递增 n_word。当读到下一个空白字符时，inword 被再次设置为 false。检查代码，查看一下如果单词之间有多个空格时，程序是否能正常运行。第 8 章讲解了如何修正这个问题，让该程序能统计文件中的单词量。

7.5 条件运算符：?:

C 提供条件表达式 (*conditional expression*) 作为表达 if else 语句的一种便捷方式，该表达式使用?: 条件运算符。该运算符分为两部分，需要 3 个运算对象。回忆一下，带一个运算对象的运算符称为一元运算符，带两个运算对象的运算符称为二元运算符。以此类推，带 3 个运算对象的运算符称为三元运算符。条件运算符是 C 语言中唯一的三元运算符。下面的代码得到一个数的绝对值：

```
x = (y < 0) ? -y : y;
```

在=和;之间的内容就是条件表达式，该语句的意思是“如果 y 小于 0，那么 x = -y；否则，x = y”。用 if else 可以这样表达：

```
if (y < 0)
    x = -y;
else
    x = y;
```

条件表达式的通用形式如下：

```
expression1 ? expression2 : expression3
```

如果 *expression1* 为真（非 0），那么整个条件表达式的值与 *expression2* 的值相同；如果 *expression1* 为假（0），那么整个条件表达式的值与 *expression3* 的值相同。

需要把两个值中的一个赋给变量时，就可以用条件表达式。典型的例子是，把两个值中的最大值赋给变量：

```
max = (a > b) ? a : b;
```

如果 *a* 大于 *b*，那么将 *max* 设置为 *a*；否则，设置为 *b*。

通常，条件运算符完成的任务用 *if* *else* 语句也可以完成。但是，使用条件运算符的代码更简洁，而且编译器可以生成更紧凑的程序代码。

我们来看程序清单 7.8 中的油漆程序，该程序计算刷给定平方英尺的面积需要多少罐油漆。基本算法很简单：用平方英尺数除以每罐油漆能刷的面积。但是，商店只卖整罐油漆，不会拆分来卖，所以如果计算结果是 1.7 罐，就需要两罐。因此，该程序计算得到带小数的结果时应该进 1。条件运算符常用于处理这种情况，而且还要根据单复数分别打印 *can* 和 *cans*。

程序清单 7.8 paint.c 程序

```
/* paint.c -- 使用条件运算符 */
#include <stdio.h>
#define COVERAGE 350           // 每罐油漆可刷的面积（单位：平方英尺）
int main(void)
{
    int sq_feet;
    int cans;

    printf("Enter number of square feet to be painted:\n");
    while (scanf("%d", &sq_feet) == 1)
    {
        cans = sq_feet / COVERAGE;
        cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
        printf("You need %d %s of paint.\n", cans,
               cans == 1 ? "can" : "cans");
        printf("Enter next value (q to quit):\n");
    }

    return 0;
}
```

下面是该程序的运行示例：

```
Enter number of square feet to be painted:
349
You need 1 can of paint.
Enter next value (q to quit):
351
You need 2 cans of paint.
Enter next value (q to quit):
q
```

该程序使用的变量都是 *int* 类型，除法的计算结果 (*sq_feet* / *COVERAGE*) 会被截断。也就是说， $351/350$ 得 1。所以，*cans* 被截断成整数部分。如果 *sq_feet* % *COVERAGE* 得 0，说明 *sq_feet* 被

COVERAGE 整除，cans 的值不变；否则，肯定有余数，就要给 cans 加 1。这由下面的语句完成：

```
cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
```

该语句把+=右侧表达式的值加上 cans，再赋给 cans。右侧表达式是一个条件表达式，根据 sq_feet 是否能被 COVERAGE 整除，其值为 0 或 1。

printf() 函数中的参数也是一个条件表达式：

```
cans == 1 ? "can" : "cans");
```

如果 cans 的值是 1，则打印 can；否则，打印 cans。这也说明了条件运算符的第 2 个和第 3 个运算对象可以是字符串。

小结：条件运算符

条件运算符：?:

一般注解：

条件运算符需要 3 个运算对象，每个运算对象都是一个表达式。其通用形式如下：

```
expression1 ? expression2 : expression3
```

如果 expression1 为真，整个条件表达式的值是 expression2 的值；否则，是 expression3 的值。

示例：

```
(5 > 3) ? 1 : 2 值为 1
```

```
(3 > 5) ? 1 : 2 值为 2
```

```
(a > b) ? a : b 如果 a >b, 则取较大的值
```

7.6 循环辅助：continue 和 break

一般而言，程序进入循环后，在下一次循环测试之前会执行完循环体中的所有语句。continue 和 break 语句可以根据循环体中的测试结果来忽略一部分循环内容，甚至结束循环。

7.6.1 continue 语句

3 种循环都可以使用 continue 语句。执行到该语句时，会跳过本次迭代的剩余部分，并开始下一轮迭代。如果 continue 语句在嵌套循环内，则只会影响包含该语句的内层循环。程序清单 7.9 中的简短程序演示了如何使用 continue。

程序清单 7.9 skippart.c 程序

```
/* skippart.c -- 使用 continue 跳过部分循环 */
#include <stdio.h>
int main(void)
{
    const float MIN = 0.0f;
    const float MAX = 100.0f;

    float score;
    float total = 0.0f;
    int n = 0;
    float min = MAX;
```

```

float max = MIN;

printf("Enter the first score (q to quit): ");
while (scanf("%f", &score) == 1)
{
    if (score < MIN || score > MAX)
    {
        printf("%0.1f is an invalid value. Try again: ", score);
        continue; // 跳转至 while 循环的测试条件
    }
    printf("Accepting %0.1f:\n", score);
    min = (score < min) ? score : min;
    max = (score > max) ? score : max;
    total += score;
    n++;
    printf("Enter next score (q to quit): ");
}
if (n > 0)
{
    printf("Average of %d scores is %0.1f.\n", n, total / n);
    printf("Low = %0.1f, high = %0.1f\n", min, max);
}
else
    printf("No valid scores were entered.\n");
return 0;
}

```

在程序清单 7.9 中, while 循环读取输入, 直至用户输入非数值数据。循环中的 if 语句筛选出无效的分数。假设输入 188, 程序会报告: 188 is an invalid value。在本例中, continue 语句让程序跳过处理有效输入部分的代码。程序开始下一轮循环, 准备读取下一个输入值。

注意, 有两种方法可以避免使用 continue, 一是省略 continue, 把剩余部分放在一个 else 块中:

```

if (score < 0 || score > 100)
    /* printf() 语句 */
else
{
    /* 语句 */
}

```

另一种方法是, 用以下格式来代替:

```

if (score >= 0 && score <= 100)
{
    /* 语句 */
}

```

这种情况下, 使用 continue 的好处是减少主语句组中的一级缩进。当语句很长或嵌套较多时, 紧凑简洁的格式提高了代码的可读性。

continue 还可用作占位符。例如, 下面的循环读取并丢弃输入的数据, 直至读到行末尾:

```

while (getchar() != '\n')
;

```

当程序已经读取一行中的某些内容, 要跳至下一行开始处时, 这种用法很方便。问题是, 一般很难注意到一个单独的分号。如果使用 continue, 可读性会更高:

```
while (getchar() != '\n')
    continue;
```

如果用了 `continue` 没有简化代码反而让代码更复杂，就不要使用 `continue`。例如，考虑下面的程序段：

```
while ((ch = getchar()) != '\n')
{
    if (ch == '\t')
        continue;
    putchar(ch);
}
```

该循环跳过制表符，并在读到换行符时退出循环。以上代码这样表示更简洁：

```
while ((ch = getchar()) != '\n')
    if (ch != '\t')
        putchar(ch);
```

通常，在这种情况下，把 `if` 的测试条件的关系反过来便可避免使用 `continue`。

以上介绍了 `continue` 语句让程序跳过循环体的余下部分。那么，从何处开始继续循环？对于 `while` 和 `do while` 循环，执行 `continue` 语句后的下一个行为是对循环的测试表达式求值。考虑下面的循环：

```
count = 0;
while (count < 10)
{
    ch = getchar();
    if (ch == '\n')
        continue;
    putchar(ch);
    count++;
}
```

该循环读取 10 个字符（除换行符外，因为当 `ch` 是换行符时，程序会跳过 `count++`；语句）并重新显示它们，其中不包括换行符。执行 `continue` 后，下一个被求值的表达式是循环测试条件。

对于 `for` 循环，执行 `continue` 后的下一个行为是对更新表达式求值，然后是对循环测试表达式求值。例如，考虑下面的循环：

```
for (count = 0; count < 10; count++)
{
    ch = getchar();
    if (ch == '\n')
        continue;
    putchar(ch);
}
```

该例中，执行完 `continue` 后，首先递增 `count`，然后将递增后的值和 10 作比较。因此，该循环与上面 `while` 循环的例子稍有不同。`while` 循环的例子中，除了换行符，其余字符都显示；而本例中，换行符也计算在内，所以读取的 10 个字符中包含换行符。

7.6.2 break 语句

程序执行到循环中的 `break` 语句时，会终止包含它的循环，并继续执行下一阶段。把程序清单 7.9 中的 `continue` 替换成 `break`，在输入 188 时，不是跳至执行下一轮循环，而是导致退出当前循环。图 7.3 比较了 `break` 和 `continue`。如果 `break` 语句位于嵌套循环内，它只会影响包含它的当前循环。

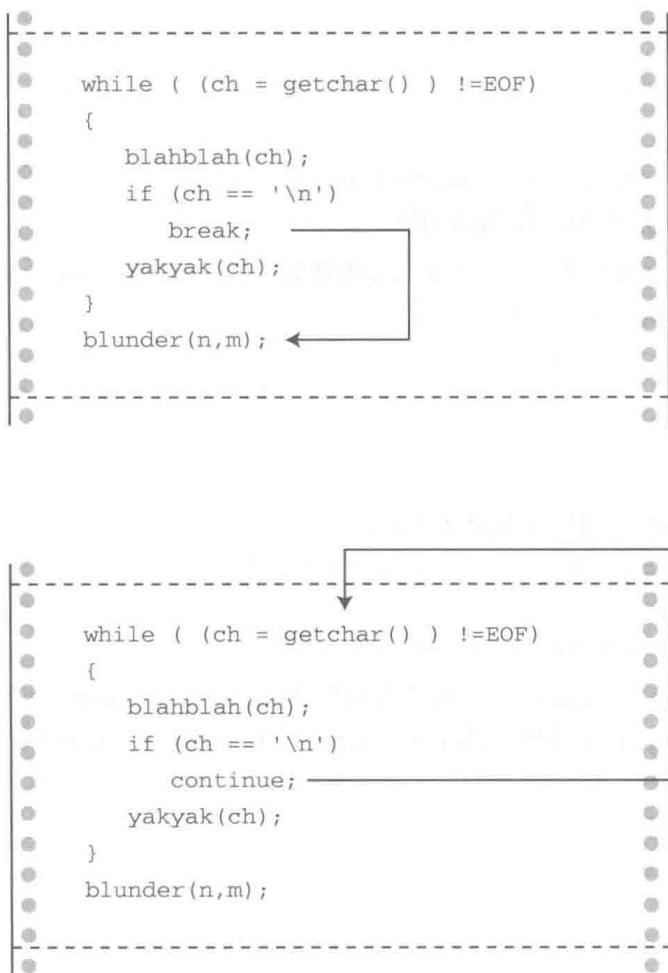


图 7.3 比较 break 和 continue

break 还可用于因其他原因退出循环的情况。程序清单 7.10 用一个循环计算矩形的面积。如果用户输入非数字作为矩形的长或宽，则终止循环。

程序清单 7.10 break.c 程序

```

/* break.c -- 使用 break 退出循环 */
#include <stdio.h>
int main(void)
{
    float length, width;

    printf("Enter the length of the rectangle:\n");
    while (scanf("%f", &length) == 1)
    {
        printf("Length = %0.2f:\n", length);
        printf("Enter its width:\n");
        if (scanf("%f", &width) != 1)
            break;
        printf("Width = %0.2f:\n", width);
        printf("Area = %0.2f:\n", length * width);
        printf("Enter the length of the rectangle:\n");
    }
    printf("Done.\n");

```

```
    return 0;
}
```

可以这样控制循环：

```
while (scanf("%f %f", &length, &width) == 2)
```

但是，用 break 可以方便显示用户输入的值。

和 continue 一样，如果用了 break 代码反而更复杂，就不要使用 break。例如，考虑下面的循环：

```
while ((ch = getchar()) != '\n')
{
    if (ch == '\t')
        break;
    putchar(ch);
}
```

如果把两个测试条件放在一起，逻辑就更清晰了：

```
while ((ch = getchar()) != '\n' && ch != '\t')
    putchar(ch);
```

break 语句对于稍后讨论的 switch 语句而言至关重要。

在 for 循环中的 break 和 continue 的情况不同，执行完 break 语句后会直接执行循环后面的第 1 条语句，连更新部分也跳过。嵌套循环内层的 break 只会让程序跳出包含它的当前循环，要跳出外层循环还需要一个 break：

```
int p, q;

scanf("%d", &p);
while (p > 0)
{
    printf("%d\n", p);
    scanf("%d", &q);
    while (q > 0)
    {
        printf("%d\n", p*q);
        if (q > 100)
            break; // 跳出内层循环
        scanf("%d", &q);
    }
    if (q > 100)
        break; // 跳出外层循环
    scanf("%d", &p);
}
```

7.7 多重选择：switch 和 break

使用条件运算符和 if else 语句很容易编写二选一的程序。然而，有时程序需要在多个选项中进行选择。可以用 if else if...else 来完成。但是，大多数情况下使用 switch 语句更方便。程序清单 7.11 演示了如何使用 switch 语句。该程序读入一个字母，然后打印出与该字母开头的动物名。

程序清单 7.11 animals.c 程序

```
/* animals.c -- 使用 switch 语句 */
#include <stdio.h>
```

```

#include <ctype.h>
int main(void)
{
    char ch;

    printf("Give me a letter of the alphabet, and I will give ");
    printf("an animal name\nbeginning with that letter.\n");
    printf("Please type in a letter; type # to end my act.\n");
    while ((ch = getchar()) != '#')
    {
        if ('\n' == ch)
            continue;
        if (islower(ch)) /* 只接受小写字母*/
            switch (ch)
        {
            case 'a':
                printf("argali, a wild sheep of Asia\n");
                break;
            case 'b':
                printf("babirusa, a wild pig of Malay\n");
                break;
            case 'c':
                printf("coati, racoonlike mammal\n");
                break;
            case 'd':
                printf("desman, aquatic, molelike critter\n");
                break;
            case 'e':
                printf("echidna, the spiny anteater\n");
                break;
            case 'f':
                printf("fisher, brownish marten\n");
                break;
            default:
                printf("That's a stumper!\n");
        } /* switch 结束 */
    }
    else
        printf("I recognize only lowercase letters.\n");
    while (getchar() != '\n')
        continue; /* 跳过输入行的剩余部分 */
    printf("Please type another letter or a #.\n");
} /* while 循环结束 */
printf("Bye!\n");

return 0;
}

```

篇幅有限，我们只编到 f，后面的字母以此类推。在进一步解释该程序之前，先看看输出示例：

```

Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
Please type in a letter; type # to end my act.
a [enter]
argali, a wild sheep of Asia
Please type another letter or a #.

```

```
dab [enter]
desman, aquatic, molelike critter
Please type another letter or a #.

r [enter]
That's a stumper!
Please type another letter or a #.

Q [enter]
I recognize only lowercase letters.
Please type another letter or a #.

# [enter]
Bye!
```

该程序的两个主要特点是：使用了 switch 语句和它对输出的处理。我们先分析 switch 的工作原理。

7.7.1 switch 语句

要对紧跟在关键字 switch 后圆括号中的表达式求值。在程序清单 7.11 中，该表达式是刚输入给 ch 的值。然后程序扫描标签（这里指，case 'a' :、case 'b' : 等）列表，直到发现一个匹配的值为止。然后程序跳转至那一行。如果没有匹配的标签怎么办？如果有 default : 标签行，就跳转至该行；否则，程序继续执行在 switch 后面的语句。

break 语句在其中起什么作用？它让程序离开 switch 语句，跳至 switch 语句后面的下一条语句（见图 7.4）。如果没有 break 语句，就会从匹配标签开始执行到 switch 末尾。例如，如果删除该程序中的所有 break 语句，运行程序后输入 d，其交互的输出结果如下：

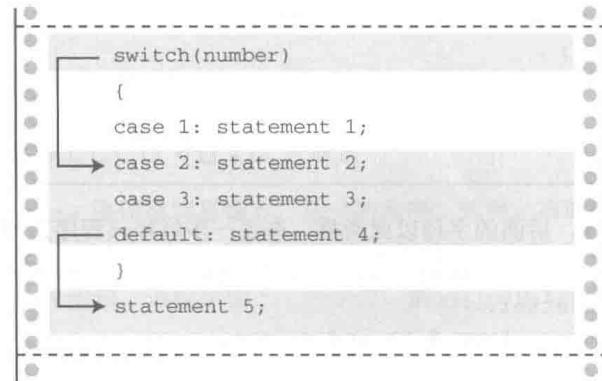
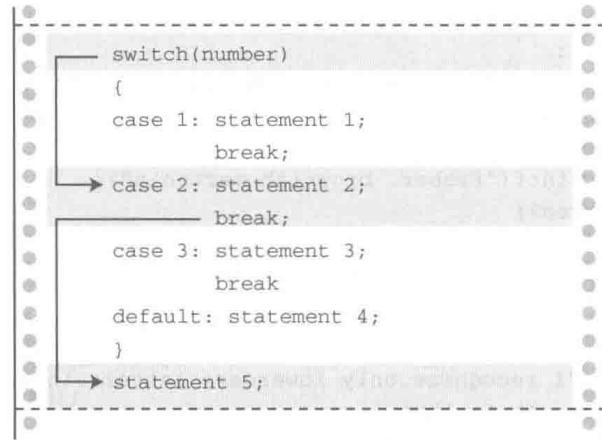


图 7.4 switch 中有 break 和没有 break 的程序流

```

Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
Please type in a letter; type # to end my act.
d [enter]
desman, aquatic, molelike critter
echidna, the spiny anteater
fisher, a brownish marten
That's a stumper!
Please type another letter or a #.
# [enter]
Bye!

```

如上所示，执行了从 case 'd': 到 switch 语句末尾的所有语句。

顺带一提，break 语句可用于循环和 switch 语句中，但是 continue 只能用于循环中。尽管如此，如果 switch 语句在一个循环中，continue 便可作为 switch 语句的一部分。这种情况下，就像在其他循环中一样，continue 让程序跳出循环的剩余部分，包括 switch 语句的其他部分。

如果读者熟悉 Pascal，会发现 switch 语句和 Pascal 的 case 语句类似。它们最大的区别在于，如果只希望处理某个带标签的语句，就必须在 switch 语句中使用 break 语句。另外，C 语言的 case 一般都指定一个值，不能使用一个范围。

switch 在圆括号中的测试表达式的值应该是一个整数值（包括 char 类型）。case 标签必须是整数类型（包括 char 类型）的常量或整型常量表达式（即，表达式中只包含整型常量）。不能用变量作为 case 标签。switch 的构造如下：

```

switch ( 整型表达式 )
{
    case 常量 1:
        语句      <--可选
    case 常量 2:
        语句      <--可选
    default :
        语句      <--可选
}

```

7.7.2 只读每行的首字符

animals.c（程序清单 7.11）的另一个独特之处是它读取输入的方式。运行程序时读者可能注意到了，当输入 dab 时，只处理了第 1 个字符。这种丢弃一行中其他字符的行为，经常出现在响应单字符的交互程序中。可以用下面的代码实现这样的行为：

```

while (getchar() != '\n')
    continue; /* 跳过输入行的其余部分 */

```

循环从输入中读取字符，包括按下 Enter 键产生的换行符。注意，函数的返回值并没有赋给 ch，以上代码所做的只是读取并丢弃字符。由于最后丢弃的字符是换行符，所以下一个被读取的字符是下一行的首字母。在外层的 while 循环中，getchar() 读取首字母并赋给 ch。

假设用户一开始就按下 Enter 键，那么程序读到的首个字符就是换行符。下面的代码处理这种情况：

```

if (ch == '\n')
    continue;

```

7.7.3 多重标签

如程序清单 7.12 所示，可以在 switch 语句中使用多重 case 标签。

程序清单 7.12 vowels.c 程序

```
// vowels.c -- 使用多重标签
#include <stdio.h>
int main(void)
{
    char ch;
    int a_ct, e_ct, i_ct, o_ct, u_ct;

    a_ct = e_ct = i_ct = o_ct = u_ct = 0;

    printf("Enter some text; enter # to quit.\n");
    while ((ch = getchar()) != '#')
    {
        switch (ch)
        {
            case 'a':
            case 'A': a_ct++;
                        break;
            case 'e':
            case 'E': e_ct++;
                        break;
            case 'i':
            case 'I': i_ct++;
                        break;
            case 'o':
            case 'O': o_ct++;
                        break;
            case 'u':
            case 'U': u_ct++;
                        break;
            default: break;
        } // switch 结束
    } // while 循环结束
    printf("number of vowels: A   E   I   O   U\n");
    printf("%d %d %d %d %d\n",
           a_ct, e_ct, i_ct, o_ct, u_ct);

    return 0;
}
```

假设如果 ch 是字母 i, switch 语句会定位到标签为 case 'i' : 的位置。由于该标签没有关联 break 语句，所以程序流直接执行下一条语句，即 i_ct++;。如果 ch 是字母 I, 程序流会直接定位到 case 'I' :。本质上，两个标签都指的是相同的语句。

严格地说，case 'U' 的 break 语句并不需要。因为即使删除这条 break 语句，程序流会接着执行 switch 中的下一条语句，即 default : break;。所以，可以把 case 'U' 的 break 语句去掉以缩短代码。但是从另一方面看，保留这条 break 语句可以防止以后在添加新的 case (例如，把 y 作为元音) 时遗漏 break 语句。

下面是该程序的运行示例:

```
Enter some text; enter # to quit.
I see under the overseer.#  
number of vowels:      A    E    I    O    U  
                      0    7    1    1    1
```

在该例中, 如果使用 ctype.h 系列的 toupper() 函数(参见表 7.2)可以避免使用多重标签, 在进行测试之前就把字母转换成大写字母:

```
while ((ch = getchar()) != '#')  
{  
    ch = toupper(ch);  
    switch (ch)  
    {  
        case 'A': a_ct++;  
                    break;  
        case 'E': e_ct++;  
                    break;  
        case 'I': i_ct++;  
                    break;  
        case 'O': o_ct++;  
                    break;  
        case 'U': u_ct++;  
                    break;  
        default: break;  
    } // switch 结束  
} // while 循环结束
```

或者, 也可以先不转换 ch, 把 toupper(ch) 放进 switch 的测试条件中: switch(toupper(ch))。

小结: 带多重选择的 switch 语句

关键字: switch

一般注解:

程序根据 expression 的值跳转至相应的 case 标签处。然后, 执行剩下的所有语句, 除非执行到 break 语句进行重定向。expression 和 case 标签都必须是整数值(包括 char 类型), 标签必须是常量或完全由常量组成的表达式。如果没有 case 标签与 expression 的值匹配, 控制则转至标有 default 的语句(如果有的话); 否则, 将转至执行紧跟在 switch 语句后面的语句。

形式:

```
switch ( expression )  
{  
    case label1 : statement1 // 使用 break 跳出 switch  
    case label2 : statement2  
    default      : statement3  
}
```

可以有多个标签语句, default 语句可选。

示例:

```
switch (choice)  
{  
    case 1 :  
    case 2 : printf("Darn tootin'!\n"); break;  
    case 3 : printf("Quite right!\n");  
    case 4 : printf("Good show!\n"); break;
```

```
default: printf("Have a nice day.\n");
}
```

如果 choice 的值是 1 或 2，打印第 1 条消息；如果 choice 的值是 3，打印第 2 条和第 3 条消息（程序继续执行后续的语句，因为 case 3 后面没有 break 语句）；如果 choice 的值是 4，则打印第 3 条消息；如果 choice 的值是其他值只打印最后一条消息。

7.7.4 switch 和 if else

何时使用 switch？何时使用 if else？你经常会别无选择。如果是根据浮点类型的变量或表达式来选择，就无法使用 switch。如果根据变量在某范围内决定程序流的去向，使用 switch 就很麻烦，这种情况用 if 就很方便：

```
if (integer < 1000 && integer > 2)
```

使用 switch 要涵盖以上范围，需要为每个整数（3~999）设置 case 标签。但是，如果使用 switch，程序通常运行快一些，生成的代码少一些。

7.8 goto 语句

早期版本的 BASIC 和 FORTRAN 所依赖的 goto 语句，在 C 中仍然可用。但是 C 和其他两种语言不同，没有 goto 语句 C 程序也能运行良好。Kernighan 和 Ritchie 提到 goto 语句“易被滥用”，并建议“谨慎使用，或者根本不用”。首先，介绍一下如何使用 goto 语句；然后，讲解为什么通常不需要它。

goto 语句有两部分：goto 和标签名。标签的命名遵循变量命名规则，如下所示：

```
goto part2;
```

要让这条语句正常工作，函数还必须包含另一条标为 part2 的语句，该语句以标签名后紧跟一个冒号开始：

```
part2: printf("Refined analysis:\n");
```

7.8.1 避免使用 goto

原则上，根本不用在 C 程序中使用 goto 语句。但是，如果你曾经学过 FORTRAN 或 BASIC（goto 对这两种语言而言都必不可少），可能还会依赖用 goto 来编程。为了帮助你克服这个习惯，我们先概述一些使用 goto 的常见情况，然后再介绍 C 的解决方案。

■ 处理包含多条语句的 if 语句：

```
if (size > 12)
    goto a;
goto b;
a: cost = cost * 1.05;
flag = 2;
b: bill = cost * flag;
```

对于以前的 BASIC 和 FORTRAN，只有直接跟在 if 条件后面的一条语句才属于 if，不能使用块或复合语句。我们把以上模式转换成等价的 C 代码，标准 C 用复合语句或块来处理这种情况：

```
if (size > 12)
{
    cost = cost * 1.05;
    flag = 2;
```

```

}
bill = cost * flag;

```

■ 二选一：

```

if (ibex > 14)
    goto a;
sheds = 2;
goto b;
a: sheds= 3;
b: help = 2 * sheds;

```

C 通过 if else 表达二选一更清楚：

```

if (ibex > 14)
    sheds = 3;
else
    sheds = 2;
help = 2 * sheds;

```

实际上，新版的 BASIC 和 FORTRAN 已经把 else 纳入新的语法中。

■ 创建不确定循环：

```

readin: scanf("%d", &score);
if (score < 0)
    goto stage2;
lots of statements
goto readin;
stage2: more stuff;

```

C 用 while 循环代替：

```

scanf("%d", &score);
while (score <= 0)
{
    lots of statements
    scanf("%d", &score);
}
more stuff;

```

■ 跳转至循环末尾，并开始下一轮迭代。C 使用 continue 语句代替。

■ 跳出循环。C 使用 break 语句。实际上，break 和 continue 是 goto 的特殊形式。使用 break 和 continue 的好处是：其名称已经表明它们的用法，而且这些语句不使用标签，所以不用担心把标签放错位置导致的危险。

■ 胡乱跳转至程序的不同部分。简而言之，不要这样做！

但是，C 程序员可以接受一种 goto 的用法——出现问题时从一组嵌套循环中跳出（一条 break 语句只能跳出当前循环）：

```

while (funct > 0)
{
    for (i = 1, i <= 100; i++)
    {
        for (j = 1; j <= 50; j++)
        {
            其他语句
            if (问题)
                goto help;
            其他语句
        }
    }
}

```

```

    }
    其他语句
}
其他语句
}
其他语句
help: 语句

```

从其他例子中也能看出，程序中使用其他形式比使用 `goto` 的条理更清晰。当多种情况混在一起时，这种差异更加明显。哪些 `goto` 语句可以帮助 `if` 语句？哪些可以模仿 `if else`？哪些控制循环？哪些是因为程序无路可走才不得已放在那里？过度地使用 `goto` 语句，会让程序错综复杂。如果不熟悉 `goto` 语句，就不要使用它。如果已经习惯使用 `goto` 语句，试着改掉这个毛病。讽刺地是，虽然 C 根本不需要 `goto`，但是它的 `goto` 比其他语言的 `goto` 好用，因为 C 允许在标签中使用描述性的单词而不是数字。

小结：程序跳转

关键字：`break`、`continue`、`goto`

一般注解：

这 3 种语句都能使程序流从程序的一处跳转至另一处。

break 语句：

所有的循环和 `switch` 语句都可以使用 `break` 语句。它使程序控制跳出当前循环或 `switch` 语句的剩余部分，并继续执行跟在循环或 `switch` 后面的语句。

示例：

```

switch (number)
{
    case 4: printf("That's a good choice.\n");
              break;
    case 5: printf("That's a fair choice.\n");
              break;
    default: printf("That's a poor choice.\n");
}

```

continue 语句：

所有的循环都可以使用 `continue` 语句，但是 `switch` 语句不行。`continue` 语句使程序控制跳出循环的剩余部分。对于 `while` 或 `for` 循环，程序执行到 `continue` 语句后会开始进入下一轮迭代。对于 `do while` 循环，对出口条件求值后，如有必要会进入下一轮迭代。

示例：

```

while ((ch = getchar()) != '\n')
{
    if (ch == ' ')
        continue;
    putchar(ch);
    chcount++;
}

```

以上程序段把用户输入的字符再次显示在屏幕上，并统计非空格字符。

goto 语句：

`goto` 语句使程序控制跳转至相应标签语句。冒号用于分隔标签和标签语句。标签名遵循变量命名规则。标签语句可以出现在 `goto` 的前面或后面。

形式:

```
goto label ;
.
.
.
```

label : statement

示例:

```
top : ch = getchar();
.
.
.
if (ch != 'y')
goto top;
```

7.9 关键概念

智能的一个方面是，根据情况做出相应的响应。所以，选择语句是开发具有智能行为程序的基础。C 语言通过 if、if else 和 switch 语句，以及条件运算符 (?:) 可以实现智能选择。

if 和 if else 语句使用测试条件来判断执行哪些语句。所有非零值都被视为 true，零被视为 false。测试通常涉及关系表达式（比较两个值）、逻辑表达式（用逻辑运算符组合或更改其他表达式）。

要记住一个通用原则，如果要测试两个条件，应该使用逻辑运算符把两个完整的测试表达式组合起来。例如，下面这些是错误的：

```
if (a < x < z)           // 错误，没有使用逻辑运算符
...
if (ch != 'q' && != 'Q') // 错误，缺少完整的测试表达式
...
```

正确的方式是用逻辑运算符连接两个关系表达式：

```
if (a < x && x < z)       // 使用&&组合两个表达式
...
if (ch != 'q' && ch != 'Q') // 使用&&组合两个表达式
...
```

对比这两章和前几章的程序示例可以发现：使用第 6 章、第 7 章介绍的语句，可以写出功能更强大、更有趣的程序。

7.10 本章小结

本章介绍了很多内容，我们来总结一下。if 语句使用测试条件控制程序是否执行测试条件后面的一条简单语句或复合语句。如果测试表达式的值是非零值，则执行语句；如果测试表达式的值是零，则不执行语句。if else 语句可用于二选一的情况。如果测试条件是非零，则执行 else 前面的语句；如果测试表达式的值是零，则执行 else 后面的语句。在 else 后面使用另一个 if 语句形成 else if，可构造多选一的结构。

测试条件通常都是关系表达式，即用一个关系运算符（如，<或==）的表达式。使用 C 的逻辑运算符，可以把关系表达式组合成更复杂的测试条件。

在多数情况下，用条件运算符 (?:) 写成的表达式比 if else 语句更简洁。

`ctype.h` 系列的字符函数（如，`isspace()` 和 `isalpha()`）为创建以分类字符为基础的测试表达式提供了便捷的工具。

`switch` 语句可以在一系列以整数作为标签的语句中进行选择。如果紧跟在 `switch` 关键字后的测试条件的整数值与某标签匹配，程序就转至执行匹配的标签语句，然后在遇到 `break` 之前，继续执行标签语句后面的语句。

`break`、`continue` 和 `goto` 语句都是跳转语句，使程序流跳转至程序的另一处。`break` 语句使程序跳转至紧跟在包含 `break` 语句的循环或 `switch` 末尾的下一条语句。`continue` 语句使程序跳出当前循环的剩余部分，并开始下一轮迭代。

7.11 复习题

复习题的参考答案在附录 A 中。

1. 判断下列表达式是 true 还是 false。

- a. `100 > 3 && 'a' > 'c'`
- b. `100 > 3 || 'a' > 'c'`
- c. `!(100 > 3)`

2. 根据下列描述的条件，分别构造一个表达式：

- a. number 等于或大于 90，但是小于 100
- b. ch 不是字符 q 或 k
- c. number 在 1~9 之间（包括 1 和 9），但不是 5
- d. number 不在 1~9 之间

3. 下面的程序关系表达式过于复杂，而且还有些错误，请简化并改正。

```
#include <stdio.h>
int main(void)                                /* 1 */
{
    int weight, height; /* weight 以磅为单位, height 以英寸为单位 */
/* 4 */
    scanf("%d , weight, height);                /* 5 */
    if (weight < 100 && height > 64)           /* 6 */
        if (height >= 72)                         /* 7 */
            printf("You are very tall for your weight.\n");
        else if (height < 72 && > 64)             /* 9 */
            printf("You are tall for your weight.\n"); /* 10 */
        else if (weight > 300 && !(weight <= 300)) /* 11 */
            && height < 48)                         /* 12 */
        if (!(height >= 48))                      /* 13 */
            printf(" You are quite short for your weight.\n");
        else                                         /* 15 */
            printf("Your weight is ideal.\n");       /* 16 */
/* 17 */

    return 0;
}
```

4. 下列个表达式的值是多少？

- a. `5 > 2`
- b. `3 + 4 > 2 && 3 < 2`
- c. `x >= y || y > x`

- d. $d = 5 + (6 > 2)$
 e. $'X' > 'T' ? 10 : 5$
 f. $x > y ? y > x : x > y$

5. 下面的程序将打印什么？

```
#include <stdio.h>
int main(void)
{
    int num;
    for (num = 1; num <= 11; num++)
    {
        if (num % 3 == 0)
            putchar('$');
        else
            putchar('*');
            putchar('#');
            putchar('%');
    }
    putchar('\n');
    return 0;
}
```

6. 下面的程序将打印什么？

```
#include <stdio.h>
int main(void)
{
    int i = 0;
    while (i < 3) {
        switch (i++) {
            case 0: printf("fat ");
            case 1: printf("hat ");
            case 2: printf("cat ");
            default: printf("Oh no!");
        }
        putchar('\n');
    }
    return 0;
}
```

7. 下面的程序有哪些错误？

```
#include <stdio.h>
int main(void)
{
    char ch;
    int lc = 0; /* 统计小写字母 */
    int uc = 0; /* 统计大写字母 */
    int oc = 0; /* 统计其他字母 */

    while ((ch = getchar()) != '#')
    {
        if ('a' <= ch >= 'z')
            lc++;
        else if (!(ch < 'A') || !(ch > 'Z'))
            uc++;
        oc++;
    }
}
```

```

    }
    printf("%d lowercase, %d uppercase, %d other, lc, uc, oc);
    return 0;
}

```

8. 下面的程序将打印什么？

```

/* retire.c */
#include <stdio.h>
int main(void)
{
    int age = 20;
    while (age++ <= 65)
    {
        if ((age % 20) == 0) /* age 是否能被 20 整除? */
            printf("You are %d. Here is a raise.\n", age);
        if (age == 65)
            printf("You are %d. Here is your gold watch.\n", age);
    }
    return 0;
}

```

9. 给定下面的输入时，以下程序将打印什么？

```

q
c
h
b
#include <stdio.h>
int main(void)
{
    char ch;

    while ((ch = getchar()) != '#')
    {
        if (ch == '\n')
            continue;
        printf("Step 1\n");
        if (ch == 'c')
            continue;
        else if (ch == 'b')
            break;
        else if (ch == 'h')
            goto laststep;
        printf("Step 2\n");
    laststep: printf("Step 3\n");
    }
    printf("Done\n");
    return 0;
}

```

10. 重写复习题 9，但这次不能使用 continue 和 goto 语句。

7.12 编程练习

- 编写一个程序读取输入，读到#字符停止，然后报告读取的空格数、换行符数和其他字符的数量。

2. 编写一个程序读取输入，读到#字符停止。程序要打印每个输入的字符以及对应的 ASCII 码（十进制）。一行打印 8 个字符。建议：使用字符计数和求模运算符（%）在每 8 个循环周期时打印一个换行符。
3. 编写一个程序，读取整数直到用户输入 0。输入结束后，程序应报告用户输入的偶数（不包括 0）个数、这些偶数的平均值、输入的奇数个数及其奇数的平均值。
4. 使用 if else 语句编写一个程序读取输入，读到#停止。用感叹号替换句号，用两个感叹号替换原来的感叹号，最后报告进行了多少次替换。
5. 使用 switch 重写练习 4。
6. 编写程序读取输入，读到#停止，报告 ei 出现的次数。

注意

该程序要记录前一个字符和当前字符。用“Receive your eieio award”这样的输入来测试。

7. 编写一个程序，提示用户输入一周工作的小时数，然后打印工资总额、税金和净收入。做如下假设：
 - a. 基本工资 = 1000 美元/小时
 - b. 加班（超过 40 小时）= 1.5 倍的时间
 - c. 税率：前 300 美元为 15%
续 150 美元为 20%
余下的为 25%

用#define 定义符号常量。不用在意是否符合当前的税法。

8. 修改练习 7 的假设 a，让程序可以给出一个供选择的工资等级菜单。使用 switch 完成工资等级选择。运行程序后，显示的菜单应该类似这样：

```
*****
Enter the number corresponding to the desired pay rate or action:
1) $8.75/hr           2) $9.33/hr
3) $10.00/hr          4) $11.20/hr
5) quit
*****
```

如果选择 1~4 其中的一个数字，程序应该询问用户工作的小时数。程序要通过循环运行，除非用户输入 5。如果输入 1~5 以外的数字，程序应提醒用户输入正确的选项，然后再重复显示菜单提示用户输入。使用#define 创建符号常量表示各工资等级和税率。

9. 编写一个程序，只接受正整数输入，然后显示所有小于或等于该数的素数。
10. 1988 年的美国联邦税收计划是近代最简单的税收方案。它分为 4 个类别，每个类别有两个等级。下面是该税收计划的摘要（美元数为应征税的收入）：

类别	税金
单身	17850 美元按 15% 计，超出部分按 28% 计
户主	23900 美元按 15% 计，超出部分按 28% 计
已婚，共有	29750 美元按 15% 计，超出部分按 28% 计
已婚，离异	14875 美元按 15% 计，超出部分按 28% 计

例如，一位工资为 20000 美元的单身纳税人，应缴纳税费 $0.15 \times 17850 + 0.28 \times (20000 - 17850)$ 美元。编写一个程序，让用户指定缴纳税金的种类和应纳税收入，然后计算税金。程序应通过循环让用户可以多次输入。

11. ABC 邮购杂货店出售的洋蓟售价为 2.05 美元/磅，甜菜售价为 1.15 美元/磅，胡萝卜售价为 1.09 美元/磅。在添加运费之前，100 美元的订单有 5% 的打折优惠。少于或等于 5 磅的订单收取 6.5 美元的运费和包装费，5 磅~20 磅的订单收取 14 美元的运费和包装费，超过 20 磅的订单在 14 美元的基础上每续重 1 磅增加 0.5 美元。编写一个程序，在循环中用 switch 语句实现用户输入不同的字母时有不同的响应，即输入 a 的响应是让用户输入洋蓟的磅数，b 是甜菜的磅数，c 是胡萝卜的磅数，q 是退出订购。程序要记录累计的重量。即，如果用户输入 4 磅的甜菜，然后输入 5 磅的甜菜，程序应报告 9 磅的甜菜。然后，该程序要计算货物总价、折扣（如果有的话）、运费和包装费。随后，程序应显示所有的购买信息：物品售价、订购的重量（单位：磅）、订购的蔬菜费用、订单的总费用、折扣（如果有的话）、运费和包装费，以及所有的费用总额。

字符输入/输出和输入验证

本章介绍以下内容：

- 更详细地介绍输入、输出以及缓冲输入和无缓冲输入的区别
- 如何通过键盘模拟文件结尾条件
- 如何使用重定向把程序和文件相连接
- 创建更友好的用户界面

在涉及计算机的话题时，我们经常会提到输入 (*input*) 和输出 (*output*)。我们谈论输入和输出设备（如键盘、U 盘、扫描仪和激光打印机），讲解如何处理输入数据和输出数据，讨论执行输入和输出任务的函数。本章主要介绍用于输入和输出的函数（简称 I/O 函数）。

I/O 函数（如 `printf()`、`scanf()`、`getchar()`、`putchar()` 等）负责把信息传送到程序中。前几章简单介绍过这些函数，本章将详细介绍它们的基本概念。同时，还会介绍如何设计与用户交互的界面。

最初，输入/输出函数不是 C 定义的一部分，C 把开发这些函数的任务留给编译器的实现者来完成。在实际应用中，UNIX 系统中的 C 实现为这些函数提供了一个模型。ANSI C 库吸取成功的经验，把大量的 UNIX I/O 函数囊括其中，包括一些我们曾经用过的。由于必须保证这些标准函数在不同的计算机环境中能正常工作，所以它们很少使用某些特殊系统才有的特性。因此，许多 C 供应商会利用硬件的特性，额外提供一些 I/O 函数。其他函数或函数系列需要特殊的操作系统支持，如 Windows 或 Macintosh OS 提供的特殊图形界面。这些有针对性、非标准的函数让程序员能更有效地使用特定计算机编写程序。本章只着重讲解所有系统都通用的标准 I/O 函数，用这些函数编写的可移植程序很容易从一个系统移植到另一个系统。处理文件输入/输出的程序也可以使用这些函数。

许多程序都有输入验证，即判断用户的输入是否与程序期望的输入匹配。本章将演示一些与输入验证相关的问题和解决方案。

8.1 单字符 I/O: `getchar()` 和 `putchar()`

第 7 章中提到过，`getchar()` 和 `putchar()` 每次只处理一个字符。你可能认为这种方法实在太笨拙了，毕竟与我们的阅读方式相差甚远。但是，这种方法很适合计算机。而且，这是绝大多数文本（即，普通文字）处理程序所用的核心方法。为了帮助读者回忆这些函数的工作方式，请看程序清单 8.1。该程序获取从键盘输入的字符，并把这些字符发送到屏幕上。程序使用 `while` 循环，当读到#字符时停止。

程序清单 8.1 `echo.c` 程序

```
/* echo.c -- 重复输入 */
#include <stdio.h>
int main(void)
{
```

```

char ch;

while ((ch = getchar()) != '#')
    putchar(ch);

return 0;
}

```

自从 ANSI C 标准发布以后，C 就把 stdio.h 头文件与使用 getchar() 和 putchar() 相关联，这就是为什么程序中要包含这个头文件的原因（其实，getchar() 和 putchar() 都不是真正的函数，它们被定义为供预处理器使用的宏，我们在第 16 章中再详细讨论）。运行该程序后，与用户的交互如下：

```

Hello, there. I would[enter]
Hello, there. I would
like a #3 bag of potatoes.[enter]
like a

```

读者可能好奇，为何输入的字符能直接显示在屏幕上？如果用一个特殊字符（如，#）来结束输入，就无法在文本中使用这个字符，是否有更好的方法结束输入？要回答这些问题，首先要了解 C 程序如何处理键盘输入，尤其是缓冲和标准输入文件的概念。

8.2 缓冲区

如果在老式系统运行程序清单 8.1，你输入文本时可能显示如下：

```
HHeelllloo,, tthheerree.. II wwoouulldd[enter]
```

```
lliikkee aa #
```

以上行为是个例外。像这样回显用户输入的字符后立即重复打印该字符是属于无缓冲（或直接）输入，即正在等待的程序可立即使用输入的字符。对于该例，大部分系统在用户按下 Enter 键之前不会重复打印刚输入的字符，这种输入形式属于缓冲输入。用户输入的字符被收集并储存在一个被称为缓冲区（buffer）的临时存储区，按下 Enter 键后，程序才可使用用户输入的字符。图 8.1 比较了这两种输入。

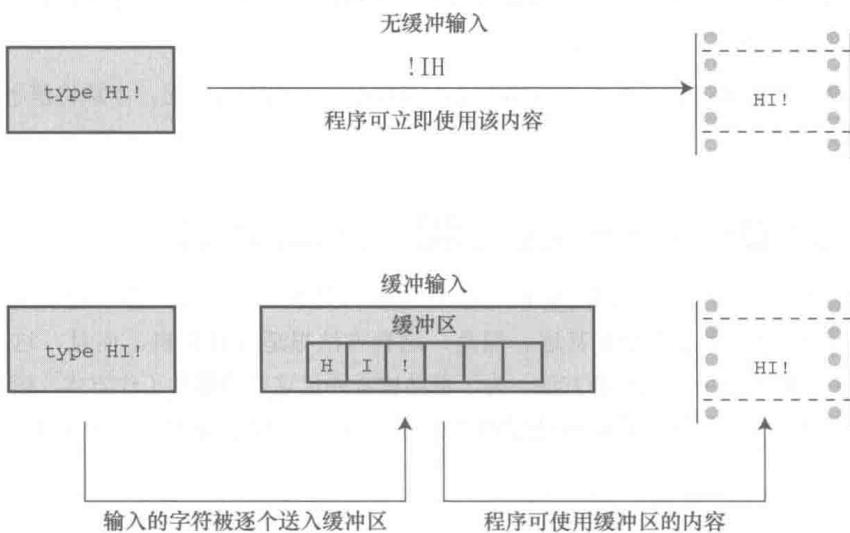


图 8.1 缓冲输入和无缓冲输入

为什么要有缓冲区？首先，把若干字符作为一个块进行传输比逐个发送这些字符节约时间。其次，如果用户打错字符，可以直接通过键盘修正错误。当最后按下 Enter 键时，传输的是正确的输入。

虽然缓冲输入好处很多，但是某些交互式程序也需要无缓冲输入。例如，在游戏中，你希望按下一个键就执行相应的指令。因此，缓冲输入和无缓冲输入都有用武之地。

缓冲分为两类：完全缓冲 I/O 和行缓冲 I/O。完全缓冲输入指的是当缓冲区被填满时才刷新缓冲区（内容被发送至目的地），通常出现在文件输入中。缓冲区的大小取决于系统，常见的大小是 512 字节和 4096 字节。行缓冲 I/O 指的是在出现换行符时刷新缓冲区。键盘输入通常是行缓冲输入，所以在按下 **Enter** 键后才刷新缓冲区。

那么，使用缓冲输入还是无缓冲输入？ANSI C 和后续的 C 标准都规定输入是缓冲的，不过最初 K&R 把这个决定权交给了编译器的编写者。读者可以运行 `echo.c` 程序观察输出的情况，了解所用的输出类型。

ANSI C 决定把缓冲输入作为标准的原因是：一些计算机不允许无缓冲输入。如果你的计算机允许无缓冲输入，那么你所用的 C 编译器很可能会提供一个无缓冲输入的选项。例如，许多 IBM PC 兼容机的编译器都为支持无缓冲输入提供一系列特殊的函数，其原型都在 `conio.h` 头文件中。这些函数包括用于回显无缓冲输入的 `getche()` 函数和用于无回显无缓冲输入的 `getch()` 函数（回显输入意味着用户输入的字符直接显示在屏幕上，无回显输入意味着击键后对应的字符不显示）。UNIX 系统使用另一种不同的方式控制缓冲。在 UNIX 系统中，可以使用 `ioctl()` 函数（该函数属于 UNIX 库，但是不属于 C 标准）指定待输入的类型，然后用 `getchar()` 执行相应的操作。在 ANSI C 中，用 `setbuf()` 和 `setvbuf()` 函数（详见第 13 章）控制缓冲，但是受限于一些系统的内部设置，这些函数可能不起作用。总之，ANSI 没有提供调用无缓冲输入的标准方式，这意味着是否能进行无缓冲输入取决于计算机系统。在这里要对使用无缓冲输入的朋友说声抱歉，本书假设所有的输入都是缓冲输入。

8.3 结束键盘输入

在 `echo.c` 程序（程序清单 8.1）中，只要输入的字符中不含#，那么程序在读到#时才会结束。但是，#也是一个普通的字符，有时不可避免要用到。应该用一个在文本中用不到的字符来标记输入完成，这样的字符不会无意间出现在输入中，在你不希望结束程序的时候终止程序。C 的确提供了这样的字符，不过在此之前，先来了解一下 C 处理文件的方式。

8.3.1 文件、流和键盘输入

文件 (*file*) 是存储器中储存信息的区域。通常，文件都保存在某种永久存储器中（如，硬盘、U 盘或 DVD 等）。毫无疑问，文件对于计算机系统相当重要。例如，你编写的 C 程序就保存在文件中，用来编译 C 程序的程序也保存在文件中。后者说明，某些程序需要访问指定的文件。当编译储存在名为 `echo.c` 文件中的程序时，编译器打开 `echo.c` 文件并读取其中的内容。当编译器处理完后，会关闭该文件。其他程序，例如文字处理器，不仅要打开、读取和关闭文件，还要把数据写入文件。

C 是一门强大、灵活的语言，有许多用于打开、读取、写入和关闭文件的库函数。从较低层面上，C 可以使用主机操作系统的基本文件工具直接处理文件，这些直接调用操作系统的函数被称为底层 I/O (*low-level I/O*)。由于计算机系统各不相同，所以不可能为普通的底层 I/O 函数创建标准库，ANSI C 也不打算这样做。然而从较高层面上，C 还可以通过标准 I/O 包 (*standard I/O package*) 来处理文件。这涉及创建用于处理文件的标准模型和一套标准 I/O 函数。在这一层面上，具体的 C 实现负责处理不同系统的差异，以便用户使用统一的界面。

上面讨论的差异指的是什么？例如，不同的系统储存文件的方式不同。有些系统把文件的内容储存在一处，而文件相关的信息储存在另一处；有些系统在文件中创建一份文件描述。在处理文件方面，有些系统使用单个换行符标记行末尾，而其他系统可能使用回车符和换行符的组合来表示行末尾。有些系统用最

小字节来衡量文件的大小，有些系统则以字节块的大小来衡量。

如果使用标准 I/O 包，就不用考虑这些差异。因此，可以用 `if (ch == '\n')` 检查换行符。即使系统实际用的是回车符和换行符的组合来标记行末尾，I/O 函数会在两种表示法之间相互转换。

从概念上看，C 程序处理的是流而不是直接处理文件。流（stream）是一个实际输入或输出映射的理想化数据流。这意味着不同属性和不同种类的输入，由属性更统一的流来表示。于是，打开文件的过程就是把流与文件相关联，而且读写都通过流来完成。

第 13 章将更详细地讨论文件。本章着重理解 C 把输入和输出设备视为存储设备上的普通文件，尤其是把键盘和显示设备视为每个 C 程序自动打开的文件。`stdin` 流表示键盘输入，`stdout` 流表示屏幕输出。`getchar()`、`putchar()`、`printf()` 和 `scanf()` 函数都是标准 I/O 包的成员，处理这两个流。

以上讨论的内容说明，可以用处理文件的方式来处理键盘输入。例如，程序读文件时要能检测文件的末尾才知道应在何处停止。因此，C 的输入函数内置了文件结尾检测器。既然可以把键盘输入视为文件，那么也应该能使用文件结尾检测器结束键盘输入。下面我们从文件开始，学习如何结束文件。

8.3.2 文件结尾

计算机操作系统要以某种方式判断文件的开始和结束。检测文件结尾的一种方法是，在文件末尾放一个特殊的字符标记文件结尾。CP/M、IBM-DOS 和 MS-DOS 的文本文件曾经用过这种方法。如今，这些操作系统可以使用内嵌的 `Ctrl+Z` 字符来标记文件结尾。这曾经是操作系统使用的唯一标记，不过现在有一些其他的选择，例如记录文件的大小。所以现代的文本文件不一定有嵌入的 `Ctrl+Z`，但是如果有，该操作系统会将其视为一个文件结尾标记。图 8.2 演示了这种方法。

散文原文：

```
Ishphat the robot
slid open the hatch
and shouted his challenge.
```

文件中的散文：

```
Ishphat the robot\n slid open the hatch\n and shouted his challenge.\n^Z
```

图 8.2 带文件结尾标记的文件

操作系统使用的另一种方法是储存文件大小的信息。如果文件有 3000 字节，程序在读到 3000 字节时便达到文件的末尾。MS-DOS 及其相关系统使用这种方法处理二进制文件，因为用这种方法可以在文件中储存所有的字符，包括 `Ctrl+Z`。新版的 DOS 也使用这种方法处理文本文件。UNIX 使用这种方法处理所有的文件。

无论操作系统实际使用何种方法检测文件结尾，在 C 语言中，用 `getchar()` 读取文件检测到文件结尾时将返回一个特殊的值，即 EOF（end of file 的缩写）。`scanf()` 函数检测到文件结尾时也返回 EOF。通常，EOF 定义在 `stdio.h` 文件中：

```
#define EOF (-1)
```

为什么是 -1？因为 `getchar()` 函数的返回值通常都介于 0~127，这些值对应标准字符集。但是，如果系统能识别扩展字符集，该函数的返回值可能在 0~255 之间。无论哪种情况，-1 都不对应任何字符，所以，该值可用于标记文件结尾。

某些系统也许把 EOF 定义为 -1 以外的值，但是定义的值一定与输入字符所产生的返回值不同。如果包

含 stdio.h 文件，并使用 EOF 符号，就不必担心 EOF 值不同的问题。这里关键要理解 EOF 是一个值，标志着检测到文件结尾，并不是在文件中找得到的符号。

那么，如何在程序中使用 EOF？把 getchar() 的返回值和 EOF 作比较。如果两值不同，就说明没有到达文件结尾。也就是说，可以使用下面这样的表达式：

```
while ((ch = getchar()) != EOF)
```

如果正在读取的是键盘输入不是文件会怎样？绝大部分系统（不是全部）都有办法通过键盘模拟文件结尾条件。了解这些以后，读者可以重写程序清单 8.1 的程序，如程序清单 8.2 所示。

程序清单 8.2 echo_eof.c 程序

```
/* echo_eof.c -- 重复输入，直到文件结尾 */
#include <stdio.h>
int main(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);

    return 0;
}
```

注意下面几点。

- 不用定义 EOF，因为 stdio.h 中已经定义过了。
- 不用担心 EOF 的实际值，因为 EOF 在 stdio.h 中用#define 预处理指令定义，可直接使用，不必再编写代码假定 EOF 为某值。
- 变量 ch 的类型从 char 变为 int，因为 char 类型的变量只能表示 0~255 的无符号整数，但是 EOF 的值是 -1。还好，getchar() 函数实际返回值的类型是 int，所以它可以读取 EOF 字符。如果实现使用有符号的 char 类型，也可以把 ch 声明为 char 类型，但最好还是用更通用的形式。
- 由于 getchar() 函数的返回类型是 int，如果把 getchar() 的返回值赋给 char 类型的变量，一些编译器会警告可能丢失数据。
- ch 是整数不会影响 putchar()，该函数仍然会打印等价的字符。
- 使用该程序进行键盘输入，要设法输入 EOF 字符。不能只输入字符 EOF，也不能只输入 -1（输入 -1 会传送两个字符：一个连字符和一个数字 1）。正确的方法是，必须找出当前系统的要求。例如，在大多数 UNIX 和 Linux 系统中，在一行开始处按下 Ctrl+D 会传输文件结尾信号。许多微型计算机系统都把一行开始处的 Ctrl+Z 识别为文件结尾信号，一些系统把任意位置的 Ctrl+Z 解释成文件结尾信号。

下面是在 UNIX 系统下运行 echo_eof.c 程序的缓冲示例：

```
She walks in beauty, like the night
She walks in beauty, like the night
Of cloudless skies and starry skies...
Of cloudless skies and starry skies...
Lord Byron
Lord Byron
[Ctrl+D]
```

每次按下 Enter 键，系统便会处理缓冲区中储存的字符，并在下一行打印该输入行的副本。这个过程一直持续到以 UNIX 风格模拟文件结尾（按下 Ctrl+D）。在 PC 中，要按下 Ctrl+Z。

我们暂停一会儿。既然 echo_eof.c 程序能把用户输入的内容拷贝到屏幕上，那么考虑一下该程序还可以做什么。假设以某种方式把一个文件传送给它，然后它把文件中的内容打印在屏幕上，当到达文件结尾发现 EOF 信号时停止。或者，假设以某种方式把程序的输出定向到一个文件，然后通过键盘输入数据，用 echo_eof.c 来储存在文件中输入的内容。假设同时使用这两种方法：把输入从一个文件定向到 echo_eof.c 中，并把输出发送至另一个文件，然后便可以使用 echo_eof.c 来拷贝文件。这个小程序有查看文件内容、创建一个新文件、拷贝文件的潜力，没想到一个小程序竟然如此多才多艺！关键是要控制输入流和输出流，这是我们下一个要讨论的主题。

注意 模拟 EOF 和图形界面

模拟 EOF 的概念是在使用文本界面的命令行环境中产生的。在这种环境中，用户通过击键与程序交互，由操作系统生成 EOF 信号。但是在一些实际应用中，却不能很好地转换成图形界面（如 Windows 和 Macintosh），这些用户界面包含更复杂的鼠标移动和按钮点击。程序要模拟 EOF 的行为依赖于编译器和项目类型。例如，Ctrl+Z 可以结束输入或整个程序，这取决于特定的设置。

8.4 重定向和文件

输入和输出涉及函数、数据和设备。例如，考虑 echo_eof.c，该程序使用输入函数 getchar()。输出设备（我们假设）是键盘，输入数据流由字符组成。假设你希望输入函数和数据类型不变，仅改变程序查找数据的位置。那么，程序如何知道去哪里查找输入？

在默认情况下，C 程序使用标准 I/O 包查找标准输入作为输入源。这就是前面介绍过的 stdin 流，它是把数据读入计算机的常用方式。它可以是一个过时的设备，如磁带、穿孔卡或电传打印机，或者（假设）是键盘，甚至是一些先进技术，如语音输入。然而，现代计算机非常灵活，可以让它到处查找输入。尤其是，可以让一个程序从文件中查找输入，而不是从键盘。

程序可以通过两种方式使用文件。第 1 种方法是，显式使用特定的函数打开文件、关闭文件、读取文件、写入文件，诸如此类。我们在第 13 章中再详细介绍这种方法。第 2 种方法是，设计能与键盘和屏幕互动的程序，通过不同的渠道重定向输入至文件和从文件输出。换言之，把 stdin 流重新赋给文件。继续使用 getchar() 函数从输入流中获取数据，但它并不关心从流的什么位置获取数据。虽然这种重定向的方法在某些方面有些限制，但是用起来比较简单，而且能让读者熟悉普通的文件处理技术。

重定向的一个主要问题与操作系统有关，与 C 无关。尽管如此，许多 C 环境中（包括 UNIX、Linux 和 Windows 命令提示模式）都有重定向特性，而且一些 C 实现还在某些缺乏重定向特性的系统中模拟它。在 UNIX 上运行苹果 OS X，可以用 UNIX 命令行模式启动 Terminal 应用程序。接下来我们介绍 UNIX、Linux 和 Windows 的重定向。

8.4.1 UNIX、Linux 和 DOS 重定向

UNIX（运行命令行模式时）、Linux（ditto）和 Window 命令行提示（模仿旧式 DOS 命令行环境）都能重定向输入、输出。重定向输入让程序使用文件而不是键盘来输入，重定向输出让程序输出至文件而不是屏幕。

1. 重定向输入

假设已经编译了 echo_eof.c 程序，并把可执行版本放入一个名为 echo_eof（或者在 Windows 系统中名为 echo_eof.exe）的文件中。运行该程序，输入可执行文件名：

```
echo_eof
```

该程序的运行情况和前面描述的一样，获取用户从键盘输入的输入。现在，假设你要用该程序处理名为 words 的文本文件。文本文件 (*text file*) 是内含文本的文件，其中储存的数据是我们可识别的字符。文件的内容可以是一篇散文或者 C 程序。内含机器语言指令的文件（如储存可执行程序的文件）不是文本文件。由于该程序的操作对象是字符，所以要使用文本文档。只需用下面的命令代替上面的命令即可：

```
echo_eof < words
```

< 符号是 UNIX 和 DOS/Windows 的重定向运算符。该运算符使 words 文件与 stdin 流相关联，把文件中的内容导入 echo_eof 程序。echo_eof 程序本身并不知道（或不关心）输入的内容是来自文件还是键盘，它只知道这是需要导入的字符流，所以它读取这些内容并把字符逐个打印在屏幕上，直至读到文件结尾。因为 C 把文件和 I/O 设备放在一个层面，所以文件就是现在的 I/O 设备。试试看！

注意 重定向

对于 UNIX、Linux 和 Windows 命令提示，<两侧的空格是可选的。一些系统，如 AmigaDOS（那些喜欢怀旧的人使用的系统），支持重定向，但是在重定向符号和文件名之间不允许有空格。

下面是一个特殊的 words 文件的运行示例，\$ 是 UNIX 和 Linux 的标准提示符。在 Windows/DOS 系统中见到的 DOS 提示可能是 A> 或 C>。

```
$ echo_eof < words
The world is too much with us: late and soon,
Getting and spending, we lay waste our powers:
Little we see in Nature that is ours;
We have given our hearts away, a sordid boon!
$
```

2. 重定向输出

现在假设要用 echo_eof 把键盘输入的内容发送到名为 mywords 的文件中。然后，输入以下命令并开始输入：

```
echo_eof>mywords
```

> 符号是第 2 个重定向运算符。它创建了一个名为 mywords 的新文件，然后把 echo_eof 的输出（即，你输入字符的副本）重定向至该文件中。重定向把 stdout 从显示设备（即，显示器）赋给 mywords 文件。如果已经有一个名为 mywords 的文件，通常会擦除该文件的内容，然后替换新的内容（但是，许多操作系统有保护现有文件的选项，使其成为只读文件）。所有出现在屏幕的字母都是你刚才输入的，其副本储存在文件中。在下一行的开始处按下 Ctrl+D（UNIX）或 Ctrl+Z（DOS）即可结束该程序。如果不知道输入什么内容，可参照下面的示例。这里，我们使用 UNIX 提示符 \$。记住在每行的末尾单击 Enter 键，这样才能把缓冲区的内容发送给程序。

```
$ echo_eof > mywords
You should have no problem recalling which redirection
operator does what. Just remember that each operator points
in the direction the information flows. Think of it as
a funnel.
[Ctrl+D]
$
```

按下 Ctrl+D 或 Ctrl+Z 后，程序会结束，你的系统会提示返回。程序是否起作用了？UNIX 的 ls 命令或 Windows 命令行提示模式的 dir 命令可以列出文件名，会显示 mywords 文件已存在。可以使用 UNIX 或 Linux 的 cat 或 DOS 的 type 命令检查文件中的内容，或者再次使用 echo_eof，这次把文件重定向到程序：

```
$ echo_eof < mywords
You should have no problem recalling which redirection
operator does what. Just remember that each operator points
in the direction the information flows. Think of it as a
funnel.
$
```

3. 组合重定向

现在，假设你希望制作一份 mywords 文件的副本，并命名为 savewords。只需输入以下命令即可：

```
echo_eof < mywords > savewords
```

下面的命令也起作用，因为命令与重定向运算符的顺序无关：

```
echo_eof > savewords < mywords
```

注意：在一条命令中，输入文件名和输出文件名不能相同。

```
echo_eof < mywords > mywords....<--错误
```

原因是> mywords 在输入之前已导致原 mywords 的长度被截断为 0。

总之，在 UNIX、Linux 或 Windows/DOS 系统中使用两个重定向运算符（<和>）时，要遵循以下原则。

- 重定向运算符连接一个可执行程序（包括标准操作系统命令）和一个数据文件，不能用于连接一个数据文件和另一个数据文件，也不能用于连接一个程序和另一个程序。
- 使用重定向运算符不能读取多个文件的输入，也不能把输出定向至多个文件。
- 通常，文件名和运算符之间的空格不是必须的，除非是偶尔在 UNIX shell、Linux shell 或 Windows 命令行提示模式中使用的有特殊含义的字符。例如，我们用过的 echo_eof<words。

以上介绍的都是正确的例子，下面来看一下错误的例子，addup 和 count 是两个可执行程序，fish 和 beets 是两个文本文件：

fish > beets	←违反第 1 条规则
addup < count	←违反第 1 条规则
addup < fish < beets	←违反第 2 条规则
count > beets fish	←违反第 2 条规则

UNIX、Linux 或 Windows/DOS 还有>>运算符，该运算符可以把数据添加到现有文件的末尾，而 | 运算符能把一个文件的输出连接到另一个文件的输入。欲了解所有相关运算符的内容，请参阅 UNIX 的相关书籍，如 *UNIX Primer Plus, Third Edition* (Wilson、Pierce 和 Wessler 合著)。

4. 注释

重定位让你能使用键盘输入程序文件。要完成这一任务，程序要测试文件的末尾。例如，第 7 章演示的统计单词程序（程序清单 7.7），计算单词个数直至遇到第 1 个|字符。把 ch 的 char 类型改成 int 类型，把循环测试中的|替换成 EOF，便可用该程序来计算文本文件中的单词量。

重定向是一个命令行概念，因为我们要在命令行输入特殊的符号发出指令。如果不使用命令行环境，也可以使用重定向。首先，一些集成开发环境提供了菜单选项，让用户指定重定向。其次，对于 Windows 系统，可以打开命令提示窗口，并在命令行运行可执行文件。Microsoft Visual Studio 的默认设置是把可执行文件放在项目文件夹的子文件夹，称为 Debug。文件名和项目名的基本名相同，文件名的扩展名为.exe。

默认情况下，Xcode 在给项目命名后才能命名可执行文件，并将其放在 Debug 文件夹中。在 UNIX 系统中，可以通过 Terminal 工具运行可执行文件。从使用上看，Terminal 比命令行编译器（GCC 或 Clang）简单。

如果用不了重定向，可以用程序直接打开文件。程序清单 8.3 演示了一个注释较少的示例。我们学到第 13 章时再详细讲解。待读取的文件应该与可执行文件位于同一目录。

程序清单 8.3 file_eof.c 程序

```
// file_eof.c --打开一个文件并显示该文件
#include <stdio.h>
#include <stdlib.h>           // 为了使用 exit()
int main()
{
    int ch;
    FILE * fp;
    char fname[50];           // 储存文件名

    printf("Enter the name of the file: ");
    scanf("%s", fname);
    fp = fopen(fname, "r");    // 打开待读取文件
    if (fp == NULL)           // 如果失败
    {
        printf("Failed to open file. Bye\n");
        exit(1);              // 退出程序
    }
    // getc(fp)从打开的文件中获取一个字符
    while ((ch = getc(fp)) != EOF)
        putchar(ch);
    fclose(fp);               // 关闭文件

    return 0;
}
```

小结：如何重定向输入和输出

绝大部分 C 系统都可以使用重定向，可以通过操作系统重定向所有程序，或只在 C 编译器允许的情况下重定向 C 程序。假设 prog 是可执行程序名，file1 和 file2 是文件名。

把输出重定向至文件：>

prog >file1

把输入重定向至文件：<

prog <file2

组合重定向：

prog <file2 >file1

prog >file1 <file2

这两种形式都是把 file2 作为输入、file1 作为输出。

留白：

一些系统要求重定向运算符左侧有一个空格，右侧没有空格。而其他系统（如，UNIX）允许在重定位运算符两侧有空格或没有空格。

8.5 创建更友好的用户界面

大部分人偶尔会写一些中看不中用的程序。还好，C 提供了大量工具让输入更顺畅，处理过程更顺利。不过，学习这些工具会导致新的问题。本节的目标是，指导读者解决这些问题并创建更友好的用户界面，让交互数据输入更方便，减少错误输入的影响。

8.5.1 使用缓冲输入

缓冲输入用起来比较方便，因为在把输入发送给程序之前，用户可以编辑输入。但是，在使用输入的字符时，它也会给程序员带来麻烦。前面示例中看到的问题是，缓冲输入要求用户按下 **Enter** 键发送输入。这一动作也传送了换行符，程序必须妥善处理这个麻烦的换行符。我们以一个猜谜程序为例。用户选择一个数字，程序猜用户选中的数字是多少。该程序使用的方法单调乏味，先不要在意算法，我们关注的重点在输入和输出。查看程序清单 8.4，这是猜谜程序的最初版本，后面我们会改进。

程序清单 8.4 guess.c 程序

```
/* guess.c -- 一个拖沓且错误的猜数字程序 */
#include <stdio.h>
int main(void)
{
    int guess = 1;

    printf("Pick an integer from 1 to 100. I will try to guess ");
    printf("it.\nRespond with a y if my guess is right and with");
    printf("\nn if it is wrong.\n");
    printf("Uh...is your number %d?\n", guess);
    while (getchar() != 'y')      /* 获取响应，与 y 做对比 */
        printf("Well, then, is it %d?\n", ++guess);
    printf("I knew I could do it!\n");

    return 0;
}
```

下面是程序的运行示例：

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
Well, then, is it 3?
n
Well, then, is it 4?
Well, then, is it 5?
y
I knew I could do it!
```

撇开这个程序糟糕的算法不谈，我们先选择一个数字。注意，每次输入 **n** 时，程序打印了两条消息。这是由于程序读取 **n** 作为用户否定了数字 1，然后还读取了一个换行符作为用户否定了数字 2。

一种解决方案是，使用 **while** 循环丢弃输入行最后剩余的内容，包括换行符。这种方法的优点是，能把 **no** 和 **no way** 这样的响应视为简单的 **n**。程序清单 8.4 的版本会把 **no** 当作两个响应。下面用循环修正

这个问题：

```
while (getchar() != 'y') /* 获得响应，与 y 做对比 */
{
    printf("Well, then, is it %d?\n", ++guess);
    while (getchar() != '\n')
        continue; /* 跳过剩余的输入行 */
}
```

使用以上循环后，该程序的输出示例如下：

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
no
Well, then, is it 3?
no sir
Well, then, is it 4?
forget it
Well, then, is it 5?
y
I knew I could do it!
```

这的确是解决了换行符的问题。但是，该程序还是会把 f 被视为 n。我们用 if 语句筛选其他响应。首先，添加一个 char 类型的变量储存响应：

```
char response;
```

修改后的循环如下：

```
while ((response = getchar()) != 'y') /* 获得响应 */
{
    if (response == 'n')
        printf("Well, then, is it %d?\n", ++guess);
    else
        printf("Sorry, I understand only y or n.\n");
    while (getchar() != '\n')
        continue; /* 跳过剩余的输入行 */
}
```

现在，程序的运行示例如下：

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
no
Well, then, is it 3?
no sir
Well, then, is it 4?
forget it
Sorry, I understand only y or n.
n
Well, then, is it 5?
y
I knew I could do it!
```

在编写交互式程序时，应该事先预料到用户可能会输入错误，然后设计程序处理用户的错误输入。在用户出错时通知用户再次输入。

当然，无论你的提示写得多么清楚，总会有人误解，然后抱怨这个程序设计得多么糟糕。

8.5.2 混合数值和字符输入

假设程序要求用 `getchar()` 处理字符输入，用 `scanf()` 处理数值输入，这两个函数都能很好地完成任务，但是不能把它们混用。因为 `getchar()` 读取每个字符，包括空格、制表符和换行符；而 `scanf()` 在读取数字时则会跳过空格、制表符和换行符。

我们通过程序清单 8.5 来解释这种情况导致的问题。该程序读入一个字符和两个数字，然后根据输入的两个数字指定的行数和列数打印该字符。

程序清单 8.5 showchar1.c 程序

```
/* showchar1.c -- 有较大 I/O 问题的程序 */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;           /* 待打印字符 */
    int rows, cols;  /* 行数和列数 */
    printf("Enter a character and two integers:\n");
    while ((ch = getchar()) != '\n')
    {
        scanf("%d %d", &rows, &cols);
        display(ch, rows, cols);
        printf("Enter another character and two integers:\n");
        printf("Enter a newline to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}

void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n'); /* 结束一行并开始新的一行 */
    }
}
```

注意，该程序以 `int` 类型读取字符（这样做可以检测 EOF），但是却以 `char` 类型把字符传递给 `display()` 函数。因为 `char` 比 `int` 小，一些编译器会给出类型转换的警告。可以忽略这些警告，或者用下面的强制类型转换消除警告：

```
display(char(ch), rows, cols);
```

在该程序中，`main()` 负责获取数据，`display()` 函数负责打印数据。下面是该程序的一个运行示例，

看看有什么问题：

```
Enter a character and two integers:
c 2 3
ccc
ccc
Enter another character and two integers;
Enter a newline to quit.
Bye.
```

该程序开始时运行良好。你输入 c 2 3，程序打印 c 字符 2 行 3 列。然后，程序提示输入第 2 组数据，还没等你输入数据程序就退出了！这是什么情况？又是换行符在捣乱，这次是输入行中紧跟在 3 后面的换行符。scanf() 函数把这个换行符留在输入队列中。和 scanf() 不同，getchar() 不会跳过换行符，所以在进入下一轮迭代时，你还没来得及输入字符，它就读取了换行符，然后将其赋给 ch。而 ch 是换行符正式终止循环的条件。

要解决这个问题，程序要跳过一轮输入结束与下一轮输入开始之间的所有换行符或空格。另外，如果该程序不在 getchar() 测试时，而在 scanf() 阶段终止程序会更好。修改后的版本如程序清单 8.6 所示。

程序清单 8.6 showchar2.c 程序

```
/* showchar2.c -- 按指定的行列打印字符 */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;                      /* 待打印字符 */
    int rows, cols;              /* 行数和列数 */

    printf("Enter a character and two integers:\n");
    while ((ch = getchar()) != '\n')
    {
        if (scanf("%d %d", &rows, &cols) != 2)
            break;
        display(ch, rows, cols);
        while (getchar() != '\n')
            continue;
        printf("Enter another character and two integers:\n");
        printf("Enter a newline to quit.\n");
    }
    printf("Bye.\n");
}

return 0;
}

void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n');      /* 结束一行并开始新的一行 */
    }
}
```

`while` 循环实现了丢弃 `scanf()` 输入后面所有字符（包括换行符）的功能，为循环的下一轮读取做好了准备。该程序的运行示例如下：

```
Enter a character and two integers:
c 1 2
cc
Enter another character and two integers;
Enter a newline to quit.
! 3 6
!!!!!
!!!!!
!!!!!
Enter another character and two integers;
Enter a newline to quit.

Bye.
```

在 `if` 语句中使用一个 `break` 语句，可以在 `scanf()` 的返回值不等于 2 时终止程序，即如果一个或两个输入值不是整数或者遇到文件结尾就终止程序。

8.6 输入验证

在实际应用中，用户不一定会按照程序的指令行事。用户的输入和程序期望的输入不匹配时常发生，这会导致程序运行失败。作为程序员，除了完成编程的本职工作，还要事先预料一些可能的输入错误，这样才能编写出能检测并处理这些问题的程序。

例如，假设你编写了一个处理非负数整数的循环，但是用户很可能输入一个负数。你可以使用关系表达式来排除这种情况：

```
long n;
scanf("%ld", &n);           // 获取第 1 个值
while (n >= 0)             // 检测不在范围内的值
{
    // 处理 n
    scanf("%ld", &n); // 获取下一个值
}
```

另一类潜在的陷阱是，用户可能输入错误类型的值，如字符 `q`。排除这种情况的一种方法是，检查 `scanf()` 的返回值。回忆一下，`scanf()` 返回成功读取项的个数。因此，下面的表达式当且仅当用户输入一个整数时才为真：

```
scanf("%ld", &n) == 1
```

结合上面的 `while` 循环，可改进为：

```
long n;
while (scanf("%ld", &n) == 1 && n >= 0)
{
    // 处理 n
}
```

`while` 循环条件可以描述为“当输入是一个整数且该整数为正时”。

对于最后的例子，当用户输入错误类型的值时，程序结束。然而，也可以让程序友好些，提示用户再次输入正确类型的值。在这种情况下，要处理有问题的输入。如果 `scanf()` 没有成功读取，就会将其留在输入队列中。这里要明确，输入实际上是字符流。可以使用 `getchar()` 函数逐字符地读取输入，甚至可以

把这些想法都结合在一个函数中，如下所示：

```
long get_long(void)
{
    long input;
    char ch;
    while (scanf("%ld", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch); // 处理错误的输入
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}
```

该函数要把一个 int 类型的值读入变量 input 中。如果读取失败，函数则进入外层 while 循环体。然后内层循环逐字符地读取错误的输入。注意，该函数丢弃该输入行的所有剩余内容。还有一个方法是，只丢弃下一个字符或单词，然后该函数提示用户再次输入。外层循环重复运行，直到用户成功输入整数，此时 scanf() 的返回值为 1。

在用户输入整数后，程序可以检查该值是否有效。考虑一个例子，要求用户输入一个上限和一个下限来定义值的范围。在该例中，你可能希望程序检查第 1 个值是否大于第 2 个值（通常假设第 1 个值是较小的那个值），除此之外还要检查这些值是否在允许的范围内。例如，当前的档案查找一般不会接受 1958 年以前和 2014 年以后的查询任务。这个限制可以在一个函数中实现。

假设程序中包含了 stdbool.h 头文件。如果当前系统不允许使用 _Bool，把 bool 替换成 int，把 true 替换成 1，把 false 替换成 0 即可。注意，如果输入无效，该函数返回 true，所以函数名为 bad_limits()：

```
bool bad_limits(long begin, long end, long low, long high)
{
    bool not_good = false;
    if (begin > end)
    {
        printf("%ld isn't smaller than %ld.\n", begin, end);
        not_good = true;
    }
    if (begin < low || end < low)
    {
        printf("Values must be %ld or greater.\n", low);
        not_good = true;
    }
    if (begin > high || end > high)
    {
        printf("Values must be %ld or less.\n", high);
        not_good = true;
    }

    return not_good;
}
```

程序清单 8.7 使用了上面的两个函数为一个进行算术运算的函数提供整数，该函数计算特定范围内所有整数的平方和。程序限制了范围的上限是 10000000，下限是-10000000。

程序清单 8.7 checking.c 程序

```

// checking.c -- 输入验证
#include <stdio.h>
#include <stdbool.h>
// 验证输入是一个整数
long get_long(void);
// 验证范围的上下限是否有效
bool bad_limits(long begin, long end,
                 long low, long high);
// 计算 a~b 之间的整数平方和
double sum_squares(long a, long b);
int main(void)
{
    const long MIN = -10000000L;      // 范围的下限
    const long MAX = +10000000L;      // 范围的上限
    long start;                      // 用户指定的范围最小值
    long stop;                       // 用户指定的范围最大值
    double answer;

    printf("This program computes the sum of the squares of "
           "integers in a range.\nThe lower bound should not "
           "be less than -10000000 and\nthe upper bound "
           "should not be more than +10000000.\nEnter the "
           "limits (enter 0 for both limits to quit):\n"
           "lower limit: ");
    start = get_long();
    printf("upper limit: ");
    stop = get_long();
    while (start != 0 || stop != 0)
    {
        if (bad_limits(start, stop, MIN, MAX))
            printf("Please try again.\n");
        else
        {
            answer = sum_squares(start, stop);
            printf("The sum of the squares of the integers ");
            printf("from %ld to %ld is %g\n",
                   start, stop, answer);
        }
        printf("Enter the limits (enter 0 for both "
               "limits to quit):\n");
        printf("lower limit: ");
        start = get_long();
        printf("upper limit: ");
        stop = get_long();
    }
    printf("Done.\n");

    return 0;
}

long get_long(void)
{
    long input;

```

```

char ch;

while (scanf("%ld", &input) != 1)
{
    while ((ch = getchar()) != '\n')
        putchar(ch); // 处理错误输入
    printf(" is not an integer.\nPlease enter an ");
    printf("integer value, such as 25, -178, or 3: ");
}

return input;
}

double sum_squares(long a, long b)
{
    double total = 0;
    long i;

    for (i = a; i <= b; i++)
        total += (double) i * (double) i;

    return total;
}

bool bad_limits(long begin, long end,
    long low, long high)
{
    bool not_good = false;

    if (begin > end)
    {
        printf("%ld isn't smaller than %ld.\n", begin, end);
        not_good = true;
    }
    if (begin < low || end < low)
    {
        printf("Values must be %ld or greater.\n", low);
        not_good = true;
    }
    if (begin > high || end > high)
    {
        printf("Values must be %ld or less.\n", high);
        not_good = true;
    }
}

return not_good;
}

```

下面是该程序的输出示例：

```

This program computes the sum of the squares of integers in a range.
The lower bound should not be less than -10000000 and
the upper bound should not be more than +10000000.
Enter the limits (enter 0 for both limits to quit):
lower limit: low
low is not an integer.

```

```

Please enter an integer value, such as 25, -178, or 3: 3
upper limit: a big number
a big number is not an integer.
Please enter an integer value, such as 25, -178, or 3: 12
The sum of the squares of the integers from 3 to 12 is 645
Enter the limits (enter 0 for both limits to quit):
lower limit: 80
upper limit: 10
80 isn't smaller than 10.
Please try again.
Enter the limits (enter 0 for both limits to quit):
lower limit: 0
upper limit: 0
Done.

```

8.6.1 分析程序

虽然 `checking.c` 程序的核心计算部分 (`sum_squares()` 函数) 很短, 但是输入验证部分比以往程序示例要复杂。接下来分析其中的一些要素, 先着重讨论程序的整体结构。

程序遵循模块化的编程思想, 使用独立函数 (模块) 来验证输入和管理显示。程序越大, 使用模块化编程就越重要。

`main()` 函数管理程序流, 为其他函数委派任务。它使用 `get_long()` 获取值、`while` 循环处理值、`badlimits()` 函数检查值是否有效、`sum_squares()` 函数处理实际的计算:

```

start = get_long();
printf("upper limit: ");
stop = get_long();
while (start != 0 || stop != 0)
{
    if (bad_limits(start, stop, MIN, MAX))
        printf("Please try again.\n");
    else
    {
        answer = sum_squares(start, stop);
        printf("The sum of the squares of the integers ");
        printf("from %ld to %ld is %g\n", start, stop, answer);
    }
    printf("Enter the limits (enter 0 for both "
           "limits to quit):\n");
    printf("lower limit: ");
    start = get_long();
    printf("upper limit: ");
    stop = get_long();
}

```

8.6.2 输入流和数字

在编写处理错误输入的代码时 (如程序清单 8.7), 应该很清楚 C 是如何处理输入的。考虑下面的输入:

is 28 12.4

在我们眼中, 这就像是一个由字符、整数和浮点数组成的字符串。但是对 C 程序而言, 这是一个字节流。第 1 个字节是字母 `i` 的字符编码, 第 2 个字节是字母 `s` 的字符编码, 第 3 个字节是空格字符的字符编码, 第 4 个字节是数字 `2` 的字符编码, 等等。所以, 如果 `get_long()` 函数处理这一行输入, 第 1 个字符

是非数字，那么整行输入都会被丢弃，包括其中的数字，因为这些数字只是该输入行中的其他字符：

```
while ((ch = getchar()) != '\n')
    putchar(ch); // 处理错误的输入
```

虽然输入流由字符组成，但是也可以设置 `scanf()` 函数把它们转换成数值。例如，考虑下面的输入：

42

如果在 `scanf()` 函数中使用 `%c` 转换说明，它只会读取字符 4 并将其储存在 `char` 类型的变量中。如果使用 `%s` 转换说明，它会读取字符 4 和字符 2 这两个字符，并将其储存在字符数组中。如果使用 `%d` 转换说明，`scanf()` 同样会读取两个字符，但是随后会计算出它们对应的整数值： $4 \times 10 + 2$ ，即 42，然后将表示该整数的二进制数储存在 `int` 类型的变量中。如果使用 `%f` 转换说明，`scanf()` 也会读取两个字符，计算出它们对应的数值 42.0，用内部的浮点表示法表示该值，并将结果储存在 `float` 类型的变量中。

简而言之，输入由字符组成，但是 `scanf()` 可以把输入转换成整数值或浮点数值。使用转换说明（如 `%d` 或 `%f`）限制了可接受输入的字符类型，而 `getchar()` 和使用 `%c` 的 `scanf()` 接受所有的字符。

8.7 菜单浏览

许多计算机程序都把菜单作为用户界面的一部分。菜单给用户提供方便的同时，却给程序员带来了一些麻烦。我们看看其中涉及了哪些问题。

菜单给用户提供了一份响应程序的选项。假设有下面一个例子：

```
Enter the letter of your choice:
a. advice          b. bell
c. count          q. quit
```

理想状态是，用户输入程序所列选项之一，然后程序根据用户所选项完成任务。作为一名程序员，自然希望这一过程能顺利进行。因此，第 1 个目标是：当用户遵循指令时程序顺利运行；第 2 个目标是：当用户没有遵循指令时，程序也能顺利运行。显而易见，要实现第 2 个目标难度较大，因为很难预料用户在使用程序时的所有错误情况。

现在的应用程序通常使用图形界面，可以点击按钮、查看对话框、触摸图标，而不是我们示例中的命令行模式。但是，两者的处理过程大致相同：给用户提供选项、检查并执行用户的响应、保护程序不受误操作的影响。除了界面不同，它们底层的程序结构也几乎相同。但是，使用图形界面更容易通过限制选项控制输入。

8.7.1 任务

我们来更具体地分析一个菜单程序需要执行哪些任务。它要获取用户的响应，根据响应选择要执行的动作。另外，程序应该提供返回菜单的选项。`C` 的 `switch` 语句是根据选项决定行为的好工具，用户的每个选择都可以对应一个特定的 `case` 标签。使用 `while` 语句可以实现重复访问菜单的功能。因此，我们写出以下伪代码：

```
获取选项
当选项不是 'q' 时
    转至相应的选项并执行
    获取下一个选项
```

8.7.2 使执行更顺利

当你决定实现这个程序时，就要开始考虑如何让程序顺利运行（顺利运行指的是，处理正确输入和错

误输入时都能顺利运行)。例如, 你能做的是让“获取选项”部分的代码筛选掉不合适的响应, 只把正确的响应传入 switch。这表明需要为输入过程提供一个只返回正确响应的函数。结合 while 循环和 switch 语句, 其程序结构如下:

```
#include <stdio.h>
char get_choice(void);
void count(void);
int main(void)
{
    int choice;

    while ((choice = get_choice()) != 'q')
    {
        switch (choice)
        {
            case 'a': printf("Buy low, sell high.\n");
                        break;
            case 'b': putchar('\a'); /* ANSI */
                        break;
            case 'c': count();
                        break;
            default:  printf("Program error!\n");
                        break;
        }
    }
    return 0;
}
```

定义 get_choice() 函数只能返回'a'、'b'、'c' 和 'q'。get_choice() 的用法和 getchar() 相同, 两个函数都是获取一个值, 并与终止值(该例中是'q')作比较。我们尽量简化实际的菜单选项, 以便读者把注意力集中在程序结构上。稍后再讨论 count() 函数。default 语句可以方便调试。如果 get_choice() 函数没能把返回值限制为菜单指定的几个选项值, default 语句有助于发现问题所在。

get_choice() 函数

下面的伪代码是设计这个函数的一种方案:

显示选项

获取用户的响应

当响应不合适时

提示用户再次输入

获取用户的响应

下面是一个简单而笨拙的实现:

```
char get_choice(void)
{
    int ch;
    printf("Enter the letter of your choice:\n");
    printf("a. advice          b. bell\n");
    printf("c. count           q. quit\n");
    ch = getchar();
    while ((ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
    }
}
```

```

    ch = getchar();
}
return ch;
}

```

缓冲输入依旧带来些麻烦，程序把用户每次按下 Return 键产生的换行符视为错误响应。为了让程序的界面更流畅，该函数应该跳过这些换行符。

这类问题有多种解决方案。一种是用名为 `get_first()` 的新函数替换 `getchar()` 函数，读取一行的第 1 个字符并丢弃剩余的字符。这种方法的优点是，把类似 `act` 这样的输入视为简单的 `a`，而不是继续把 `act` 中的 `c` 作为选项 `c` 的一个有效的响应。我们重写输入函数如下：

```

char get_choice(void)
{
    int ch;
    printf("Enter the letter of your choice:\n");
    printf("a. advice          b. bell\n");
    printf("c. count           q. quit\n");
    ch = get_first();
    while ((ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = getfirst();
    }
    return ch;
}
char get_first(void)
{
    int ch;
    ch = getchar(); /* 读取下一个字符 */
    while (getchar() != '\n')
        continue; /* 跳过该行剩下的内容 */
    return ch;
}

```

8.7.3 混合字符和数值输入

前面分析过混合字符和数值输入会产生一些问题，创建菜单也有这样的问题。例如，假设 `count()` 函数（选择 `c`）的代码如下：

```

void count(void)
{
    int n, i;
    printf("Count how far? Enter an integer:\n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
}

```

如果输入 3 作为响应，`scanf()` 会读取 3 并把换行符留在输入队列中。下次调用 `get_choice()` 将导致 `get_first()` 返回这个换行符，从而导致我们不希望出现的行为。

重写 `get_first()`，使其返回下一个非空白字符而不仅仅是下一个字符，即可修复这个问题。我们把这个任务留给读者作为练习。另一种方法是在 `count()` 函数中清理换行符，如下所示：

```

void count(void)
{
    int n, i;

```

```

printf("Count how far? Enter an integer:\n");
n = get_int();
for (i = 1; i <= n; i++)
    printf("%d\n", i);
while (getchar() != '\n')
    continue;
}

```

该函数借鉴了程序清单 8.7 中的 `get_long()` 函数，将其改为 `get_int()` 获取 `int` 类型的数据而不是 `long` 类型的数据。回忆一下，原来的 `get_long()` 函数如何检查有效输入和让用户重新输入。程序清单 8.8 演示了菜单程序的最终版本。

程序清单 8.8 menuette.c 程序

```

/* menuette.c -- 菜单程序 */
#include <stdio.h>
char get_choice(void);
char get_first(void);
int get_int(void);
void count(void);
int main(void)
{
    int choice;
    void count(void);

    while ((choice = get_choice()) != 'q')
    {
        switch (choice)
        {
            case 'a': printf("Buy low, sell high.\n");
                        break;
            case 'b': putchar('\a'); /* ANSI */
                        break;
            case 'c': count();
                        break;
            default:  printf("Program error!\n");
                        break;
        }
    }
    printf("Bye.\n");

    return 0;
}

void count(void)
{
    int n, i;

    printf("Count how far? Enter an integer:\n");
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while (getchar() != '\n')
        continue;
}

char get_choice(void)

```

```

{
    int ch;

    printf("Enter the letter of your choice:\n");
    printf("a. advice          b. bell\n");
    printf("c. count           q. quit\n");
    ch = get_first();
    while ((ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = get_first();
    }

    return ch;
}

char get_first(void)
{
    int ch;

    ch = getchar();
    while (getchar() != '\n')
        continue;

    return ch;
}

int get_int(void)
{
    int input;
    char ch;

    while (scanf("%d", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch); // 处理错误输出
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}

```

下面是该程序的一个运行示例：

```

Enter the letter of your choice:
a. advice          b. bell
c. count           q. quit
a
Buy low, sell high.
Enter the letter of your choice:
a. advice          b. bell
c. count           q. quit
count
Count how far? Enter an integer:
two

```

```

two is not an integer.
Please enter an integer value, such as 25, -178, or 3: 5

1
2
3
4
5
Enter the letter of your choice:
a. advice          b. bell
c. count           q. quit
d
Please respond with a, b, c, or q.
q

```

要写出一个自己十分满意的菜单界面不容易。但是，在开发了一种可行的方案后，可以在其他情况下复用这个菜单界面。

学完以上程序示例后，还要注意在处理较复杂的任务时，如何让函数把任务委派给另一个函数。这样让程序更模块化。

8.8 关键概念

C 程序把输入作为传入的字节流。`getchar()` 函数把每个字符解释成一个字符编码。`scanf()` 函数以同样的方式看待输入，但是根据转换说明，它可以把字符输入转换成数值。许多操作系统都提供重定向，允许用文件代替键盘输入，用文件代替显示器输出。

程序通常接受特殊形式的输入。可以在设计程序时考虑用户在输入时可能犯的错误，在输入验证部分处理这些错误情况，让程序更强健更友好。

对于一个小型程序，输入验证可能是代码中最复杂的部分。处理这类问题有多种方案。例如，如果用户输入错误类型的信息，可以终止程序，也可以给用户提供有限次或无限次机会重新输入。

8.9 本章小结

许多程序使用 `getchar()` 逐字符读取输入。通常，系统使用行缓冲输入，即当用户按下 `Enter` 键后输入才被传送给程序。按下 `Enter` 键也传送了一个换行符，编程时要注意处理这个换行符。ANSI C 把缓冲输入作为标准。

通过标准 I/O 包中的一系列函数，以统一的方式处理不同系统中的不同文件形式，是 C 语言的特性之一。`getchar()` 和 `scanf()` 函数也属于这一系列。当检测到文件结尾时，这两个函数都返回 `EOF`（被定义在 `stdio.h` 头文件中）。在不同系统中模拟文件结尾条件的方式稍有不同。在 UNIX 系统中，在一行开始处按下 `Ctrl+D` 可以模拟文件结尾条件；而在 DOS 系统中则使用 `Ctrl+Z`。

许多操作系统（包括 UNIX 和 DOS）都有重定向的特性，因此可以用文件代替键盘和屏幕进行输入和输出。读到 `EOF` 即停止读取的程序可用于键盘输入和模拟文件结尾信号，或者用于重定向文件。

混合使用 `getchar()` 和 `scanf()` 时，如果在调用 `getchar()` 之前，`scanf()` 在输入行留下一个换行符，会导致一些问题。不过，意识到这个问题就可以在程序中妥善处理。

编写程序时，要认真设计用户界面。事先预料一些用户可能会犯的错误，然后设计程序妥善处理这些错误情况。

8.10 复习题

复习题的参考答案在附录 A 中。

1. `putchar(getchar())` 是一个有效表达式，它实现什么功能？`getchar(putchar())` 是否也是有效表达式？
2. 下面的语句分别完成什么任务？
 - a. `putchar('H');`
 - b. `putchar('\007');`
 - c. `putchar('\n');`
 - d. `putchar('\b');`
3. 假设有一个名为 `count` 的可执行程序，用于统计输入的字符数。设计一个使用 `count` 程序统计 `essay` 文件中字符数的命令行，并把统计结果保存在 `essayct` 文件中。
4. 给定复习题 3 中的程序和文件，下面哪一条是有效的命令？
 - a. `essayct <essay`
 - b. `count essay`
 - c. `essay >count`
5. EOF 是什么？
6. 对于给定的输出（`ch` 是 `int` 类型，而且是缓冲输入），下面各程序段的输出分别是什么？
 - a. 输入如下：
`If you quit, I will.[enter]`
程序段如下：

```
while ((ch = getchar()) != 'i')
    putchar(ch);
```
 - b. 输入如下：
`Harhar[enter]`
程序段如下：

```
while ((ch = getchar()) != '\n')
{
    putchar(ch++);
    putchar(++ch);
```
7. C 如何处理不同计算机系统中的不同文件和换行约定？
8. 在使用缓冲输入的系统中，把数值和字符混合输入会遇到什么潜在的问题？

8.11 编程练习

下面的一些程序要求输入以 EOF 终止。如果你的操作系统很难或根本无法使用重定向，请使用一些其他的测试来终止输入，如读到`&`字符时停止。

1. 设计一个程序，统计在读到文件结尾之前读取的字符数。
2. 编写一个程序，在遇到 EOF 之前，把输入作为字符流读取。程序要打印每个输入的字符及其相应的 ASCII 十进制值。注意，在 ASCII 序列中，空格字符前面的字符都是非打印字符，要特殊处理

这些字符。如果非打印字符是换行符或制表符，则分别打印\n或\t。否则，使用控制字符表示法。例如，ASCII的1是Ctrl+A，可显示为^A。注意，A的ASCII值是Ctrl+A的值加上64。其他非打印字符也有类似的关系。除每次遇到换行符打印新的一行之外，每行打印10对值。(注意：不同的操作系统其控制字符可能不同。)

3. 编写一个程序，在遇到EOF之前，把输入作为字符流读取。该程序要报告输入中的大写字母和小写字母的个数。假设大小写字母数值是连续的。或者使用ctype.h库中合适的分类函数更方便。
4. 编写一个程序，在遇到EOF之前，把输入作为字符流读取。该程序要报告平均每个单词的字母数。不要把空白统计为单词的字母。实际上，标点符号也不应该统计，但是现在暂时不考虑这么多(如果你比较在意这点，考虑使用ctype.h系列中的ispunct()函数)。
5. 修改程序清单8.4的猜数字程序，使用更智能的猜测策略。例如，程序最初猜50，询问用户是猜大了、猜小了还是猜对了。如果猜小了，那么下一次猜测的值应是50和100中值，也就是75。如果这次猜大了，那么下一次猜测的值应是50和75的中值，等等。使用二分查找(binary search)策略，如果用户没有欺骗程序，那么程序很快就会猜到正确的答案。
6. 修改程序清单8.8中的get_first()函数，让该函数返回读取的第一个非空白字符，并在一个简单的程序中测试。
7. 修改第7章的编程练习8，用字符代替数字标记菜单的选项。用q代替5作为结束输入的标记。
8. 编写一个程序，显示一个提供加法、减法、乘法、除法的菜单。获得用户选择的选项后，程序提示用户输入两个数字，然后执行用户刚才选择的操作。该程序只接受菜单提供的选项。程序使用float类型的变量储存用户输入的数字，如果用户输入失败，则允许再次输入。进行除法运算时，如果用户输入0作为第2个数(除数)，程序应提示用户重新输入一个新值。该程序的一个运行示例如下：

```

Enter the operation of your choice:
a. add          s. subtract
m. multiply     d. divide
q. quit
a
Enter first number: 22 .4
Enter second number: one
one is not an number.
Please enter a number, such as 2.5, -1.78E8, or 3: 1
22.4 + 1 = 23.4
Enter the operation of your choice:
a. add          s. subtract
m. multiply     d. divide
q. quit
d
Enter first number: 18.4
Enter second number: 0
Enter a number other than 0: 0.2
18.4 / 0.2 = 92
Enter the operation of your choice:
a. add          s. subtract
m. multiply     d. divide
q. quit
q
Bye.

```

本章介绍以下内容：

- 关键字：return
- 运算符：*（一元）、&（一元）
- 函数及其定义方式
- 如何使用参数和返回值
- 如何把指针变量用作函数参数
- 函数类型
- ANSI C 原型
- 递归

如何组织程序？C 的设计思想是，把函数用作构件块。我们已经用过 C 标准库的函数，如 printf()、scanf()、getchar()、putchar() 和 strlen()。现在要进一步学习如何创建自己的函数。前面章节中已大致介绍了相关过程，本章将巩固以前学过的知识并做进一步的拓展。

9.1 复习函数

首先，什么是函数？函数（function）是完成特定任务的独立程序代码单元。语法规则定义了函数的结构和使用方式。虽然 C 中的函数和其他语言中的函数、子程序、过程作用相同，但是细节上略有不同。一些函数执行某些动作，如 printf() 把数据打印到屏幕上；一些函数找出一个值供程序使用，如 strlen() 把指定字符串的长度返回给程序。一般而言，函数可以同时具备以上两种功能。

为什么要使用函数？首先，使用函数可以省去编写重复代码的苦差。如果程序要多次完成某项任务，那么只需编写一个合适的函数，就可以在需要时使用这个函数，或者在不同的程序中使用该函数，就像许多程序中使用 putchar() 一样。其次，即使程序只完成某项任务一次，也值得使用函数。因为函数让程序更加模块化，从而提高了程序代码的可读性，更方便后期修改、完善。例如，假设要编写一个程序完成以下任务：

- 读入一系列数字；
- 分类这些数字；
- 找出这些数字的平均值；
- 打印一份柱状图。

可以使用下面的程序：

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
```

```

float list[SIZE];

readlist(list, SIZE);
sort(list, SIZE);
average(list, SIZE);
bargraph(list, SIZE);
return 0;
}

```

当然，还要编写 4 个函数 `readlist()`、`sort()`、`average()` 和 `bargraph()` 的实现细节。描述性的函数名能清楚地表达函数的用途和组织结构。然后，单独设计和测试每个函数，直到函数都能正常完成任务。如果这些函数够通用，还可以用于其他程序。

许多程序员喜欢把函数看作是根据传入信息（输入）及其生成的值或响应的动作（输出）来定义的“黑盒”。如果不是自己编写函数，根本不用关心黑盒的内部行为。例如，使用 `printf()` 时，只需知道给该函数传入格式字符串或一些参数以及 `printf()` 生成的输出，无需了解 `printf()` 的内部代码。以这种方式看待函数有助于把注意力集中在程序的整体设计，而不是函数的实现细节上。因此，在动手编写代码之前，仔细考虑一下函数应该完成什么任务，以及函数和程序整体的关系。

如何了解函数？首先要知道如何正确地定义函数、如何调用函数和如何建立函数间的通信。我们从一个简单的程序示例开始，帮助读者理清这些内容，然后再详细讲解。

9.1.1 创建并使用简单函数

我们的第 1 个目标是创建一个在一行打印 40 个星号的函数，并在一个打印表头的程序中使用该函数。如程序清单 9.1 所示，该程序由 `main()` 和 `starbar()` 组成。

程序清单 9.1 `lethead1.c` 程序

```

/* lethead1.c */
#include <stdio.h>
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40

void starbar(void); /* 函数原型 */

int main(void)
{
    starbar();
    printf("%s\n", NAME);
    printf("%s\n", ADDRESS);
    printf("%s\n", PLACE);
    starbar(); /* 使用函数 */

    return 0;
}

void starbar(void) /* 定义函数 */
{
    int count;

    for (count = 1; count <= WIDTH; count++)
        putchar('*');
}

```

```
    putchar ('\n');
}
```

该程序的输出如下：

```
*****
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
*****
```

9.1.2 分析程序

该程序要注意以下几点。

- 程序在 3 处使用了 starbar 标识符：函数原型（*function prototype*）告诉编译器函数 starbar() 的类型；函数调用（*function call*）表明在此处执行函数；函数定义（*function definition*）明确地指定了函数要做什么。
- 函数和变量一样，有多种类型。任何程序在使用函数之前都要声明该函数的类型。因此，在 main() 函数定义的前面出现了下面的 ANSI C 风格的函数原型：

```
void starbar(void);
```

圆括号表明 starbar 是一个函数名。第 1 个 void 是函数类型，void 类型表明函数没有返回值。第 2 个 void（在圆括号中）表明该函数不带参数。分号表明这是在声明函数，不是定义函数。也就是说，这行声明了程序将使用一个名为 starbar()、没有返回值、没有参数的函数，并告诉编译器在别处查找该函数的定义。对于不识别 ANSI C 风格原型的编译器，只需声明函数的类型，如下所示：

```
void starbar();
```

注意，一些老版本的编译器甚至连 void 都识别不了。如果使用这种编译器，就要把没有返回值的函数声明为 int 类型。当然，最好还是换一个新的编译器。

- 一般而言，函数原型指明了函数的返回值类型和函数接受的参数类型。这些信息称为该函数的签名（*signature*）。对于 starbar() 函数而言，其签名是该函数没有返回值，没有参数。
- 程序把 starbar() 原型置于 main() 的前面。当然，也可以放在 main() 里面的声明变量处。放在哪个位置都可以。
- 在 main() 中，执行到下面的语句时调用了 starbar() 函数：

```
starbar();
```

这是调用 void 类型函数的一种形式。当计算机执行到 starbar(); 语句时，会找到该函数的定义并执行其中的内容。执行完 starbar() 中的代码后，计算机返回主调函数（*calling function*）继续执行下一行（本例中，主调函数是 main()），见图 9.1（更确切地说，编译器把 C 程序翻译成执行以上操作的机器语言代码）。

- 程序中 starbar() 和 main() 的定义形式相同。首先函数头包括函数类型、函数名和圆括号，接着是左花括号、变量声明、函数表达式语句，最后以右花括号结束（见图 9.2）。注意，函数头中的 starbar() 后面没有分号，告诉编译器这是定义 starbar()，而不是调用函数或声明函数原型。
- 程序把 starbar() 和 main() 放在一个文件中。当然，也可以把它们分别放在两个文件中。把函数都放在一个文件中的单文件形式比较容易编译，而使用多个文件方便在不同的程序中使用同一个

函数。如果把函数放在一个单独的文件中，要把#define 和#include 指令也放入该文件。我们稍后会讨论使用多个文件的情况。现在，先把所有的函数都放在一个文件中。main() 的右花括号告诉编译器该函数结束的位置，后面的 starbar() 函数头告诉编译器 starbar() 是一个函数。

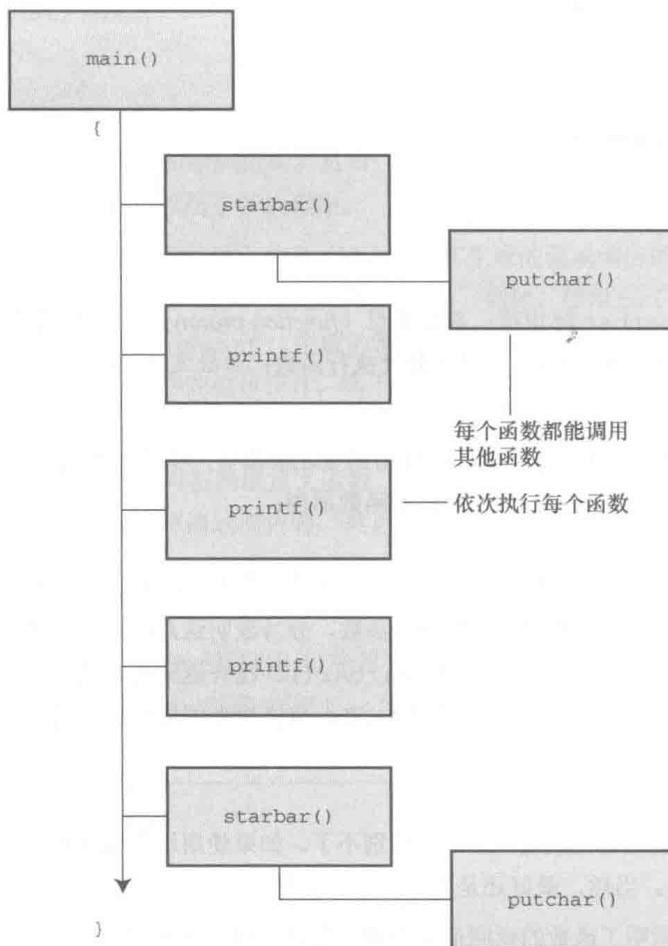


图 9.1 lethead1.c (程序清单 9.1) 的程序流

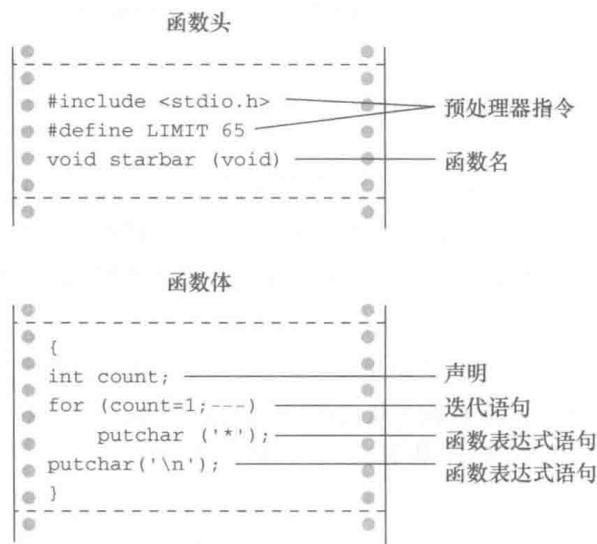


图 9.2 简单函数的结构

- starbar() 函数中的变量 count 是局部变量 (*local variable*)，意思是该变量只属于 starbar() 函数。可以在程序中的其他地方（包括 main() 中）使用 count，这不会引起名称冲突，它们是同名的不同变量。

如果把 starbar() 看作是一个黑盒，那么它的行为是打印一行星号。不用给该函数提供任何输入，因为调用它不需要其他信息。而且，它没有返回值，所以也不给 main() 提供（或返回）任何信息。简而言之，starbar() 不需要与主调函数通信。

接下来介绍一个函数间需要通信的例子。

9.1.3 函数参数

在程序清单 9.1 的输出中，如果文字能居中，信头会更加美观。可以通过在打印文字之前打印一定数量的空格来实现，这和打印一定数量的星号 (starbar() 函数) 类似，只不过现在要打印的是一定数量的空格。虽然这是两个任务，但是任务非常相似，与其分别为它们编写一个函数，不如写一个更通用的函数，可以在两种情况下使用。我们设计一个新的函数 show_n_char()（显示一个字符 n 次）。唯一要改变的是使用内置的值来显示字符和重复的次数，show_n_char() 将使用函数参数来传递这些值。

我们来具体分析。假设可用的空间是 40 个字符宽。调用 show_n_char('*', 40) 应该正好打印一行 40 个星号，就像 starbar() 之前做的那样。第 2 行 GIGATHINK, INC. 的空格怎么处理？GIGATHINK, INC. 是 15 个字符宽，所以第 1 个版本中，文字后面有 25 个空格。为了让文字居中，文字的左侧应该有 12 个空格，右侧有 13 个空格。因此，可以调用 show_n_char('*', 12)。

show_n_char() 与 starbar() 很相似，但是 show_n_char() 带有参数。从功能上看，前者不会添加换行符，而后者会，因为 show_n_char() 要把空格和文本打印成一行。程序清单 9.2 是修改后的版本。为强调参数的工作原理，程序使用了不同的参数形式。

程序清单 9.2 letthead2.c 程序

```
/* letthead2.c */
#include <stdio.h>
#include <string.h>           /* 为 strlen() 提供原型 */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '

void show_n_char(char ch, int num);

int main(void)
{
    int spaces;

    show_n_char('*', WIDTH);          /* 用符号常量作为参数 */
    putchar('\n');
    show_n_char(SPACES, 12);         /* 用符号常量作为参数 */
    printf("%s\n", NAME);
    spaces = (WIDTH - strlen(ADDRESS)) / 2; /* 计算要跳过多少个空格 */

    show_n_char(SPACES, spaces);      /* 用一个变量作为参数 */
    printf("%s\n", ADDRESS);
```

```

show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);

printf("%s\n", PLACE); /* 用一个表达式作为参数 */
show_n_char('*', WIDTH);
putchar('\n');

return 0;
}

/* show_n_char() 函数的定义 */
void show_n_char(char ch, int num)
{
    int count;

    for (count = 1; count <= num; count++)
        putchar(ch);
}

```

该函数的运行结果如下：

```

*****
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
*****

```

下面我们回顾一下如何编写一个带参数的函数，然后介绍这种函数的用法。

9.1.4 定义带形式参数的函数

函数定义从下面的 ANSI C 风格的函数头开始：

```
void show_n_char(char ch, int num)
```

该行告知编译器 `show_n_char()` 使用两个参数 `ch` 和 `num`，`ch` 是 `char` 类型，`num` 是 `int` 类型。这两个变量被称为形式参数 (*formal argument*，但是最近的标准推荐使用 *formal parameter*)，简称形参。和定义在函数中变量一样，形式参数也是局部变量，属该函数私有。这意味着在其他函数中使用同名变量不会引起名称冲突。每次调用函数，就会给这些变量赋值。

注意，ANSI C 要求在每个变量前都声明其类型。也就是说，不能像普通变量声明那样使用同一类型的变量列表：

```
void dibs(int x, y, z) /* 无效的函数头 */
void dubs(int x, int y, int z) /* 有效的函数头 */
```

ANSI C 也接受 ANSI C 之前的形式，但是将其视为废弃不用的形式：

```
void show_n_char(ch, num)
char ch;
int num;
```

这里，圆括号中只有参数名列表，而参数的类型在后面声明。注意，普通的局部变量在左花括号之后声明，而上面的变量在函数左花括号之前声明。如果变量是同一类型，这种形式可以用逗号分隔变量名列表，如下所示：

```
void dibs(x, y, z)
int x, y, z; /* 有效 */
```

当前的标准正逐渐淘汰 ANSI 之前的形式。读者应对此有所了解，以便能看懂以前编写的程序，但是

自己编写程序时应使用现在的标准形式（C99 和 C11 标准继续警告这些过时的用法即将被淘汰）。

虽然 `show_n_char()` 接受来自 `main()` 的值，但是它没有返回值。因此，`show_n_char()` 的类型是 `void`。

下面，我们来学习如何使用函数。

9.1.5 声明带形式参数函数的原型

在使用函数之前，要用 ANSI C 形式声明函数原型：

```
void show_n_char(char ch, int num);
```

当函数接受参数时，函数原型用逗号分隔的列表指明参数的数量和类型。根据个人喜好，你也可以省略变量名：

```
void show_n_char(char, int);
```

在原型中使用变量名并没有实际创建变量，`char` 仅代表了一个 `char` 类型的变量，以此类推。

再次提醒读者注意，ANSI C 也接受过去的声明函数形式，即圆括号内没有参数列表：

```
void show_n_char();
```

这种形式最终会从标准中剔除。即使没有被剔除，现在函数原型的设计也更有优势（稍后会介绍）。了解这种形式的写法是为了以后读得懂以前写的代码。

9.1.6 调用带实际参数的函数

在函数调用中，实际参数 (*actual argument*, 简称实参) 提供了 `ch` 和 `num` 的值。考虑程序清单 9.2 中第 1 次调用 `show_n_char()`：

```
show_n_char(SPACE, 12);
```

实际参数是空格字符和 12。这两个值被赋给 `show_n_char()` 中相应的形式参数：变量 `ch` 和 `num`。简而言之，形式参数是被调函数 (*called function*) 中的变量，实际参数是主调函数 (*calling function*) 赋给被调函数的具体值。如上例所示，实际参数可以是常量、变量，甚至是更复杂的表达式。无论实际参数是何种形式都要被求值，然后该值被拷贝给被调函数相应的形式参数。以程序清单 9.2 中最后一次调用 `show_n_char()` 为例：

```
show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
```

构成该函数第 2 个实际参数的是一个很长的表达式，对该表达式求值为 10。然后，10 被赋给变量 `num`。被调函数不知道也不关心传入的数值是来自常量、变量还是一般表达式。再次强调，实际参数是具体的值，该值要被赋给作为形式参数的变量（见图 9.3）。因为被调函数使用的值是从主调函数中拷贝而来，所以无论被调函数对拷贝数据进行什么操作，都不会影响主调函数中的原始数据。

注意 实际参数和形式参数

实际参数是出现在函数调用圆括号中的表达式。形式参数是函数定义的函数头中声明的变量。调用函数时，创建了声明为形式参数的变量并初始化为实际参数的求值结果。程序清单 9.2 中，`'*'` 和 `WIDTH` 都是第 1 次调用 `show_n_char()` 时的实际参数，而 `SPACE` 和 11 是第 2 次调用 `show_n_char()` 时的实际参数。在函数定义中，`ch` 和 `num` 都是该函数的形式参数。



图 9.3 形式参数和实际参数

9.1.7 黑盒视角

从黑盒的视角看 `show_n_char()`，待显示的字符和显示的次数是输入。执行后的结果是打印指定数量的字符。输入以参数的形式被传递给函数。这些信息清楚地表明了如何在 `main()` 中使用该函数。而且，这也可作为编写该函数的设计说明。

黑盒方法的核心部分是：`ch`、`num` 和 `count` 都是 `show_n_char()` 私有的局部变量。如果在 `main()` 中使用同名变量，那么它们相互独立，互不影响。也就是说，如果 `main()` 有一个 `count` 变量，那么改变它的值不会改变 `show_n_char()` 中的 `count`，反之亦然。黑盒里发生了什么对主调函数是不可见的。

9.1.8 使用 return 从函数中返回值

前面介绍了如何把信息从主调函数传递给被调函数。反过来，函数的返回值可以把信息从被调函数传回主调函数。为进一步说明，我们将创建一个返回两个参数中较小值的函数。由于函数被设计用来处理 `int` 类型的值，所以被命名为 `imin()`。另外，还要创建一个简单的 `main()`，用于检查 `imin()` 是否正常工作。这种被设计用于测试函数的程序有时被称为驱动程序 (*driver*)，该驱动程序调用一个函数。如果函数成功通过了测试，就可以安装在一个更重要的程序中使用。程序清单 9.3 演示了这个驱动程序和返回最小值的函数。

程序清单 9.3 lesser.c 程序

```

/* lesser.c -- 找出两个整数中较小的一个 */
#include <stdio.h>
int imin(int, int);

int main(void)
{
    int evill, evil2;

    printf("Enter a pair of integers (q to quit):\n");
  
```

```

while (scanf("%d %d", &evil1, &evil2) == 2)
{
    printf("The lesser of %d and %d is %d.\n",
           evil1, evil2, imin(evil1, evil2));
    printf("Enter a pair of integers (q to quit):\n");
}
printf("Bye.\n");

return 0;
}

int imin(int n, int m)
{
    int min;

    if (n < m)
        min = n;
    else
        min = m;

    return min;
}

```

回忆一下，`scanf()`返回成功读数据的个数，所以如果输入不是两个整数会导致循环终止。下面是一个运行示例：

```

Enter a pair of integers (q to quit):
509 333
The lesser of 509 and 333 is 333.
Enter a pair of integers (q to quit):
-9393 6
The lesser of -9393 and 6 is -9393.
Enter a pair of integers (q to quit):
q
Bye.

```

关键字 `return` 后面的表达式的值就是函数的返回值。在该例中，该函数返回的值就是变量 `min` 的值。因为 `min` 是 `int` 类型的变量，所以 `imin()` 函数的类型也是 `int`。

变量 `min` 属于 `imin()` 函数私有，但是 `return` 语句把 `min` 的值传回了主调函数。下面这条语句的作用是把 `min` 的值赋给 `lesser`：

```
lesser = imin(n, m);
```

是否能像写成下面这样：

```
imin(n, m);
lesser = min;
```

不能。因为主调函数甚至不知道 `min` 的存在。记住，`imin()` 中的变量是 `imin()` 的局部变量。函数调用 `imin(evil1, evil2)` 只是把两个变量的值拷贝了一份。

返回值不仅可以赋给变量，也可以被用作表达式的一部分。例如，可以这样：

```
answer = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + answer, LIMIT));
```

返回值不一定是变量的值，也可以是任意表达式的值。例如，可以用以下的代码简化程序示例：

```
/* 返回最小值的函数，第 2 个版本 */
imin(int n, int m)
```

```
{
    return (n < m) ? n : m;
}
```

条件表达式的值是 n 和 m 中的较小者，该值要被返回给主调函数。虽然这里不要求用圆括号把返回值括起来，但是如果想让程序条理更清楚或统一风格，可以把返回值放在圆括号内。

如果函数返回值的类型与函数声明的类型不匹配会怎样？

```
int what_if(int n)
{
    double z = 100.0 / (double) n;
    return z; // 会发生什么?
}
```

实际得到的返回值相当于把函数中指定的返回值赋给与函数类型相同的变量所得到的值。因此在本例中，相当于把 z 的值赋给 int 类型的变量，然后返回 int 类型变量的值。例如，假设有下面的函数调用：

```
result = what_if(64);
```

虽然在 what_if() 函数中赋给 z 的值是 1.5625，但是 return 语句返回确实 int 类型的值 1。

使用 return 语句的另一个作用是，终止函数并把控制返回给主调函数的下一条语句。因此，可以这样编写 imin()：

```
/* 返回最小值的函数，第 3 个版本 */
imin(int n, int m)
{
    if (n < m)
        return n;
    else
        return m;
}
```

许多 C 程序员都认为只在函数末尾使用一次 return 语句比较好，因为这样做更方便浏览程序的人理解函数的控制流。但是，在函数中使用多个 return 语句也没有错。无论如何，对用户而言，这 3 个版本的函数用起来都一样，因为所有的输入和输出都完全相同，不同的是函数内部的实现细节。下面的版本也没问题：

```
/* 返回最小值的函数，第 4 个版本 */
imin(int n, int m)
{
    if (n < m)
        return n;
    else
        return m;
    printf("Professor Fleppard is like totally a fopdoodle.\n");
}
```

return 语句导致 printf() 语句永远不会被执行。如果 Fleppard 教授在自己的程序中使用这个版本的函数，可能永远不知道编写这个函数的学生对他的看法。

另外，还可以这样使用 return：

```
return;
```

这条语句会导致终止函数，并把控制返回给主调函数。因为 return 后面没有任何表达式，所以没有返回值，只有在 void 函数中才会用到这种形式。

9.1.9 函数类型

声明函数时必须声明函数的类型。带返回值的函数类型应该与其返回值类型相同，而没有返回值的函数应

声明为 `void` 类型。如果没有声明函数的类型，旧版本的 C 编译器会假定函数的类型是 `int`。这一惯例源于 C 的早期，那时的函数绝大多数都是 `int` 类型。然而，C99 标准不再支持 `int` 类型函数的这种假定设置。

类型声明是函数定义的一部分。要记住，函数类型指的是返回值的类型，不是函数参数的类型。例如，下面的函数头定义了一个带两个 `int` 类型参数的函数，但是其返回值是 `double` 类型。

```
double klink(int a, int b)
```

要正确地使用函数，程序在第 1 次使用函数之前必须知道函数的类型。方法之一是，把完整的函数定义放在第 1 次调用函数的前面。然而，这种方法增加了程序的阅读难度。而且，要使用的函数可能在 C 库或其他文件中。因此，通常的做法是提前声明函数，把函数的信息告知编译器。例如，程序清单 9.3 中的 `main()` 函数包含以下几行代码：

```
#include <stdio.h>
int imin(int, int);
int main(void)
{
    int evill, evil2, lesser;
```

第 2 行代码说明 `imin` 是一个函数名，有两个 `int` 类型的形参，且返回 `int` 类型的值。现在，编译器在程序中调用 `imin()` 函数时就知道应该如何处理。

在程序清单 9.3 中，我们把函数的前置声明放在主调函数外面。当然，也可以放在主调函数里面。例如，重写 `lesser.c`（程序清单 9.3）的开头部分：

```
#include <stdio.h>
int main(void)
{
    int imin(int, int); /* 声明 imin() 函数的原型 */
    int evill, evil2, lesser;
```

注意在这两种情况中，函数原型都声明在使用函数之前。

ANSI C 标准库中，函数被分成多个系列，每一系列都有各自的头文件。这些头文件中除了其他内容，还包含了本系列所有函数的声明。例如，`stdio.h` 头文件包含了标准 I/O 库函数（如，`printf()` 和 `scanf()`）的声明。`math.h` 头文件包含了各种数学函数的声明。例如，下面的声明：

```
double sqrt(double);
```

告知编译器 `sqrt()` 函数有一个 `double` 类型的形参，而且返回 `double` 类型的值。不要混淆函数的声明和定义。函数声明告知编译器函数的类型，而函数定义则提供实际的代码。在程序中包含 `math.h` 头文件告知编译器：`sqrt()` 返回 `double` 类型，但是 `sqrt()` 函数的代码在另一个库函数的文件中。

9.2 ANSI C 函数原型

在 ANSI C 标准之前，声明函数的方案有缺陷，因为只需要声明函数的类型，不用声明任何参数。下面我们看一下使用旧式的函数声明会导致什么问题。

下面是 ANSI 之前的函数声明，告知编译器 `imin()` 返回 `int` 类型的值：

```
int imin();
```

然而，以上函数声明并未给出 `imin()` 函数的参数个数和类型。因此，如果调用 `imin()` 时使用的参数个数不对或类型不匹配，编译器根本不会察觉出来。

9.2.1 问题所在

我们看看与 `imax()` 函数相关的一些示例，该函数与 `imin()` 函数关系密切。程序清单 9.4 演示了一个

程序，用过去声明函数的方式声明了 `imax()` 函数，然后错误地使用该函数。

程序清单 9.4 misuse.c 程序

```
/* misuse.c -- 错误地使用函数 */
#include <stdio.h>
int imax(); /* 旧式函数声明 */

int main(void)
{
    printf("The maximum of %d and %d is %d.\n", 3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n", 3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(n, m)
int n, m;
{
    return (n > m ? n : m);
}
```

第 1 次调用 `printf()` 时省略了 `imax()` 的一个参数，第 2 次调用 `printf()` 时用两个浮点参数而不是整数参数。尽管有些问题，但程序可以编译和运行。

下面是使用 Xcode 4.6 运行的输出示例：

```
The maximum of 3 and 5 is 1606416656.
The maximum of 3 and 5 is 3886.
```

使用 `gcc` 运行该程序，输出的值是 1359379472 和 1359377160。这两个编译器都运行正常，之所以输出错误的结果，是因为它们运行的程序没有使用函数原型。

到底是哪里出了问题？由于不同系统的内部机制不同，所以出现问题的具体情况也不同。下面介绍的是使用 PC 和 VAX 的情况。主调函数把它的参数储存在被称为栈（*stack*）的临时存储区，被调函数从栈中读取这些参数。对于该例，这两个过程并未相互协调。主调函数根据函数调用中的实际参数决定传递的类型，而被调函数根据它的形式参数读取值。因此，函数调用 `imax(3)` 把一个整数放在栈中。当 `imax()` 函数开始执行时，它从栈中读取两个整数。而实际上栈中只存放了一个待读取的整数，所以读取的第 2 个值是当时恰好在栈中的其他值。

第 2 次使用 `imax()` 函数时，它传递的是 `float` 类型的值。这次把两个 `double` 类型的值放在栈中（回忆一下，当 `float` 类型被作为参数传递时会被升级为 `double` 类型）。在我们的系统中，两个 `double` 类型的值就是两个 64 位的值，所以 128 位的数据被放在栈中。当 `imax()` 从栈中读取两个 `int` 类型的值时，它从栈中读取前 64 位。在我们的系统中，每个 `int` 类型的变量占用 32 位。这些数据对应两个整数，其中较大的是 3886。

9.2.2 ANSI 的解决方案

针对参数不匹配的问题，ANSI C 标准要求在函数声明时还要声明变量的类型，即使用函数原型(*function prototype*) 来声明函数的返回类型、参数的数量和每个参数的类型。未标明 `imax()` 函数有两个 `int` 类型的参数，可以使用下面两种函数原型来声明：

```
int imax(int, int);
int imax(int a, int b);
```

第 1 种形式使用以逗号分隔的类型列表，第 2 种形式在类型后面添加了变量名。注意，这里的变量名

是假名，不必与函数定义的形式参数名一致。

有了这些信息，编译器可以检查函数调用是否与函数原型匹配。参数的数量是否正确？参数的类型是否匹配？以 `imax()` 为例，如果两个参数都是数字，但是类型不匹配，编译器会把实际参数的类型转换成形式参数的类型。例如，`imax(3.0, 5.0)` 会被转换成 `imax(3, 5)`。我们用函数原型替换程序清单 9.4 中的函数声明，如程序清单 9.5 所示。

程序清单 9.5 `proto.c` 程序

```
/* proto.c -- 使用函数原型 */
#include <stdio.h>
int imax(int, int);           /* 函数原型 */
int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
           3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
           3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(int n, int m)
{
    return (n > m ? n : m);
}
```

编译程序清单 9.5 时，我们的编译器给出调用的 `imax()` 函数参数太少的错误消息。

如果是类型不匹配会怎样？为探索这个问题，我们用 `imax(3, 5)` 替换 `imax(3)`，然后再次编译该程序。这次编译器没有给出任何错误信息，程序的输出如下：

```
The maximum of 3 and 5 is 5.
The maximum of 3 and 5 is 5.
```

如上文所述，第 2 次调用中的 `3.0` 和 `5.0` 被转换成 `3` 和 `5`，以便函数能正确地处理输入。

虽然没有错误消息，但是我们的编译器还是给出了警告：`double` 转换成 `int` 可能会导致丢失数据。例如，下面的函数调用：

```
imax(3.9, 5.4)
```

相当于：

```
imax(3, 5)
```

错误和警告的区别是：错误导致无法编译，而警告仍然允许编译。一些编译器在进行类似的类型转换时不会通知用户，因为 C 标准中对此未作要求。不过，许多编译器都允许用户选择警告级别来控制编译器在描述警告时的详细程度。

9.2.3 无参数和未指定参数

假设有下面的函数原型：

```
void print_name();
```

一个支持 ANSI C 的编译器会假定用户没有用函数原型来声明函数，它将不会检查参数。为了表明函数确实没有参数，应该在圆括号中使用 `void` 关键字：

```
void print_name(void);
```

支持 ANSI C 的编译器解释为 `print_name()` 不接受任何参数。然后在调用该函数时，编译器会检查以确保没有使用参数。

一些函数接受（如，`printf()` 和 `scanf()`）许多参数。例如对于 `printf()`，第 1 个参数是字符串，但是其余参数的类型和数量都不固定。对于这种情况，ANSI C 允许使用部分原型。例如，对于 `printf()` 可以使用下面的原型：

```
int printf(const char *, ...);
```

这种原型表明，第 1 个参数是一个字符串（第 11 章中将详细介绍），可能还有其他未指定的参数。

C 库通过 `stdarg.h` 头文件提供了一个定义这类（形参数量不固定的）函数的标准方法。第 16 章中详细介绍相关内容。

9.2.4 函数原型的优点

函数原型是 C 语言的一个强有力的工具，它让编译器捕获在使用函数时可能出现的许多错误或疏漏。如果编译器没有发现这些问题，就很难觉察出来。是否必须使用函数原型？不一定。你也可以使用旧式的函数声明（即不用声明任何形参），但是这样做的弊大于利。

有一种方法可以省略函数原型却保留函数原型的优点。首先要明白，之所以使用函数原型，是为了让编译器在第 1 次执行到该函数之前就知道如何使用它。因此，把整个函数定义放在第 1 次调用该函数之前，也有相同的效果。此时，函数定义也相当于函数原型。对于较小的函数，这种用法很普遍：

```
// 下面这行代码既是函数定义，也是函数原型
int imax(int a, int b) { return a > b ? a : b; }
int main()
{
    int x, z;
    ...
    z = imax(x, 50);
}
```

9.3 递归

C 允许函数调用它自己，这种调用过程称为递归（recursion）。递归有时难以捉摸，有时却很方便实用。结束递归是使用递归的难点，因为如果递归代码中没有终止递归的条件测试部分，一个调用自己的函数会无限递归。

可以使用循环的地方通常都可以使用递归。有时用循环解决问题比较好，但有时用递归更好。递归方案更简洁，但效率却没有循环高。

9.3.1 演示递归

我们通过一个程序示例，来学习什么是递归。程序清单 9.6 中的 `main()` 函数调用 `up_and_down()` 函数，这次调用称为“第 1 级递归”。然后 `up_and_down()` 调用自己，这次调用称为“第 2 级递归”。接着第 2 级递归调用第 3 级递归，以此类推。该程序示例共有 4 级递归。为了进一步深入研究递归时发生了什么，程序不仅显示了变量 `n` 的值，还显示了储存 `n` 的内存地址 `&n`。（本章稍后会详细讨论 `&` 运算符，`printf()` 函数使用 `%p` 转换说明打印地址，如果你的系统不支持这种格式，请使用 `%u` 或 `%lu` 替代 `%p`）。

程序清单 9.6 recur.c 程序

```
/* recur.c -- 递归演示 */
#include <stdio.h>
void up_and_down(int);

int main(void)
{
    up_and_down(1);
    return 0;
}

void up_and_down(int n)
{
    printf("Level %d: n location %p\n", n, &n); // #1
    if (n < 4)
        up_and_down(n + 1);
    printf("LEVEL %d: n location %p\n", n, &n); // #2
}
```

下面是在我们系统中的输出：

```
Level 1: n location 0x0012ff48
Level 2: n location 0x0012ff3c
Level 3: n location 0x0012ff30
Level 4: n location 0x0012ff24
LEVEL 4: n location 0x0012ff24
LEVEL 3: n location 0x0012ff30
LEVEL 2: n location 0x0012ff3c
LEVEL 1: n location 0x0012ff48
```

我们来仔细分析程序中的递归是如何工作的。首先，`main()` 调用了带参数 1 的 `up_and_down()` 函数，执行结果是 `up_and_down()` 中的形式参数 `n` 的值是 1，所以打印语句#1 打印 Level 1。然后，由于 `n` 小于 4，`up_and_down()`（第 1 级）调用实际参数为 `n + 1`（或 2）的 `up_and_down()`（第 2 级）。于是第 2 级调用中的 `n` 的值是 2，打印语句#1 打印 Level 2。与此类似，下面两次调用打印的分别是 Level 3 和 Level 4。

当执行到第 4 级时，`n` 的值是 4，所以 `if` 测试条件为假。`up_and_down()` 函数不再调用自己。第 4 级调用接着执行打印语句#2，即打印 LEVEL 4，因为 `n` 的值是 4。此时，第 4 级调用结束，控制被传回它的主调函数（即第 3 级调用）。在第 3 级调用中，执行的最后一条语句是调用 `if` 语句中的第 4 级调用。被调函数（第 4 级调用）把控制返回在这个位置，因此，第 3 级调用继续执行后面的代码，打印语句#2 打印 LEVEL 3。然后第 3 级调用结束，控制被传回第 2 级调用，接着打印 LEVEL 2，以此类推。

注意，每级递归的变量 `n` 都属于本级递归私有。这从程序输出的地址值可以看出（当然，不同的系统表示的地址格式不同，这里关键要注意，Level 1 和 LEVEL 1 的地址相同，Level 2 和 LEVEL 2 的地址相同，等等）。

如果觉得不好理解，可以假设有一条函数调用链——`fun1()` 调用 `fun2()`、`fun2()` 调用 `fun3()`、`fun3()` 调用 `fun4()`。当 `fun4()` 结束时，控制传回 `fun3()`；当 `fun3()` 结束时，控制传回 `fun2()`；当 `fun2()` 结束时，控制传回 `fun1()`。递归的情况与此类似，只不过 `fun1()`、`fun2()`、`fun3()` 和 `fun4()` 都是相同的函数。

9.3.2 递归的基本原理

初次接触递归会觉得较难理解。为了帮助读者理解递归过程，下面以程序清单 9.6 为例讲解几个要点。

第 1，每级函数调用都有自己的变量。也就是说，第 1 级的 n 和第 2 级的 n 不同，所以程序创建了 4 个单独的变量，每个变量名都是 n，但是它们的值各不相同。当程序最终返回 up_and_down() 的第 1 级调用时，最初的 n 仍然是它的初值 1（见图 9.4）。

变量	n	n	n	n
第1级调用后	1			
第2级调用后	1	2		
第3级调用后	1	2	3	
第4级调用后	1	2	3	4
从第4级调用返回后	1	2	3	
从第3级调用返回后	1	2		
从第2级调用返回后	1			
从第1级调用返回后				(全部结束)

图 9.4 递归中的变量

第 2，每次函数调用都会返回一次。当函数执行完毕后，控制权将被传回上一级递归。程序必须按顺序逐级返回递归，从某级 up_and_down() 返回上一级的 up_and_down()，不能跳级回到 main() 中的第 1 级调用。

第 3，递归函数中位于递归调用之前的语句，均按被调函数的顺序执行。例如，程序清单 9.6 中的打印语句 #1 位于递归调用之前，它按照递归的顺序：第 1 级、第 2 级、第 3 级和第 4 级，被执行了 4 次。

第 4，递归函数中位于递归调用之后的语句，均按被调函数相反的顺序执行。例如，打印语句 #2 位于递归调用之后，其执行的顺序是第 4 级、第 3 级、第 2 级、第 1 级。递归调用的这种特性在解决涉及相反顺序的编程问题时很有用。稍后将介绍一个这样的例子。

第 5，虽然每级递归都有自己的变量，但是并没有拷贝函数的代码。程序按顺序执行函数中的代码，而递归调用就相当于又从头开始执行函数的代码。除了为每次递归调用创建变量外，递归调用非常类似于一个循环语句。实际上，递归有时可用循环来代替，循环有时也能用递归来代替。

最后，递归函数必须包含能让递归调用停止的语句。通常，递归函数都使用 if 或其他等价的测试条件在函数形参等于某特定值时终止递归。为此，每次递归调用的形参都要使用不同的值。例如，程序清单 9.6 中的 up_and_down(n) 调用 up_and_down(n+1)。最终，实际参数等于 4 时，if 的测试条件 (n < 4) 为假。

9.3.3 尾递归

最简单的递归形式是把递归调用置于函数的末尾，即正好在 return 语句之前。这种形式的递归被称为尾递归 (*tail recursion*)，因为递归调用在函数的末尾。尾递归是最简单的递归形式，因为它相当于循环。

下面要介绍的程序示例中，分别用循环和尾递归计算阶乘。一个正整数的阶乘 (*factorial*) 是从 1 到该整数的所有整数的乘积。例如，3 的阶乘（写作 3!）是 $1 \times 2 \times 3$ 。另外，0! 等于 1，负数没有阶乘。程序清单 9.7 中，第 1 个函数使用 for 循环计算阶乘，第 2 个函数使用递归计算阶乘。

程序清单 9.7 factor.c 程序

```

// factor.c -- 使用循环和递归计算阶乘
#include <stdio.h>
long fact(int n);
long rfact(int n);
int main(void)
{
    int num;

    printf("This program calculates factorials.\n");
    printf("Enter a value in the range 0-12 (q to quit):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("No negative numbers, please.\n");
        else if (num > 12)
            printf("Keep input under 13.\n");
        else
        {
            printf("loop: %d factorial = %ld\n",
                   num, fact(num));
            printf("recursion: %d factorial = %ld\n",
                   num, rfact(num));
        }
        printf("Enter a value in the range 0-12 (q to quit):\n");
    }
    printf("Bye.\n");

    return 0;
}

long fact(int n)      // 使用循环的函数
{
    long ans;

    for (ans = 1; n > 1; n--)
        ans *= n;

    return ans;
}

long rfact(int n)     // 使用递归的函数
{
    long ans;

    if (n > 0)
        ans = n * rfact(n - 1);
    else
        ans = 1;

    return ans;
}

```

测试驱动程序把输入限制在 0~12。因为 $12!$ 已快接近 5 亿，而 $13!$ 比 62 亿还大，已超过我们系统中

`long` 类型能表示的范围。要计算超过 12 的阶乘，必须使用能表示更大范围的类型，如 `double` 或 `long long`。

下面是该程序的运行示例：

```
This program calculates factorials.
Enter a value in the range 0-12 (q to quit):
5
loop: 5 factorial = 120
recursion: 5 factorial = 120
Enter a value in the range 0-12 (q to quit):
10
loop: 10 factorial = 3628800
recursion: 10 factorial = 3628800
Enter a value in the range 0-12 (q to quit):
q
Bye.
```

使用循环的函数把 `ans` 初始化为 1，然后把 `ans` 与从 $n \sim 2$ 的所有递减整数相乘。根据阶乘的公式，还应该乘以 1，但是这并不会改变结果。

现在考虑使用递归的函数。该函数的关键是 $n! = n \times (n-1)!$ 。可以这样做是因为 $(n-1)!$ 是 $n-1 \sim 1$ 的所有正整数的乘积。因此， n 乘以 $n-1$ 就得到 n 的阶乘。阶乘的这一特性很适合使用递归。如果调用函数 `rfact()`，`rfact(n)` 是 $n * rfact(n-1)$ 。因此，通过调用 `rfact(n-1)` 来计算 `rfact(n)`，如程序清单 9.7 中所示。当然，必须要在满足某条件时结束递归，可以在 n 等于 0 时把返回值设为 1。

程序清单 9.7 中使用递归的输出和使用循环的输出相同。注意，虽然 `rfact()` 的递归调用不是函数的最后一行，但是当 $n > 0$ 时，它是该函数执行的最后一条语句，因此它也是尾递归。

既然用递归和循环来计算都没问题，那么到底应该使用哪一个？一般而言，选择循环比较好。首先，每次递归都会创建一组变量，所以递归使用的内存更多，而且每次递归调用都会把创建的一组新变量放在栈中。递归调用的数量受限于内存空间。其次，由于每次函数调用要花费一定的时间，所以递归的执行速度较慢。那么，演示这个程序示例的目的是什么？因为尾递归是递归中最简单的形式，比较容易理解。在某些情况下，不能用简单的循环代替递归，因此读者还是要好好理解递归。

9.3.4 递归和倒序计算

递归在处理倒序时非常方便（在解决这类问题中，递归比循环简单）。我们要解决的问题是：编写一个函数，打印一个整数的二进制数。二进制表示法根据 2 的幂来表示数字。例如，十进制数 234 实际上是 $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$ ，所以二进制数 101 实际上是 $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ 。二进制数由 0 和 1 表示。

我们要设计一个以二进制形式表示整数的方法或算法（algorithm）。例如，如何用二进制表示十进制数 5？在二进制中，奇数的末尾一定是 1，偶数的末尾一定是 0，所以通过 $5 \% 2$ 即可确定 5 的二进制数的最后一位是 1 还是 0。一般而言，对于数字 n ，其二进制的最后一位是 $n \% 2$ 。因此，计算的第一位数字实际上是待输出二进制数的最后一位。这一规律提示我们，在递归函数的递归调用之前计算 $n \% 2$ ，在递归调用之后打印计算结果。这样，计算的第一个值正好是最后一个打印的值。

要获得下一位数字，必须把原数除以 2。这种计算方法相当于在十进制下把小数点左移一位，如果计算结果是偶数，那么二进制的下一位数就是 0；如果是奇数，就是 1。例如， $5 / 2$ 得 2（整数除法），2 是偶数（ $2 \% 2$ 得 0），所以下一位二进制数是 0。到目前为止，我们已经获得 01。继续重复这个过程。 $2 / 2$ 得 1， $1 \% 2$ 得 1，所以下一位二进制数是 1。因此，我们得到 5 的等价二进制数是 101。那么，程序应该何时停止计算？当与 2 相除的结果小于 2 时停止计算，因为只要结果大于或等于 2，就说明还有二进制位。

每次除以 2 就相当于去掉一位二进制，直到计算出最后一位为止（如果不好理解，可以拿十进制数来做类比： $628 \% 10$ 得 8，因此 8 就是该数最后一位；而 $628 / 10$ 得 62，而 $62 \% 10$ 得 2，所以该数的下一位是 2，以此类推）。程序清单 9.8 演示了上述算法。

程序清单 9.8 binary.c 程序

```
/* binary.c -- 以二进制形式打印制整数 */
#include <stdio.h>
void to_binary(unsigned long n);

int main(void)
{
    unsigned long number;
    printf("Enter an integer (q to quit):\n");
    while (scanf("%lu", &number) == 1)
    {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
    printf("Done.\n");

    return 0;
}

void to_binary(unsigned long n) /* 递归函数 */
{
    int r;

    r = n % 2;
    if (n >= 2)
        to_binary(n / 2);
    putchar(r == 0 ? '0' : '1');

    return;
}
```

在该程序中，如果 r 的值是 0，`to_binary()` 函数就显示字符 '0'；如果 r 的值是 1，`to_binary()` 函数则显示字符 '1'。条件表达式 `r == 0 ? '0' : '1'` 用于把数值转换成字符。

下面是该程序的运行示例：

```
Enter an integer (q to quit):
9
Binary equivalent: 1001
Enter an integer (q to quit):
255
Binary equivalent: 11111111
Enter an integer (q to quit):
1024
Binary equivalent: 10000000000
Enter an integer (q to quit):
q
done.
```

不用递归，是否能实现这种用二进制形式表示整数的算法？当然可以。但是由于这种算法要首先计算最后一位二进制数，所以在显示结果之前必须把所有的位数都储存在别处（例如，数组）。第 15 章中会介绍一个不用递归实现该算法的例子。

9.3.5 递归的优缺点

递归既有优点也有缺点。优点是递归为某些编程问题提供了最简单的解决方案。缺点是一些递归算法会快速消耗计算机的内存资源。另外，递归不方便阅读和维护。我们用一个例子来说明递归的优缺点。

斐波那契数列的定义如下：第 1 个和第 2 个数字都是 1，而后续的每个数字都是其前两个数字之和。例如，该数列的前几个数是：1、1、2、3、5、8、13。斐波那契数列在数学界深受喜爱，甚至有专门研究它的刊物。不过，这不在本书的讨论范围之内。下面，我们要创建一个函数，接受正整数 n ，返回相应的斐波那契数值。

首先，来看递归。递归提供一个简单的定义。如果把函数命名为 `Fibonacci()`，那么如果 n 是 1 或 2，`Fibonacci(n)` 应返回 1；对于其他数值，则应返回 `Fibonacci(n-1)+Fibonacci(n-2)`：

```
unsigned long Fibonacci(unsigned n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

这个递归函数只是重述了数学定义的递归。该函数使用了双递归 (*double recursion*)，即函数每一级递归都要调用本身两次。这暴露了一个问题。

为了说明这个问题，假设调用 `Fibonacci(40)`。这是第 1 级递归调用，将创建一个变量 n 。然后在该函数中要调用 `Fibonacci()` 两次，在第 2 级递归中要分别创建两个变量 n 。这两次调用中的每次调用又会进行两次调用，因而在第 3 级递归中要创建 4 个名为 n 的变量。此时总共创建了 7 个变量。由于每级递归创建的变量都是上一级递归的两倍，所以变量的数量呈指数增长！在第 5 章中介绍过一个计算小麦粒数的例子，按指数增长很快就会产生非常大的值。在本例中，指数增长的变量数量很快就消耗掉计算机的大量内存，很可能导致程序崩溃。

虽然这是个极端的例子，但是该例说明：在程序中使用递归要特别注意，尤其是效率优先的程序。

所有的 C 函数皆平等

程序中的每个 C 函数与其他函数都是平等的。每个函数都可以调用其他函数，或被其他函数调用。这点与 Pascal 和 Modula-2 中的过程不同，虽然过程可以嵌套在另一个过程中，但是嵌套在不同过程中的过程之间不能相互调用。

`main()` 函数是否与其他函数不同？是的，`main()` 的确有点特殊。当 `main()` 与程序中的其他函数放在一起时，最开始执行的是 `main()` 函数中的第 1 条语句，但是这也是局限之处。`main()` 也可以被自己或其他函数递归调用——尽管很少这样做。

9.4 编译多源代码文件的程序

使用多个函数最简单的方法是把它们都放在同一个文件中，然后像编译只有一个函数的文件那样编译

该文件即可。其他方法因操作系统而异，下面将举例说明。

9.4.1 UNIX

假定在 UNIX 系统中安装了 UNIX C 编译器 cc（最初的 cc 已经停用，但是许多 UNIX 系统都给 cc 命令起了一个别名用作其他编译器命令，典型的是 gcc 或 clang）。假设 file1.c 和 file2.c 是两个内含 C 函数的文件，下面的命令将编译两个文件并生成一个名为 a.out 的可执行文件：

```
cc file1.c file2.c
```

另外，还生成两个名为 file1.o 和 file2.o 的目标文件。如果后来改动了 file1.c，而 file2.c 不变，可以使用以下命令编译第 1 个文件，并与第 2 个文件的目标代码合并：

```
cc file1.c file2.o
```

UNIX 系统的 make 命令可自动管理多文件程序，但是这超出了本书的讨论范围。

注意，OS X 的 Terminal 工具可以打开 UNIX 命令行环境，但是必须先下载命令行编译器（GCC 和 Clang）。

9.4.2 Linux

假定 Linux 系统安装了 GNU C 编译器 GCC。假设 file1.c 和 file2.c 是两个内含 C 函数的文件，下面的命令将编译两个文件并生成名为 a.out 的可执行文件：

```
gcc file1.c file2.c
```

另外，还生成两个名为 file1.o 和 file2.o 的目标文件。如果后来改动了 file1.c，而 file2.c 不变，可以使用以下命令编译第 1 个文件，并与第 2 个文件的目标代码合并：

```
gcc file1.c file2.o
```

9.4.3 DOS 命令行编译器

绝大多数 DOS 命令行编译器的工作原理和 UNIX 的 cc 命令类似，只不过使用不同的名称而已。其中一个区别是，对象文件的扩展名是 .obj，而不是 .o。一些编译器生成的不是目标代码文件，而是汇编语言或其他特殊代码的中间文件。

9.4.4 Windows 和苹果的 IDE 编译器

Windows 和 Macintosh 系统使用的集成开发环境中的编译器是面向项目的。项目（project）描述的是特定程序使用的资源。资源包括源代码文件。这种 IDE 中的编译器要创建项目来运行单文件程序。对于多文件程序，要使用相应的菜单命令，把源代码文件加入一个项目中。要确保所有的源代码文件都在项目列表中列出。许多 IDE 都不用在项目列表中列出头文件（即扩展名为.h 的文件），因为项目只管理使用的源代码文件，源代码文件中的 #include 指令管理该文件中使用的头文件。但是，Xcode 要在项目中添加头文件。

9.4.5 使用头文件

如果把 main() 放在第 1 个文件中，把函数定义放在第 2 个文件中，那么第 1 个文件仍然要使用函数原型。把函数原型放在头文件中，就不用在每次使用函数文件时都写出函数的原型。C 标准库就是这样做的，例如，把 I/O 函数原型放在 stdio.h 中，把数学函数原型放在 math.h 中。你也可以这样用自定义的函数文件。

另外，程序中经常用 C 预处理器定义符号常量。这种定义只储存了那些包含#define 指令的文件。如果把程序的一个函数放进一个独立的文件中，你也可以使用#define 指令访问每个文件。最直接的方法是

在每个文件中再次输入指令，但是这个方法既耗时又容易出错。另外，还会有维护的问题：如果修改了#define 定义的值，就必须在每个文件中修改。更好的做法是，把#define 指令放进头文件，然后在每个源文件中使用#include 指令包含该文件即可。

总之，把函数原型和已定义的字符常量放在头文件中是一个良好的编程习惯。我们考虑一个例子：假设要管理 4 家酒店的客房服务，每家酒店的房价不同，但是每家酒店所有房间的房价相同。对于预订住宿多天的客户，第 2 天的房费是第 1 天的 95%，第 3 天是第 2 天的 95%，以此类推（暂不考虑这种策略的经济效益）。设计一个程序让用户指定酒店和入住天数，然后计算并显示总费用。同时，程序要实现一份菜单，允许用户反复输入数据，除非用户选择退出。

程序清单 9.9、程序清单 9.10 和程序清单 9.11 演示了如何编写这样的程序。第 1 个程序清单包含 main() 函数，提供整个程序的组织结构。第 2 个程序清单包含支持的函数，我们假设这些函数在独立的文件中。最后，程序清单 9.11 列出了一个头文件，包含了该程序所有源文件中使用的自定义符号常量和函数原型。前面介绍过，在 UNIX 和 DOS 环境中，#include "hotels.h" 指令中的双引号表明被包含的文件位于当前目录中（通常是包含源代码的目录）。如果使用 IDE，需要知道如何把头文件合并成一个项目。

程序清单 9.9 usehotel.c 控制模块

```
/* usehotel.c -- 房间费率程序 */
/* 与程序清单 9.10 一起编译      */
#include <stdio.h>
#include "hotel.h" /* 定义符号常量，声明函数 */

int main(void)
{
    int nights;
    double hotel_rate;
    int code;

    while ((code = menu()) != QUIT)
    {
        switch (code)
        {
            case 1: hotel_rate = HOTEL1;
                      break;
            case 2: hotel_rate = HOTEL2;
                      break;
            case 3: hotel_rate = HOTEL3;
                      break;
            case 4: hotel_rate = HOTEL4;
                      break;
            default: hotel_rate = 0.0;
                      printf("Oops!\n");
                      break;
        }
        nights = getnights();
        showprice(hotel_rate, nights);
    }
    printf("Thank you and goodbye.\n");

    return 0;
}
```

程序清单 9.10 hotel.c 函数支持模块

```

/* hotel.c -- 酒店管理函数 */
#include <stdio.h>
#include "hotel.h"
int menu(void)
{
    int code, status;

    printf("\n%*s\n", STARS, STARS);
    printf("Enter the number of the desired hotel:\n");
    printf("1) Fairfield Arms          2) Hotel Olympic\n");
    printf("3) Chertworthy Plaza       4) The Stockton\n");
    printf("5) quit\n");
    printf("%*s\n", STARS, STARS);
    while ((status = scanf("%d", &code)) != 1 ||
           (code < 1 || code > 5))
    {
        if (status != 1)
            scanf("%*s"); // 处理非整数输入
        printf("Enter an integer from 1 to 5, please.\n");
    }

    return code;
}

int getnights(void)
{
    int nights;

    printf("How many nights are needed? ");
    while (scanf("%d", &nights) != 1)
    {
        scanf("%*s"); // 处理非整数输入
        printf("Please enter an integer, such as 2.\n");
    }

    return nights;
}

void showprice(double rate, int nights)
{
    int n;
    double total = 0.0;
    double factor = 1.0;

    for (n = 1; n <= nights; n++, factor *= DISCOUNT)
        total += rate * factor;
    printf("The total cost will be $%.2f.\n", total);
}

```

程序清单 9.11 hotel.h 头文件

```

/* hotel.h -- 符号常量和 hotel.c 中所有函数的原型 */
#define QUIT      5

```

```

#define HOTEL1 180.00
#define HOTEL2 225.00
#define HOTEL3 255.00
#define HOTEL4 355.00
#define DISCOUNT 0.95
#define STARS "*****"

// 显示选择列表
int menu(void);

// 返回预订天数
int getnights(void);

// 根据费率、入住天数计算费用
// 并显示结果
void showprice(double rate, int nights);
下面是这个多文件程序的运行示例：

*****
Enter the number of the desired hotel:
1) Fairfield Arms      2) Hotel Olympic
3) Chertworthy Plaza  4) The Stockton
5) quit
*****
3
How many nights are needed? 1
The total cost will be $255.00.

*****
Enter the number of the desired hotel:
1) Fairfield Arms      2) Hotel Olympic
3) Chertworthy Plaza  4) The Stockton
5) quit
*****
4
How many nights are needed? 3
The total cost will be $1012.64.

*****
Enter the number of the desired hotel:
1) Fairfield Arms      2) Hotel Olympic
3) Chertworthy Plaza  4) The Stockton
5) quit
*****
5
Thank you and goodbye.

```

顺带一提，该程序中有几处编写得很巧妙。尤其是，`menu()` 和 `getnights()` 函数通过测试 `scanf()` 的返回值来跳过非数值数据，而且调用 `scanf("%*s")` 跳至下一个空白字符。注意，`menu()` 函数中是如何检查非数值输入和超出范围的数据：

```
while ((status = scanf("%d", &code)) != 1 || (code < 1 || code > 5))
```

以上代码段利用了 C 语言的两个规则：从左往右对逻辑表达式求值；一旦求值结果为假，立即停止求值。在该例中，只有在 `scanf()` 成功读入一个整数值后，才会检查 `code` 的值。

用不同的函数处理不同的任务时应检查数据的有效性。当然，首次编写 menu() 或 getnights() 函数时可以暂不添加这一功能，只写一个简单的 scanf() 即可。待基本版本运行正常后，再逐步改善各模块。

9.5 查找地址：&运算符

指针 (pointer) 是 C 语言最重要的（有时也是最复杂的）概念之一，用于储存变量的地址。前面使用的 scanf() 函数中就使用地址作为参数。概括地说，如果主调函数不使用 return 返回的值，则必须通过地址才能修改主调函数中的值。接下来，我们将介绍带地址参数的函数。首先介绍一元&运算符的用法。

一元&运算符给出变量的存储地址。如果 pooh 是变量名，那么&pooh 是变量的地址。可以把地址看作是变量在内存中的位置。假设有下面的语句：

```
pooh = 24;
```

假设 pooh 的存储地址是 0B76（PC 地址通常用十六进制形式表示）。那么，下面的语句：

```
printf("%d %p\n", pooh, &pooh);
```

将输出如下内容（%p 是输出地址的转换说明）：

```
24 0B76
```

程序清单 9.12 中使用了这个运算符查看不同函数中的同名变量分别储存在什么位置。

程序清单 9.12 loccheck.c 程序

```
/* loccheck.c -- 查看变量被储存在何处 */
#include <stdio.h>
void mikado(int); /* 函数原型 */
int main(void)
{
    int pooh = 2, bah = 5; /* main() 的局部变量 */

    printf("In main(), pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("In main(), bah = %d and &bah = %p\n", bah, &bah);
    mikado(pooh);

    return 0;
}

void mikado(int bah) /* 定义函数 */
{
    int pooh = 10; /* mikado() 的局部变量 */

    printf("In mikado(), pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("In mikado(), bah = %d and &bah = %p\n", bah, &bah);
}
```

程序清单 9.12 中使用 ANSI C 的%p 格式打印地址。我们的系统输出如下：

```
In main(), pooh = 2 and &pooh = 0x7fff5fbff8e8
In main(), bah = 5 and &bah = 0x7fff5fbff8e4
In mikado(), pooh = 10 and &pooh = 0x7fff5fbff8b8
In mikado(), bah = 2 and &bah = 0x7fff5fbff8bc
```

实现不同，%p 表示地址的方式也不同。然而，许多实现都如本例所示，以十六进制显示地址。顺带一提，每个十六进制数对应 4 位，该例显示 12 个十六进制数，对应 48 位地址。

该例的输出说明了什么？首先，两个 pooh 的地址不同，两个 bah 的地址也不同。因此，和前面介绍的一样，计算机把它们看成 4 个独立的变量。其次，函数调用 mikado(pooh) 把实际参数 (main() 中的 pooh) 的值 (2) 传递给形式参数 (mikado() 中的 bah)。注意，这种传递只传递了值。涉及的两个变量 (main() 中的 pooh 和 mikado() 中的 bah) 并未改变。

我们强调第 2 点，是因为这并不是在所有语言中都成立。例如，在 FORTRAN 中，子例程会影响主调例程的原始变量。子例程的变量名可能与原始变量不同，但是它们的地址相同。但是，在 C 语言中不是这样。每个 C 函数都有自己的变量。这样做更可取，因为这样做可以防止原始变量被被调函数中的副作用意外修改。然而，正如下节所述，这也带来了一些麻烦。

9.6 更改主调函数中的变量

有时需要在一个函数中更改其他函数的变量。例如，普通的排序任务中交换两个变量的值。假设要交换两个变量 x 和 y 的值。简单的思路是：

```
x = y;
y = x;
```

这完全不起作用，因为执行到第 2 行时，x 的原始值已经被 y 的原始值替换了。因此，要多写一行代码，储存 x 的原始值：

```
temp = x;
x = y;
y = temp;
```

上面这 3 行代码便可实现交换值的功能，可以编写成一个函数并构造一个驱动程序来测试。在程序清单 9.13 中，为清楚地表明变量属于哪个函数，在 main() 中使用变量 x 和 y，在 interchange() 中使用 u 和 v。

程序清单 9.13 swap1.c 程序

```
/* swap1.c -- 第 1 个版本的交换函数 */
#include <stdio.h>
void interchange(int u, int v); /* 声明函数 */

int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v) /* 定义函数 */
{
    int temp;

    temp = u;
    u = v;
    v = temp;
}
```

运行该程序后，输出如下：

```
Originally x = 5 and y = 10.
Now x = 5 and y = 10.
```

两个变量的值并未交换！我们在 interchange() 中添加一些打印语句来检查错误（见程序清单 9.14）。

程序清单 9.14 swap2.c 程序

```
/* swap2.c -- 查找 swap1.c 的问题 */
#include <stdio.h>
void interchange(int u, int v);

int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)
{
    int temp;

    printf("Originally u = %d and v = %d.\n", u, v);
    temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```

下面是该程序的输出：

```
Originally x = 5 and y = 10.
Originally u = 5 and v = 10.
Now u = 10 and v = 5.
Now x = 5 and y = 10.
```

看来，interchange() 没有问题，它交换了 u 和 v 的值。问题出在把结果传回 main() 时。interchange() 使用的变量并不是 main() 中的变量。因此，交换 u 和 v 的值对 x 和 y 的值没有影响！是否能用 return 语句把值传回 main()？当然可以，在 interchange() 的末尾加上下面一行语句：

```
return(u);
```

然后修改 main() 中的调用：

```
x = interchange(x, y);
```

这只能改变 x 的值，而 y 的值依旧没变。用 return 语句只能把被调函数中的一个值传回主调函数，但是现在要传回两个值。这没问题！不过，要使用指针。

9.7 指针简介

指针？什么是指针？从根本上看，指针 (pointer) 是一个值为内存地址的变量（或数据对象）。正如 char 类型变量的值是字符，int 类型变量的值是整数，指针变量的值是地址。在 C 语言中，指针有许多用法。

本章将介绍如何把指针作为函数参数使用，以及为何要这样用。

假设一个指针变量名是 `ptr`，可以编写如下语句：

```
ptr = &pooh; // 把 pooh 的地址赋给 ptr
```

对于这条语句，我们说 `ptr` “指向” `pooh`。`ptr` 和 `&pooh` 的区别是 `ptr` 是变量，而 `&pooh` 是常量。或者，`ptr` 是可修改的左值，而 `&pooh` 是右值。还可以把 `ptr` 指向别处：

```
ptr = &bah; // 把 ptr 指向 bah，而不是 pooh
```

现在 `ptr` 的值是 `bah` 的地址。

要创建指针变量，先要声明指针变量的类型。假設想把 `ptr` 声明为储存 `int` 类型变量地址的指针，就要使用下面介绍的新运算符。

9.7.1 间接运算符：*

假设已知 `ptr` 指向 `bah`，如下所示：

```
ptr = &bah;
```

然后使用间接运算符 * (*indirection operator*) 找出储存在 `bah` 中的值，该运算符有时也称为解引用运算符 (*dereferencing operator*)。不要把间接运算符和二元乘法运算符 (*) 混淆，虽然它们使用的符号相同，但语法功能不同。

```
val = *ptr; // 找出 ptr 指向的值
```

语句 `ptr = &bah;` 和 `val = *ptr;` 放在一起相当于下面的语句：

```
val = bah;
```

由此可见，使用地址和间接运算符可以间接完成上面这条语句的功能，这也是“间接运算符”名称的由来。

小结：与指针相关的运算符

地址运算符：&

一般注解：

后跟一个变量名时，`&` 给出该变量的地址。

示例：

`&nurse` 表示变量 `nurse` 的地址。

地址运算符：*

一般注解：

后跟一个指针名或地址时，`*` 给出储存在指针指向地址上的值。

示例：

```
nurse = 22;
```

```
ptr = &nurse; // 指向 nurse 的指针
```

```
val = *ptr; // 把 ptr 指向的地址上的值赋给 val
```

执行以上 3 条语句的最终结果是把 22 赋给 `val`。

9.7.2 声明指针

相信读者已经很熟悉如何声明 `int` 类型和其他基本类型的变量，那么如何声明指针变量？你也许认为

是这样声明：

```
pointer ptr; // 不能这样声明指针
```

为什么不能这样声明？因为声明指针变量时必须指定指针所指向变量的类型，因为不同的变量类型占用不同的存储空间，一些指针操作要求知道操作对象的大小。另外，程序必须知道储存在指定地址上的数据类型。`long` 和 `float` 可能占用相同的存储空间，但是它们储存数字却大相径庭。下面是一些指针的声明示例：

```
int * pi; // pi 是指向 int 类型变量的指针
char * pc; // pc 是指向 char 类型变量的指针
float * pf, * pg; // pf、pg 都是指向 float 类型变量的指针
```

类型说明符表明了指针所指向对象的类型，星号 (*) 表明声明的变量是一个指针。`int * pi;` 声明的意思是 `pi` 是一个指针，`*pi` 是 `int` 类型（见图 9.5）。

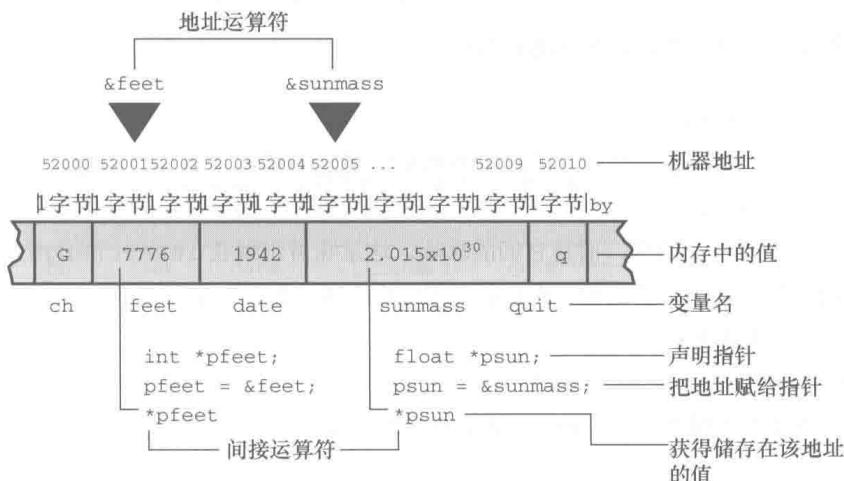


图 9.5 声明并使用指针

*和指针名之间的空格可有可无。通常，程序员在声明时使用空格，在解引用变量时省略空格。

`pc` 指向的值 (`*pc`) 是 `char` 类型。`pc` 本身是什么类型？我们描述它的类型是“指向 `char` 类型的指针”。`pc` 的值是一个地址，在大部分系统内部，该地址由一个无符号整数表示。但是，不要把指针认为是整数类型。一些处理整数的操作不能用来处理指针，反之亦然。例如，可以把两个整数相乘，但是不能把两个指针相乘。所以，指针实际上是一个新类型，不是整数类型。因此，如前所述，ANSI C 专门为指针提供了`%p` 格式的转换说明。

9.7.3 使用指针在函数间通信

我们才刚刚接触指针，指针的世界丰富多彩。本节着重介绍如何使用指针解决函数间的通信问题。请看程序清单 9.15，该程序在 `interchange()` 函数中使用了指针参数。稍后我们将对该程序做详细分析。

程序清单 9.15 swap3.c 程序

```
/* swap3.c -- 使用指针解决交换函数的问题 */
#include <stdio.h>
void interchange(int * u, int * v);

int main(void)
{
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x, y);
}
```

```

    interchange(&x, &y); // 把地址发送给函数
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int * u, int * v)
{
    int temp;
    temp = *u; // temp 获得 u 所指向对象的值
    *u = *v;
    *v = temp;
}

```

该程序是否能正常运行？下面是程序的输出：

```

Originally x = 5 and y = 10.
Now x = 10 and y = 5.

```

没问题，一切正常。接下来，我们分析程序清单 9.15 的运行情况。首先看函数调用：

```
interchange(&x, &y);
```

该函数传递的不是 x 和 y 的值，而是它们的地址。这意味着出现在 interchange() 原型和定义中的形式参数 u 和 v 将把地址作为它们的值。因此，应把它们声明为指针。由于 x 和 y 是整数，所以 u 和 v 是指向整数的指针，其声明如下：

```
void interchange (int * u, int * v)
```

接下来，在函数体中声明了一个交换值时必需的临时变量：

```
int temp;
```

通过下面的语句把 x 的值储存在 temp 中：

```
temp = *u;
```

记住，u 的值是&x，所以 u 指向 x。这意味着用*u 即可表示 x 的值，这正是我们需要的。不要写成这样：

```
temp = u; /* 不要这样做 */
```

因为这条语句赋给 temp 的是 x 的地址（u 的值就是 x 的地址），而不是 x 的值。函数要交换的是 x 和 y 的值，而不是它们的地址。

与此类似，把 y 的值赋给 x，要使用下面的语句：

```
*u = *v;
```

这条语句相当于：

```
x = y;
```

我们总结一下该程序示例做了什么。我们需要一个函数交换 x 和 y 的值。把 x 和 y 的地址传递给函数，我们让 interchange() 访问这两个函数。使用指针和*运算符，该函数可以访问储存在这些位置的值并改变它们。

可以省略 ANSI C 风格的函数原型中的形参名，如下所示：

```
void interchange(int *, int *);
```

一般而言，可以把变量相关的两类信息传递给函数。如果这种形式的函数调用，那么传递的是 x 的值：

```
function1(x);
```

如果下面形式的函数调用，那么传递的是 x 的地址：

```
function2(&x);
```

第 1 种形式要求函数定义中的形式参数必须是一个与 `x` 的类型相同的变量：

```
int function1(int num)
```

第 2 种形式要求函数定义中的形式参数必须是一个指向正确类型的指针：

```
int function2(int * ptr)
```

如果要计算或处理值，那么使用第 1 种形式的函数调用；如果要在被调函数中改变主调函数的变量，则使用第 2 种形式的函数调用。我们用过的 `scanf()` 函数就是这样。当程序要把一个值读入变量时（如本例中的 `num`），调用的是 `scanf("%d", &num)`。`scanf()` 读取一个值，然后把该值储存到指定的地址上。

对本例而言，指针让 `interchange()` 函数通过自己的局部变量改变 `main()` 中变量的值。

熟悉 Pascal 和 Modula-2 的读者应该看出第 1 种形式和 Pascal 的值参数相同，第 2 种形式和 Pascal 的变量参数类似。C++ 程序员可能认为，既然 C 和 C++ 都使用指针变量，那么 C 应该也有引用变量。让他们失望了，C 没有引用变量。对 BASIC 程序员而言，可能很难理解整个程序。如果觉得本节的内容晦涩难懂，请多做一些相关的编程练习，你会发现指针非常简单实用（见图 9.6）。



图 9.6 按字节寻址系统（如 PC）中变量的名称、地址和值

变量：名称、地址和值

通过前面的讨论发现，变量的名称、地址和变量的值之间关系密切。我们来进一步分析。

编写程序时，可以认为变量有两个属性：名称和值（还有其他性质，如类型，暂不讨论）。计算机编译和加载程序后，认为变量也有两个属性：地址和值。地址就是变量在计算机内部的名称。

在许多语言中，地址都归计算机管，对程序员隐藏。然而在 C 中，可以通过&运算符访问地址，通过*运算符获得地址上的值。例如，`&barn` 表示变量 `barn` 的地址，使用函数名即可获得变量的数值。例如，`printf("%d\n", barn)` 打印 `barn` 的值，使用*运算符即可获得储存在地址上的值。如果 `pbar = &barn;`，那么 `*pbar` 表示的是储存在 `&barn` 地址上的值。

简而言之，普通变量把值作为基本量，把地址作为通过&运算符获得的派生量，而指针变量把地址作为基本量，把值作为通过*运算符获得的派生量。

虽然打印地址可以满足读者好奇心，但是这并不是&运算符的主要用途。更重要的是使用&、*和指针可以操纵地址和地址上的内容，如 `swap3.c` 程序（程序清单 9.15）所示。

小结：函数

形式：

典型的 ANSI C 函数的定义形式为：

返回类型 名称 (形参声明列表)

函数体

形参声明列表是用逗号分隔的一系列变量声明。除形参变量外，函数的其他变量均在函数体的花括号之内声明。

示例：

```
int diff(int x, int y) // ANSI C
{ // 函数体开始
    int z;           // 声明局部变量
    z = x - y;
    return z; // 返回一个值
} // 函数体结束
```

传递值：

实参用于把值从主调函数传递给被调函数。如果变量 a 和 b 的值分别是 5 和 2，那么调用：
c = diff(a,b);

把 5 和 2 分别传递给变量 x 和 y。5 和 2 称为实际参数（简称实参），diff() 函数定义中的变量 x 和 y 称为形式参数（简称形参）。使用关键字 return 把被调函数中的一个值传回主调函数。本例中，c 接受 z 的值 3。被调函数一般不会改变主调函数中的变量，如果要改变，应使用指针作为参数。如果希望把更多的值传回主调函数，必须这么做。

函数的返回类型：

函数的返回类型指的是函数返回值的类型。如果返回值的类型与声明的返回类型不匹配，返回值将被转换成函数声明的返回类型。

函数签名：

函数的返回类型和形参列表构成了函数签名。因此，函数签名指定了传入函数的值的类型和函数返回值的类型。

示例：

```
double duff(double, int); // 函数原型
int main(void)
{
    double q, x;
    int n;
    ...
    q = duff(x, n);           // 函数调用
    ...
}
double duff(double u, int k) // 函数定义
{
    double tor;
    ...
    return tor; // 返回 double 类型的值
}
```

9.8 关键概念

如果想用 C 编出高效灵活的程序，必须理解函数。把大型程序组织成若干函数非常有用，甚至很关键。如果让一个函数处理一个任务，程序会更好理解，更方便调试。要理解函数是如何把信息从一个函数传递

到另一函数，也就是说，要理解函数参数和返回值的工作原理。另外，要明白函数形参和其他局部变量都属于函数私有，因此，声明在不同函数中的同名变量是完全不同的变量。而且，函数无法直接访问其他函数中的变量。这种限制访问保护了数据的完整性。但是，当确实需要在函数中访问另一个函数的数据时，可以把指针作为函数的参数。

9.9 本章小结

函数可以作为组成大型程序的构件块。每个函数都应该有一个单独且定义好的功能。使用参数把值传给函数，使用关键字 `return` 把值返回函数。如果函数返回的值不是 `int` 类型，则必须在函数定义和函数原型中指定函数的类型。如果需要在被调函数中修改主调函数的变量，使用地址或指针作为参数。

ANSI C 提供了一个强大的工具——函数原型，允许编译器验证函数调用中使用的参数个数和类型是否正确。

C 函数可以调用本身，这种调用方式被称为递归。一些编程问题要用递归来解决，但是递归不仅消耗内存多，效率不高，而且费时。

9.10 复习题

复习题的参考答案在附录 A 中。

1. 实际参数和形式参数的区别是什么？
2. 根据下面各函数的描述，分别编写它们的 ANSI C 函数头。注意，只需写出函数头，不用写函数体。
 - a. `donut()` 接受一个 `int` 类型的参数，打印若干（参数指定数目）个 0
 - b. `gear()` 接受两个 `int` 类型的参数，返回 `int` 类型的值
 - c. `guess()` 不接受参数，返回一个 `int` 类型的值
 - d. `stuff_it()` 接受一个 `double` 类型的值和 `double` 类型变量的地址，把第 1 个值储存在指定位置
3. 根据下面各函数的描述，分别编写它们的 ANSI C 函数头。注意，只需写出函数头，不用写函数体。
 - a. `n_to_char()` 接受一个 `int` 类型的参数，返回一个 `char` 类型的值
 - b. `digit()` 接受一个 `double` 类型的参数和一个 `int` 类型的参数，返回一个 `int` 类型的值
 - c. `which()` 接受两个可储存 `double` 类型变量的地址，返回一个 `double` 类型的地址
 - d. `random()` 不接受参数，返回一个 `int` 类型的值
4. 设计一个函数，返回两整数之和。
5. 如果把复习题 4 改成返回两个 `double` 类型的值之和，应如何修改函数？
6. 设计一个名为 `alter()` 的函数，接受两个 `int` 类型的变量 `x` 和 `y`，把它们的值分别改成两个变量之和以及两变量之差。
7. 下面的函数定义是否正确？


```
void salami(num)
{
    int num, count;
    for (count = 1; count <= num; num++)
        printf(" O salami mio!\n");
}
```

8. 编写一个函数，返回 3 个整数参数中的最大值。

9. 给定下面的输出：

```
Please choose one of the following:  
1) copy files      2) move files  
3) remove files   4) quit  
Enter the number of your choice:
```

- 编写一个函数，显示一份有 4 个选项的菜单，提示用户进行选择（输出如上所示）。
- 编写一个函数，接受两个 int 类型的参数分别表示上限和下限。该函数从用户的输入中读取整数。如果整数超出规定上下限，函数再次打印菜单（使用 a 部分的函数）提示用户输入，然后获取一个新值。如果用户输入的整数在规定范围内，该函数则把该整数返回主调函数。如果用户输入一个非整数字符，该函数应返回 4。
- 使用本题 a 和 b 部分的函数编写一个最小型的程序。最小型的意思是，该程序不需要实现菜单中各选项的功能，只需显示这些选项并获取有效的响应即可。

9.11 编程练习

- 设计一个函数 `min(x, y)`，返回两个 `double` 类型值的较小值。在一个简单的驱动程序中测试该函数。
- 设计一个函数 `chline(ch, i, j)`，打印指定的字符 `j` 行 `i` 列。在一个简单的驱动程序中测试该函数。
- 编写一个函数，接受 3 个参数：一个字符和两个整数。字符参数是待打印的字符，第 1 个整数指定一行中打印字符的次数，第 2 个整数指定打印指定字符的行数。编写一个调用该函数的程序。
- 两数的调和平均数这样计算：先得到两数的倒数，然后计算两个倒数的平均值，最后取计算结果的倒数。编写一个函数，接受两个 `double` 类型的参数，返回这两个参数的调和平均数。
- 编写并测试一个函数 `larger_of()`，该函数把两个 `double` 类型变量的值替换为较大的值。例如，`larger_of(x, y)` 会把 `x` 和 `y` 中较大的值重新赋给两个变量。
- 编写并测试一个函数，该函数以 3 个 `double` 变量的地址作为参数，把最小值放入第 1 个函数，中间值放入第 2 个变量，最大值放入第 3 个变量。
- 编写一个函数，从标准输入中读取字符，直到遇到文件结尾。程序要报告每个字符是否是字母。如果是，还要报告该字母在字母表中的数值位置。例如，`c` 和 `C` 在字母表中的位置都是 3。合并一个函数，以一个字符作为参数，如果该字符是一个字母则返回一个数值位置，否则返回 -1。
- 第 6 章的程序清单 6.20 中，`power()` 函数返回一个 `double` 类型数的正整数次幂。改进该函数，使其能正确计算负幂。另外，函数要处理 0 的任何次幂都为 0，任何数的 0 次幂都为 1（函数应报告 0 的 0 次幂未定义，因此把该值处理为 1）。要使用一个循环，并在程序中测试该函数。
- 使用递归函数重写编程练习 8。
- 为了让程序清单 9.8 中的 `to_binary()` 函数更通用，编写一个 `to_base_n()` 函数接受两个在 2~10 范围内的参数，然后以第 2 个参数中指定的进制打印第 1 个参数的数值。例如，`to_base_n(129, 8)` 显示的结果为 201，也就是 129 的八进制数。在一个完整的程序中测试该函数。
- 编写并测试 `Fibonacci()` 函数，该函数用循环代替递归计算斐波那契数。