第四章　基本程序设计技术

读者在学习中应已试验了一些程序示例，做了些练习。写程序的过程中确实要处理许多琐碎细节，但它又是很有趣的工作，是对智力的挑战。为完成一个程序，首先要分析问题、寻找解决方案，需要聪明才智和想象力，各种相关领域的知识和技术都可能得到应用。要把设计变成现实，变成可执行的程序，则既需要智力发挥，又需要有条有理的工作，还需要极端细心。一个小错就可能使程序无法编译或不能正确执行。当然，高度精确性也是现代社会的特征，写程序的过程能给读者许多有价值的体验。

学习程序设计都需要经历类似的熟悉过程，但并不是说这里的学习就是简单经验积累，也不意味着程序做多了就一定能做好。多年实践使人认识到程序设计的许多规律性，总结出许多模式、方法和技术，也进一步研究了许多理论问题。

学习程序设计，一开始就应该注意程序设计中的规律性。正确的好的程序不可能随随便便写出来，也不应该是修修补补改出来的。只有注意写好小程序，弄明白最基本的道理，才能进一步写出更大更复杂的程序。这些都是本书各章中特别强调的问题。

在进一步了解C语言的其他功能之前，我们应学会把已知的东西应用于实际程序设计，这就是本章的目的。前面讨论了三种结构化流程模式和相关语言结构。顺序模式最简单，很容易用复合语句实现。选择模式的最重要问题就是正确描述条件，考虑不同条件下应完成的动作，用条件语句实现也不困难。学习程序设计初期的主要难点是重复执行模式，它比较复杂，用循环实现时牵涉到的问题较多，这是本章讨论的重点。此外本章还要讨论基本输入输出，采用递归的方式编程，并介绍C语言的其他控制结构和控制语句。

4.1　循环程序设计

写出写好循环首先要发现计算过程中可能需要（应该用）循环。在分析问题时，应注意识别计算过程中需要重复执行的类似动作，这常常是重要线索，说明可能需要引进一个循环，统一描述和处理这些重复性动作。常见如：需要一批可以按统一规律计算的数据；需要反复从一个结果算出另一结果；需要对一系列类似数据做同样处理等等。这些情况都可看着重复性计算，如果重复次数多，或次数无法确定，就应考虑用循环结构描述。

从发现重复性动作到建立起一个循环结构，还需考虑和解决许多具体问题。通常包括：循环中涉及哪些变量？循环开始前应给它们什么初值？循环中这些变量应如何改变？在什么情况下应该继续（或应该终止）循环？循环终止后如何得到所需结果？如此等等。具体问题还包括使用语言里的哪种结构实现循环等。

本节将讨论一批程序设计实例，描述的各种典型循环程序设计问题。对于许多实例都采用了首先分析问题，发掘完成程序的线索，最终完成能解决问题的程序的叙述方式。这样讨论是为使读者能看到“从问题到程序”的思维过程。对许多例子给出了多个能解决问题的不同程序，并着重说明其差异或优缺点。这样做是希望读者能理解：程序设计不是教条，即使对一个典型问题，也没有需要背的标准解答。非极端简单的问题总有许多种解决方法，可以写出出许多有着或多或少形式的或实质的差异程序。许多程序往往各有长短。当然，能写出多个程序，并不说明这些程序都有同等价值。实际上，“正确的”程序也常有优劣之分。下面讨论里也会提出一些看法。

4.1.1 基本循环方式

假设现在要求出从13到315的所有整数之和。显然的解法是写一个循环，这时需要用

一个变量保存求出的和，计算过程中用它保存部分和。循环中顺序将各个数加上去，直至所有的数加完了，部分和就变成了完全和。为此实现这一循环，还就需要一个变量保存应加入的数的轨迹，每次重复时将这个变量加 1。这正好是 for 语句的循环形式。

假定已经定义了名为 sum 总和变量和名为 n 的循环变量。n 的取值应包括 [13, 213] 范围中所有的整数。一种典型实现方式称为*向上循环*，即让循环变量 n 从最小值开始逐步增加，直至达到取值范围的最大值。这样写出的循环具有如下形式：

```
for (sum = 0, n = 13; n <= 315; ++n)
    sum += n;
```

对于这个问题，同样可以采用向下循环的方式：

```
for (sum = 0, n = 315; n >= 13; --n)
    sum += n;
```

对于这个具体问题而言，采用向上或者向下循环的效果完全一样。许多其他循环的情况与此类似，都可以自由选择向上或者向下的方式。在这种情况下，人们一般采用向上循环的方式，因为这种方式似乎更符合人的习惯。也确实存在一些循环，对它们必须采用某种特定循环方式，否则就会导致程序错误。今后读者会遇到这样的例子。

完全可以用 while 语句重写上述循环。实际上，while 语句和 for 语句具有同样表达能力，用 for 循环表述的代码都可等价地翻译为用 while 语句写的代码，反之亦然。

有时我们需要的不是一个范围内的所有整数值，而是具有同等间隔的值。例如，需要求出 [13, 315] 间每隔 7 的各个整数之和。写出相应数学公式并不难（上例也一样），采用循环语句写也很简单：

```
for (sum = 0, n = 13; n <= 315; n += 7)
    sum += n;
```

在写循环时，人们一般不赞成采用浮点数控制循环的次数，尤其是在增量为小数或者包含小数时。例如，假设我们需要求从 0 到 100，每隔 0.2 的各数的平方和：

```
double sum, x;
for (sum = 0.0, x = 0.2; x <= 100.0; x += 0.2)
    sum += x * x;
```

因为浮点数运算有误差，我们不能保证这一循环恰好重复了 500 次，因此就不能保证它一定得到所需结果。这一循环的正确写法是：

```
int n;
double sum, x;
for (sum = 0.0, n = 1; n <= 500; ++n) {
    x = 0.2 * n;
    sum += x * x;
}
```

关于浮点误差的问题，下面还有更详细的讨论。

4.1.2 求出一系列完全平方数

问题：写一个程序，打印出 1 到 200 之间的所有完全平方数，即那些是另一个数的平方的数。通过分析可以发现，这样一个简单问题也有许多不同解法。

第一种方法

一种最显然的方法是逐个检查 1 到 200 的所有整数，遇到完全平方数时就打印。这一计算过程中有一系列重复动作（每次检查一个数），可以用循环描述。循环中用一个变量，顺序取被检查的那些值，从 1 开始每次加 1。这样就可以得到所需循环的基本框架：

```
for (n = 1; n <= 200; ++n)
    if (n 是完全平方数) 打印 n;
```

这里假设 n 是已有定义的整型变量，循环中它遍历从 1 到 200 的各个值。这种循环特别适合用 for 语句描述。

上面只是一个程序框架，其中有些部分还不是用程序语言精确描述的，需要在随后的工

作中填充。显然打印语句很容易，剩下的问题就是条件语句中的判断。由于 C 语言没有直接判断一个数是否为完全平方数的手段，这个问题还需要做进一步分析。

假设被考查的整数是 n，一种可能方式是从 1 开始检查各个整数，看是否有一个数的平方正好为 n。如果有，n 就是完全平方数；否则 n 就不是完全平方数。这个过程又构成一个循环（循环里的循环），需要用另一变量（例如 m）记录检查中所用的值。这就又产生了问题：m 取值从那里开始，什么条件下应当继续（或结束）。显然 m 应从 1 开始取值，而一旦其平方大于 n 就没有必要继续检查了，因为更大的数不会是 n 的平方根。

这样可以写出内部循环如下：

```
for (m = 1; m * m <= n; ++m)
    if (m * m == n) 打印 n;
```

循环中至多只有一个值的平方等于 n。如果 n 是完全平方数，这个值就会打印一次，且仅仅一次。把这些代码组合起来，就可以得到下面的程序：

```
#include <stdio.h>

main () {
    int m, n;
    for (n = 1; n <= 200; ++n)
        for (m = 1; m * m <= n; ++m)
            if (m * m == n)
                printf("%d ", n);
    printf("\n");  /* 最后换一行 */
}
```

第二种方法

这里要求打印 1 到 200 间的所有完全平方数，它们一定是从 1 开始的一些整数的平方。由这个想法可以得到另一种解决方案：用一个循环计数变量 n，从 1 开始逐个打印 n 值的平方，直到 n 的平方大于 200 为止。最后这句话也就是循环结束条件。按这种想法写出的程序更简单，只需一层循环。用 for 语句写出的程序主要部分如下：

```
for (n = 1; n * n <= 200; ++n)
    printf("%d ", n * n); /* 注意应当打印什么 */
```

不难将以它为基础写出一个程序，这件事留给读者完成。

还有一种可能的方法是递推。不难发现，平方数序列具有如下递推关系：

$$\alpha_1 = 1$$

$$\alpha_{n+1} = \alpha_n + 2n + 1$$

利用这个公式可以写出另一程序，还可以写出只用加法的程序。这些也请读者考虑。

从这个例子可以看到：即使是很简单的问题，也可能有许多不同的求解途径，写出的程序大相径庭。如果对问题做细致深入的分析，就可能发现许多解决问题的线索。

4.1.3 判断素数（谓词函数）

问题：写一个函数，判断一个整数（参数）是否为素数。

判断只有成立和不成立两种情况，完成判断的函数是一类特殊函数（也称为谓词），它们的返回值被作为逻辑值使用，通常用来控制程序流程，或放在条件表达式的控制部分。人们通常令判断函数返回 0 或 1，用 1 表示判断成立（在这里表示是素数），0 表示判断不成立（不是素数）。这样，要定义的函数的类型特征可以取定为：

```
int isprime(int)
```

判断一个数（例如 n）是否素数有许多高级的数学方法，这里只考虑最直接而简单的方法，就是设法确定它有无真因子。m 整除 n 可以用条件 (n%m == 0) 描述，如果 m<n 而且 m 不是 1，那它就是真因子了。这样，检查素数的最简单方法就是令变量 m 由 2 开始递增取

值，一个个试除 n。这可以通过一个循环完成。

下一个问题是 m 试除到什么时候结束。显然，试完所有小于 n 的值能保证不会漏掉任何可能性。稍加分析不难看出，只要试到 m*m>n 就够了，继续试下去已经没有意义（这一论断的合理性请读者考虑）。有了上述分析，写出下面定义已经很自然了：

```
int isprime (int n) { /* 判断一个数是否素数 */
    int m = 2;
    for ( ; m * m <= n; ++m)
        if (n % m == 0) return 0; /* 发现因子，不是素数 */
    return 1;  /* 可能性均考虑过，没有因子，是素数 */
}
```

发现 n 的一个因子已经可以做出结论，不必继续循环了。代码中的 return 语句导致函数结束，也使函数体里的循环结束。这是一种从循环中退出的方式。

这个函数定义还有不完善之处，例如，它对 1 将给出“是素数”的判断。对 0 和负数也会给出不合理结果。解决这些问题并不难，只要在循环前加上特殊情况处理。例如：

```
if (n <= 1) return 0;
```

在写一个程序（或一个函数）之前，首先应该仔细分析需要考虑的情况。完成之后还应该仔细检查，看看是否有什么遗漏。如果事先分析周全，应该能看到这些问题。

4.1.4 艰难旅程（浮点误差）

问题：假定有一只乌龟决心去做环球旅行。出发时它踌躇满志，第一秒四脚飞奔，爬了 1 米。随着体力和毅力的下降，它第二秒钟爬了 1/2 米，第三秒钟爬了 1/3 米，第四秒钟爬了 1/4 米，如此等等。现在问这只乌龟一小时能爬出多远？爬出 20 米需要多少时间？

显然，要计算的是无穷和式 $\sum_{n=1}^{\infty}\frac{1}{n}$ 前面的有限一段之和。由数学知识有 $\sum_{n=1}^{\infty}\frac{1}{n}=\infty$，也就是说，只要乌龟坚持爬下去，它不但能完成环球旅行，也能爬到宇宙的尽头。我们想用这个例子研究一下浮点数的误差问题，并对 float 和 double 类型的情况做些比较。这里先定义两个采用 float 类型的函数：

```
float distf (long n) {
    long i;
    float x = 0.0;
    for (i = 1; i <= n; ++i) x += 1/(float)i;
    return x;
}

long secondsf (float d) {
    long i;
    float x = 0.0;
    for (i = 1; x < d; ++i) x += 1/(float)i;
    return i - 1;
}
```

其中 distf 计算出乌龟 n 秒爬出的距离，而 secondsf 计算出它爬 d 米所用的秒数。注意 secondsf 在返回值时减一，因为循环最后的变量更新已经将 i 多加了一次。

现在就很容易写出完成题目的主函数了，其中有如下函数调用：

```
printf("%ld seconds, %f meters\n", 3600, distf(3600));
printf("%ld seconds, %f meters\n", secondsf(20.0), 20.0);
```

在作者所用的系统上，程序打印出：

```
3600 seconds, 8.766049 meters
```

也就是说，乌龟一小时爬出了 8 米多。但此后程序再也没有输出了，运行了很长时间也不停止。仔细检查程序没有发现错误，修改主函数放入如下语句：

```
for (x = 10.0; x <= 20.5; x += 1.0)
    printf("%ld seconds, %f meters\n", secondsf(x), x);
```

程序输出了下面几行：

```
12367 seconds, 10.000000 meters
33617 seconds, 11.000000 meters
91328 seconds, 12.000000 meters
248695 seconds, 13.000000 meters
662167 seconds, 14.000000 meters
1673859 seconds, 15.000000 meters
```

乌龟4个多小时爬到10米远，19天多才爬到15米，而后程序再也没有反应了。这使我们怀疑问题出在变量的表示范围或者精度方面。我们想到考虑如下的测试函数：

```
void test_float () {
    long i;
    float sum = 0.0, sum0 = -1.0;
    for (i = 1; sum != sum0; ++i) {
        sum0 = sum;
        sum += 1 / (float)i;
    }
    printf ("float: %ld terms at %f\n", i-1, sum);
}
```

循环的结束条件是变量sum的部分和不再变化。在同一系统里，函数很快就输出了一行：

```
float: 2097152 terms at 15.403683
```

这就是说，在加入200多万项之后，以后所加的数值很小的项已经不再起作用了。由此可见，我们对乌龟活动情况的模拟受到了数据表示精度和范围的限制。

为了进一步弄清不同数据类型带来的变化，我们采用 double 类型写出同样程序，结果发现，在20亿秒时和数还在增长，当时的输出值是：

```
double: 2000000000 seconds, 21.993629 meters
```

也就是说，经过大约63年半，乌龟爬出了21米。在大约2亿5千万秒，也就是大约8年的时候爬过了20米，这是题目第二部分的答案。由于我们所用系统中的 long 类型用32位二进制表示，在其范围内无法找到采用 double 类型的增长结束点。可以保证的一点是，这个结束点一定出现在比采用 float 类型大得多的地方。

那么，在精度允许的范围内，采用 float 和 double 类型会产生不同结果吗？我们又写出下面的比较代码：

```
int i;
float sumf = 0.0;
double sumd = 0.0;
for (i = 1; i <= 2000000; ++i) {
    sumf += 1 / (float)i;
    sumd += 1 / (double)i;
}
printf ("float: %d terms at %14.10f\n", i-1, sumf);
printf ("double: %d terms at %14.10f\n", i-1, sumd);
```

得到的结果确实不同：

```
float: 2000000 terms at  15.3110322952
double: 2000000 terms at  15.0858736534
```

精确到小数点后10位的正确值应该是：15.0858736534。到这时为止，采用 double 类型的计算结果在10位精度内还没有产生误差，而用 float 计算已经产生了明显的误差，只剩下两位正确数字了。通过这些试验，我们可以看到多次浮点数运算确实可能带来明显的误差。在解决实际问题时，运算的次数可能远远多于这个小试验，累计误差的情况更复杂。由此可以看到选择适当浮点数的需要。人们建议，如果没有特殊原因，就应选用 double 类型。float 表示范围和精度不能满足许多常规浮点运算的需要，而 long double 类型有可能影响程序效率。它们应该只用于特殊场合。

上面的试验展示了浮点数运算的误差问题。在某些特殊情况下，浮点数运算的误差积累

还可能更迅速得多。有两个情况值得特别提出：1）将一批较小的数一个个加到很大的数上，常常会导致丢掉小的数的重要部分，甚至导致小数整个被丢掉（其实，上例中的情况就是这样）。2）两个值相当接近的数相减，也可能导致结果的精度大幅度下降。

4.1.5 求立方根（迭代和逼进）

问题：已知求 x 立方根的迭代公式（递推公式）是：$x_{n+1} = \frac{1}{3}(2x_n + x/x_n^2)$，写一个函数，利用这个公式求 x 的立方根的近似值，要求达到精度 $\left|(x_{n+1} - x_n)/x_n\right| < 10^{-6}$。

给所定义函数取名为 cbrt（仿照 sqrt），其类型特征为：

```
double cbrt(double x)
```

从公式中看到，迭代中每次求下一迭代值时都要用参数 x，所以这个值需要保留。在这个例子里，事先也没有办法确定循环执行的次数，只能按题目要求给出循环结束条件，期望在这一条件能在某个时刻被满足（该条件最终将被满足是由人们对这个计算方法的研究保证的）。按照定义，判断迭代终止时要用到前后两个近似值，因此这两个值必须用两个变量保存。根据这些分析，这个函数的体可以如下实现：

```
double x1, x2;
x1 = x;
x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;
while (fabs((x2 - x1) / x1) >= 1E-6) {
    x1 = x2;
    x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;
}
return x2;
```

读者应该记得，这里的 fabs 是<math.h>里的标准数学函数，它求出参数的绝对值。这段代码有一个问题：如果初始时参数 x 值就是 0，函数执行时会发生以 0 做除数的错误。这个问题需要在函数执行的开始位置处理，加上一句：

```
if (x == 0.0) return 0.0;
```

这样写出的函数定义如下：

```
double cbrt (double x) {
    double x1, x2;
    if (x == 0.0) return 0.0;
    x1 = x;
    x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;
    while (fabs((x2 - x1) / x1) >= 1E-6) {
        x1 = x2;
        x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;
    }
    return x2;
}
```

这个函数定义直截了当，但其中有两个相同语句，似乎不够简约。采用 C 语言的其他控制语句可以消除这类重复，本章后面有相应程序示例。仅采用 while 或 for 语句也可以消除这类重复，但需要利用逗号表达式，写出的程序也不那么清晰，这里就不讨论了。

这个函数也很典型。人们研究了许多典型函数的计算方法，许多函数的计算都采用类似本函数计算方式的公式，其中需要通过一系列迭代计算，取得一系列逐渐逼进实际函数值的近似值。实现这类计算的程序通常都具有上述函数的形式：采用几个互相协作的临时性变量，通过它们值的相互配合，最终算出所需要的函数值。

4.1.6 求 sin 函数值（通项计算）

问题：定义一个函数，利用公式 $\sin x = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$ 求 $\sin x$ 的近似值。

假设要定义的函数是 double dsin(double x)（为避免与标准库函数冲突，这里另选了一个名字）。显然在计算过程中需要将各项的值不断加进来，在 n 趋向无穷时，项的值将逐渐趋向于 0。为写出这个程序，需要给近似值概念一个精确定义。例如，采用项的值小于 10^{-6} 作为结束条件，在这时结束循环，以得到的累积值作为近似值。

在这一循环中，显然需要一个保存累积和的变量，假定选用 sum；每次循环将求出一个项的值，用一个变量 t 保存这个值。下一个问题是如何算出 t 的值。直截了当的方式是每次都按通项公式 $(-1)^n \frac{x^{2n+1}}{(2n+1)!}$ 计算，有关的计算并不难写出：

```
for (t = 0.0, i = 1; i <= 2*n+1; ++i)
    t *= x/i;
```

仔细分析不难看出，这一做法将形成许多重复的计算，因为后一项的大部分计算在算前一个项时都已经做过。如果记录当前项的值，就可以很容易算出下一个项的值。也就是说，项的值可以一个个向前推。不难发现，从一个项算出下一个项的递推公式是：

$$t_n = -t_{n-1} \cdot x^2 / (2n \cdot (2n+1))$$

找到这个公式，计算各项的事情就很简单了。

整个循环的初始值应是：sum = 0.0，n = 0，t = x。现在已经很容易给出如下的函数定义：

```
double dsin (double x) {
    double sum = 0.0, t = x;
    int n = 0;
    while (t >= 1E-6 || t <= -1E-6) {
        sum = sum + t;
        n = n + 1;
        t = -t * x * x / (2*n) / (2*n + 1);
    }
    return sum;
}
```

通过分析发现了项的递推性质，在这个函数定义里节省了许多计算。假设需要算 m 个项，采用上面方法，计算中各种基本运算的次数与 m 的值成正比；如果用分别计算各项的方式，总的计算量将与 m^2 成正比。如果项数很多，两种不同方式效率差异将很明显。

还有一个情况值得提出：这个函数中循环体的执行次数依赖于参数情况，甚至很难做出近似估计。根据数学知识，当实参绝对值较小时（例如，$x \in [-\pi, \pi]$ 时），级数收敛很快，项的绝对值将迅速减小，使函数很快完成。如果实参绝对值很大，循环就可能做许多次，甚至一直做到 n 的值超出整数表示范围（n 值超出整数表示范围会发生什么事？请读者考虑）。一个可能改进是利用标准库函数 fmod，将参数值对 π 值取余数后再计算。

从这个例子和讨论可以看出，对于问题本身的理解很重要。对级数收敛性质的考虑能帮助我们认识许多情况。所以说，写程序时必须仔细考虑问题本身的性质。

这个函数也很典型。许多程序里需要计算一系列项的值，此时，寻找项值的共性就尤为重要了，因为只有这样，我们才能将计算这些项值的工作写成一个循环。如果顺序各项的值有某种递推关系，我们就应该利用它减少计算量，写出效率更高的程序来。

4.2 循环程序的问题

4.2.1 从循环中退出

有时我们可能希望从正在执行的循环中间退出来，前面判断素数的函数里就遇到了这个问题。在那里，找到了一个因子就可以做出结论了，当时是用一个 return 语句退出函数，

同时也结束了当时的循环。下面看另一个例子，说明一种常见的处理方式。

考虑写程序验证著名的哥德巴赫猜想，对 6 到 200 间各偶数找出素数分解，即找出两个素数，使它们的和等于这个偶数。利用前面的 isprime，程序主要部分可以写成：

```
for (n = 6; n <= 200; n += 2)
    for (m = 3; m <= n/2; m += 2)
        if (isprime(m) && isprime(n-m))
            printf("%d = %d + %d\n", n, m, n-m);
```

这里出现了一个问题：当某个偶数能以多种方式表示为两个素数之和时，上述程序段在执行时就会输出多对分解。例如对偶数 10，程序会输出两行：

```
10 = 3 + 7
10 = 5 + 5
```

为了验证哥德巴赫猜想，对每个偶数只需找到一种分解。找到更多分解既无实际意义，又浪费了计算时间。现在希望对每个偶数只输出一行，考虑应该如何处理。

上面程序段由两层循环组成。内层循环里的 m 每次从 3 开始递增取值，直到大于 n/2。如果在此范围中遇到多个满足条件的素数对，程序就会产生多行输出。问题是怎样在发现了一对素数后使循环停下来。这实际上要求增加对循环过程的控制。对于这类情况，人们常用的方法是把发现一对素数分解这件事也作为一个条件，加入循环控制条件。一种可行方法是引进一个新变量，例如取名 found，其值是 0 表示尚未发现满足条件的素数。一旦发现了这样的素数就为它赋值 1。为实现这一方案，内层循环初始化时应把 found 置 0，表示尚未找到分解；循环里发现了所要的素数时将 found 置 1。此外，内层循环的条件也需要修改：

```
for (n = 6; n <= 200; n += 2)
    for (found = 0, m = 3; m <= n/2 && !found; m += 2)
        if (isprime(m) && isprime(n - m)) {
            printf("%d = %d + %d\n", n, m, n-m);
            found = 1;
        }
```

代码里的!是求否定运算符，在 found 为 0 时!found 得到 1，表示条件成立。一旦发现了素数对，由于 found 被置为 1，循环条件中的 !found 不再成立了。这样，内层循环退出，程序开始下一次大循环，开始对下一偶数的处理。

这里的 found 是个专用于控制循环的辅助变量，这类作法在程序设计中也很常见。从循环中间退出是写程序时常要做的事情，C 语言（其他许多语言也一样）为此提供了专门的机制，这就是下面介绍的 break 语句。

break 语句

break 语句在形式上就是：

```
break;
```

它只能用在循环语句以及在后面将要介绍的 switch 语句里，其作用是使当前的（最内层的，因为循环等可能出现嵌套）循环语句（或 switch 语句）立刻终止，使程序从被终止的循环（或 switch）语句之后继续执行下去。break 语句实际上就是为了解决前面提出的一个问题，使人能方便地描述从循环执行中退出的动作。通常应把 break 语句放在条件语句控制之下，以便在某些条件成立时立即结束循环。

现在仍用前面有关从循环中退出的例子来说明 break 语句的使用。完成同样工作的程序段可以写成下面样子。这段代码更简单，不必引进循环控制变量。一旦条件成立（找到了所需要的一对素数），打印输出后执行 break 语句，就会导致内层循环结束。程序的执行回到外层循环的最后：更新变量 n，判断条件，并继续工作下去：

```
for (n = 6; n <= 200; n += 2)
    for (m = 3; m <= n/2; m+= 2)
        if (isprime(m) && isprime(n - m)) {
            printf("%d = %d + %d\n", n, m, n-m);
            break;
        }
```

下面是借助 break 语句重写的求立方根函数：

```
double cbrt (double x) {
    double x1, x2 = x;
    if (x == 0.0) return 0.0;
    while (1) {
        x1 = x2;
        x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;
        if (fabs((x2 - x1) / x1) < 1E-6) break;
    }
    return x2;
}
```

请读者特别注意函数中的循环结束条件。

4.2.2 循环中的几种变量

循环中有几种常见的变量，它们在循环中的使用方式非常典型，了解这些情况有助于对重复计算和循环的分析和思考。这种分类和分析也是人们写循环程序的经验总结。应当注意，这里提出的种类不是绝对的，不同种类之间常常也没有截然的界限。

1. 循环控制变量（简称循环变量）。这种变量在循环开始之前设初值，每次循环时递增（或递减）一个固定值，直到它的值达到（或超过）某个界限时循环结束。这种变量控制着循环的进行和结束，使之成为具有固定次数的循环。在最典型的 `for` 循环中常常能看到这种变量。下面三个例子中的 `n` 就是典型的循环控制变量：

   ```
   for (n = 0; n < 10; ++n)
       ... ...
   for (n = 30; n >= 0; --n)
       ... ...
   for (n = 2; n < 52; n += 4)
       ... ...
   ```

2. 累积变量。这类变量在每次循环执行中被更新，其更新常用诸如 `+=` 或 `*=` 一类运算符，而循环之前变量的初值常用相应运算符的单位元素（例如，采用加法更新的变量用 0 作为初值；采用乘法更新的变量用 1 作为初值；等等）。循环结束时，累积变量中将积累下来一个最终值，这种最终值常被作为循环计算的最终结果。

3. 递推变量。这种情况更一般，循环变量或累积变量都可以看作特殊的递推变量。递推变量常指在循环中互相协调工作的多个变量，它们亦步亦趋，每次循环通过其中一个或几个算出另一个的新值，然后依次更新各变量（后面变量取前面变量的值，推进一步）。例如，在三个递推变量 `x1`、`x2`、`x3` 的循环体里，常可以看见下面形式的语句序列：

   ```
   x1 = x2;
   x2 = x3;
   x3 = ... x1 ... x2 ...;
   ```

   我们可以形象地把 `x3` 看成是“走在前面”的一个，而 `x2` 和 `x1` 依次紧随其后，三个变量亦步亦趋地更新。

在下面程序实例里，读者可以看到许多变量能归于三类的这一类或者那一类，也会有些比较难清楚归类的变量。当然，这里的讨论只是为了给读者提供一些认识问题的线索，而不是想成为一种教条。

4.3 循环与递归

我们已经看到，一段短程序中采用了循环，就可能导致一段很长的计算，完成很复杂的工作。如果告诉读者，即使没有上面讨论的控制结构，用它们能写的东西就已经可以写了，读者可能不信。实际上，C 语言允许采用一种称为递归的写程序方式，采用这种方式，不必循环结构就可以写出复杂的程序来，写出的许多程序还特别简单清晰。下面介绍这种编程方

式，并与循环结构做一些比较。其中还要讨论另外一些重要问题。

4.3.1 阶乘和乘幂（循环，递归）

先考虑一个最简单的例子。假设现在要定义一个计算整数的阶乘的函数：

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

这不能通过平铺直叙的方式实现，因为要做的乘法次数依赖于实参值，函数定义时无法确定这个值，而且每次调用函数所用的实参也可能不同。按照已知的技术，我们可以用一个循环解决问题。阶乘函数的类型特征可确定为：`int fact(int)`。由于阶乘值随参数的增长而增长的速度非常快，可能更合适的选择是 `long fact(long)`,:

```
long fact1 (long n) {
    long fac, i;
    for (fac = 1, i = 1; i < n; ++i)
        fac *= i;
    return fac;
}
```

循环用了两个变量，其中的 `fac` 保存部分阶乘的值，其最终值就是所需的阶乘值。

上面给出的阶乘的数学定义中用到省略号，在数学里经常看到这种写法。实际上这种写法并不科学，是数学表述中常见的不精确性。因为它的意义依赖于人对省略号的理解和共识。要定义清楚阶乘的概念，就需要采用*递归定义方式*。阶乘的递归定义可写为：

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

就是说：如果 $n = 0$，那么其阶乘是 1；当n值大于 0 时，其阶乘等于$n-1$的阶乘的n倍。在这个定义的右边用到被定义的东西（阶乘），这就是*递归*一词的含义。那么这种做法是否会造成不合理的循环定义呢？实际上不会。因为定义中对特殊情况（$n = 0$）直接给出了确定的值；而对一般情况，则是将一个数的阶乘归结到比它小 1 的数的阶乘定义。按照定义，任何大于 0 的整数n的阶乘由$n-1$的阶乘定义，而$n-1$的阶乘又由$n-2$的阶乘定义，……。因此n的阶乘将最终归结到 0 的阶乘，而这已经明确给出了。这样，任何正整数的阶乘都由这个定义确定了。本论题的严格证明需要用数学归纳法。

从计算的角度看，上面的递归定义也提供了一种计算阶乘函数的方法。如果所用的描述工具（程序语言）支持用递归的方式定义计算过程，上述定义就可以直接翻译成程序。C 语言允许用递归方式定义函数，也就是说，它允许在被定义的函数体内部调用该函数自身。采用递归定义方式，定义阶乘函数变成了一件极简单的事情：

```
long fact (long n) {
    return n == 0 ? 1 : n * fact(n-1);
}
```

这一定义直接对应于阶乘函数的递归形式的数学定义，其正确性很明显。如果要计算 10 的阶乘，并将结果存入长整型变量 n，就可以直接写：

```
n = fact(10);
```

虽然上面阶乘函数的定义很简单，它所实现的计算过程却并不简单。图 4.1 形象地展示了 `fact(3)` 的计算过程。在求 `fact(3)` 时需要进一步调用 `fact(2)`，进而又调用 `fact(1)`，直到达到 `fact(0)`。`fact(0)` 直接返回结果。此后，前面的一系列调用也将顺序地得到结果值

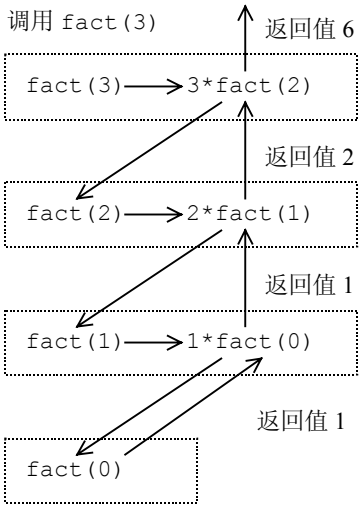

图 4.1　`fact(3)` 的计算过程

而返回。最终使 fact(3)求出了结果 6。图 4.1 中的箭头描述了函数调用与返回的流程。

若考虑 long 类型的表示范围，函数里还应增加对负参数值的处理。一个合理方式是扩充阶乘定义，令其对所有负数都给出值 1。若采纳这一定义，只需把程序里的关系运算符==换成<=运算符。如果需要，也可以考虑增加对参数的检查，在遇到负数时报告一个错误。

例：写出函数 double dexp(int n)，它求出 e（自然对数的底）的 n 次幂*。

在参数非负的情况下，可以写出如下递归形式的函数定义：

$$e^n = \begin{cases} 1 & n = 0 \\ e \times e^{n-1} & n > 0 \end{cases}$$

这可以直接实现为递归定义的函数。参数为负时原来的函数也有定义，因此需要考虑对负数的处理。显然，当 $n < 0$ 时有 $e^n = 1/e^{-n}$，而且 $-n > 0$。我们先可以定义一个处理非负参数的辅助计算函数（用递归方式），而后借助它定义所需要的 dexp：

```
double dexp1 (int n) {
    return n == 0 ? 1 : 2.71828 * dexp1(n-1);
}

double dexp (int n) {
    return n >= 0 ? depx1(n) : 1 / dexp1(-n);
}
```

dexp1 是为了实现 dexp 而定义的辅助函数，它实现计算过程的主要部分，但我们并不打算直接用它，因为它不完全。主要函数 dexp 只是根据不同情况确定如何使用 dexp1。在实际程序设计中，如果要实现的功能比较复杂，常常需要先定义好一个或一批辅助函数。确定辅助函数所需的功能并给出定义，对于写好程序常常是非常重要的。

这个函数同样可以用循环的方式写出。下面是一种定义：

```
double dexp1 (int n) {
    double x = 2.71828, d = 1;
    int i;
    if (n < 0) {
        n = -n;
        x = 1/x;
    }
    for (i = 0; i < n, ++i) d *= x;
    return d;
}
```

当 n 的值小于 0 时，循环中乘起 |n| 个 1/e，仍能得到正确结果。

上面两个例子说明，一些循环程序可以用递归的形式写出，一些递归程序也可以通过循环写出。当然，要写出一个递归的程序，就必须定义函数并在这个函数的体内调用这一函数本身，递归调用时需要用到函数的名字。

从形式看，上面的递归函数定义都用到条件表达式（或者条件语句），这是必需的，因为这类函数的定义中需要区分两类情况：一类是可以直接给出结果的情况，是递归定义的基础；另一类是需要递归的情况。在对后一类情况的处理中，总设法把复杂情况的计算归结到对较为简单情况的计算。这些都是递归定义的实质，后面还会进一步讨论有关的问题。

递归和循环

有了基本运算符、关系判断和条件表达式，加上可递归的函数定义，就有了一个表达能

* 在叙述函数的类型特征时给参数定名字，是为了在其他地方（如正文里）较方便地讨论与该参数有关的问题。后面有时采用这种写法。在其他 C 语言书籍，C 语言手册里也常用这种写法。

力很强的程序语言。理论上说，如果用其他任何程序设计语言能写出某个函数，只用上面这些机制也能写出这个函数来。这个程序语言的能力“足够强”。

那么，既然 C 语言的一小部分功能已经“足够强”了，为什么它还要提供其他功能呢？原因是实际的需要。从实际程序设计的角度，语言里还需要其他的许多有用机制，这些机制使我们可以更好、更自然、更方便地写出所需程序。

一些读者可能希望进一步了解递归和循环的关系。从理论上说，每段采用循环结构写出的程序都可以机械地改写为一个不用循环，只用递归的函数。反过来则不那么容易。要想将通过递归方式写出地程序改为循环程序，则常常需要复杂的智力劳动，常常需要借助于高级的程序设计技术。具体情况这里就不讨论了。

4.3.2 斐波那契序列（计算与时间）

数学里非常重要的 Fibonacci（斐波那契）序列 $\{F_i\}_i$ 有如下的递归定义：

$$F_0 = 1, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (n > 1)$$

很容易用递归方式给出求 F_n 的函数，可直接写出下面定义：

```
long fib (int n) {
    return n < 2 ? 1 : fib(n-1) + fib(n-2);
}
```

函数定义中考虑了实参可能为负的情况，把负数编号的序列值都硬性定义为 1。这是一种合理处置方法，写程序中常常遇到这类情况。由于 Fibonacci 序列的值增长非常快，这里用长整数类型作为函数的返回值。

问题分析

现在要讨论的问题是：上面这个函数好不好？从一方面看，这个程序确实很好，其描述方式（递归定义的函数）与 Fibonacci 序列的数学定义直接对应，很容易确定函数定义的正确性。函数的定义也很简单，容易读，容易理解。

但从另一方面看，这个定义却有一个本质性的缺陷。为说明它的缺陷，我们先看一个计算实例，看 `fib(5)` 的计算情况。`fib(5)` 的计算过程可以用图 4.2 示意。

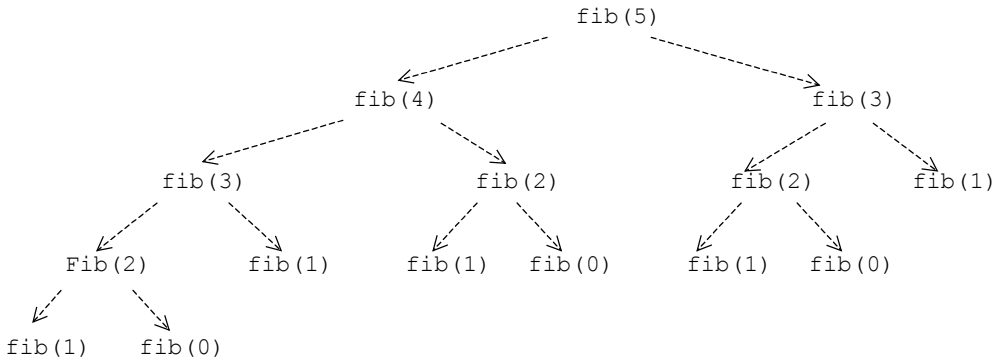

图 4.2　`fib(5)` 计算中的函数调用情况

在图 4.1 里，从 `fib(5)` 向下到 `fib(4)` 和 `fib(3)` 有连线，这表示在计算 `fib(5)` 时需要调用 `fib(4)` 和 `fib(3)`，以完成 `fib(5)` 的计算，其他连线也一样。从这个图里可以看到执行中的调用关系，统计出函数调用的次数，也可以看到许多计算有重复。还有一个情况也值得注意：参数越小，调用重复次数也越多（除 `fib(0)` 没有 `fib(1)` 多之外），例如 `fib(2)` 算了 3 次，`fib(1)` 算了 5 次。这种情况具有普遍性，以其他参数开始计算的情况

也类似。而且，启动计算所用的参数值越大，重复计算就出现得越多。

读者可能认为，有些重复计算也没关系，今天的计算机这么快，多算几次花不了多少时间。但问题不那么简单。随着参数值的增大，fib 计算中的重复工作将飞速增长，如果在计算机上试试 fib(35)、fib(36)、……，立刻就能体会到这种情况。据推算，在目前最快的微机上，算出 fib(45)的值大约需要一分钟，而 45 确实不是一个很大的值。

更严重的问题是，参数值增加 1，函数 fib 的计算时间将为原来的 1.6 倍左右（这是理论估计，实际时间可能更多）。这样，在目前最快的微机上，一个小时也算不出 fib(55)，而计算 fib(100)将需要数万年（数量级估计）。这个结果实在很令人吃惊！

这个例子反映出计算机的一个本质弱点：计算需要花时间，复杂的计算要花很多时间。这个本质特征就说明了存在着计算机将无法帮我们做好的事情。

求 Fibonacci 值的问题可以找到更好的办法（下面介绍），但有些问题不是这样。人们已发现了许多实际问题，从理论上说可以用计算机解决，因为可以写出解决问题的程序。但是这并没有从实际的角度解决问题，因为对规模稍大的实际情况（“规模”大对应于上面“大一点的参数 n”），我们的千万代子孙也等不到计算完成。这种情况下能说问题解决了吗？显然说“能”是很牵强的，理解这一点对于理解计算机也是非常重要的。

4.3.3 为计算过程计时

统计一个程序或程序片段的计算时间，能帮助我们了解程序的性质，编程序的人们常需要做这件事。为程序计时可以用普通计时器，如手表或秒表。但那样做既费事又不精确。许多程序语言或系统都提供了内部的计时功能，下面介绍 C 标准库的计时功能。

C 标准库里有几个与时间有关的函数，它们在标准头文件<time.h>里说明。如果要在程序中统计时间，程序的头部应当写：

```
#include <time.h>
```

做程序计时通常写表达式：

```
clock() / CLOCKS_PER_SEC
```

这是调用库函数 clock 的值除以标识符 CLOCKS_PER_SEC 代表的常量。这个表达式将算出一个数，表示从程序开始执行到这个表达式求值的时刻所经历的时间，以秒数计*。

作为例子，假设想确定在所用计算机上，计算一些 Fibonacci 序列值需要的时间，可以写出下面程序：

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

long fib (int n) {
    return n <= 1 ? 1 : fib(n-1) + fib(n-2);
}

int main () {
    double x;
    int n;

    for (n = 35; n < 46; ++n) {
        x = clock();
        printf("Fab %d is %ld in ", n, fib(n));
        printf(" %fs\n", (clock()-x)/CLOCKS_PER_SEC);
    }
    return 0;
}
```

* 有些老的 C 语言系统用其他常量名。例如 Turbe-C 用的是 CLK_TCK 而不是 CLOCKS_PER_SEC。如果按照这里的书写程序出现问题，请读者查一查所用程序语言系统的说明。

这里需要说明几点：首先，程序输出的是两次计时之差，它基本上等于计算 fib 所用时间。另外，每个C系统有一个最小时间单位，小于这个时间单位的计时值都是0，计时结果也是以这个时间单位步进增长的。所以得到的计时值只能作为参考。

为程序或程序中某部分计时，是人们在实现大的复杂系统、研究具体计算过程的性质等工作中经常做的事情。在程序工作中，如果发现某个程序很慢，可能做的工作一方面是仔细检查程序，估计其中各部分耗费的时间（该部分的“复杂程度”），还可以给某些部分计时，考察其实际情况。然后考虑设计或实现方法的改进。计算代价的理论研究形成了称为“算法分析和复杂性”的研究领域，该领域的研究工作已经取得了许多极其重要的成果，例如，数学家提出的目前排行第一的“最重要数学问题”就源自这一领域。

4.3.4 Fibonacci序列的迭代计算（程序正确性与循环不变式）

求Fibonacci数的问题确实有“更好”（更快）的解法。考虑下面论述：

（1）F_0 和 F_1 已知；

（2）由 F_0 和 F_1 可以计算出 F_2；

（3）一般地，若已知连续两个Fibonacci数，就可以算出序列中下一个数。

这也说明了计算第 n 个Fibonacci数的一种方式：可以从F_0 和F_1 出发，向前逐个推算，直至算出所需的F_n 为止。显然计算中需要一个计数变量，保存当时算出的Fibonacci数的下标。按这种想法写出的一个新函数定义如下：

```
long fib1 (int n) {
    long f1 = 1, f2 = 1, f3, i;
    if (n <= 1) return 1;
    for (f3 = f1 + f2, i = 2; i < n; ++i) {
        f1 = f2;
        f2 = f3;
        f3 = f1 + f2;
    }
    return f3;
}
```

函数里的循环比较复杂。要写好这个循环，必须想清需要做什么。循环的最终目的是：一旦变量 i 增加到等于参数 n，变量 f3 应当正好是所需的F_n值。为做到这点，就要求每次循环判断时，都保证 f3 的值正好是第 i 个Fibonacci数。不难看清，上面循环确实保证了这种关系。循环中使用了三个递推变量，其更新顺序和方式正好是前面介绍过的。

一个循环表示一种反复进行的计算过程，根据实际执行中所涉及的计算对象，循环体可能执行多次。在这种情况下，怎样保证对各种可能数据，某个循环都能正确完成计算呢？显然，循环里总要涉及一些每次循环都变化的变量。但仔细分析可以发现，写循环时实际上总要考虑一些变量间的某种内在关系，设法保证这些关系在循环体执行中保持不变。这类关系被称作*循环的不变式*。

前面写简单循环时，我们更多地依靠直觉，例如，对于读者已经很熟悉的循环：

```
for (sum = 0, n = 1; n <= 100; ++n)
    sum = sum + n * n;
```

所维持的不变关系就是：“在每次判断循环条件时，sum 里保存的总是前 n-1 个正整数的平方和”，这里的 n 指当时变量 n 的值。显然，在刚进入循环时 n 为 1，sum 为 0，满足上述关系；每次循环将 n 的平方加入 sum 并将 n 值加 1，这就不会破坏上述条件。在循环结束时 n<=100 不成立了，也就是说 n 变成了 101。有了这些分析后，我们可以知道，这时 sum 的值就是前 100 个数的平方和。

实际上，要写好一个循环，最重要就是弄清楚“循环中应当维持的什么东西不变”。应当想想在循环中需要维持变量间的什么关系，才能保证当循环结束时，各有关变量都能处在

所需的状态。写完循环后还要仔细检查，看提出的要求是否满足了。对于简单循环，这些事情可能很自然。在处理复杂的循环时，有意识地思考这个问题就更重要了。在上面 Fibonacci 函数的循环里，保持的就是 f3 与 i 之间的关系：保证每次循环判断时 f3 的值正好是第 i 个 Fibonacci 数的值。有了这一点，函数能计算出正确结果也就确信无疑了。

下面问题是：这个函数确实比前一个好吗？

首先应当看到，新函数定义比前一个定义复杂很多，其中不但区分了各种情况，还出现了一个复杂的循环。弄清程序能否完成所需计算不再那么简单了，要经过仔细思考，还需要借助于循环不变式一类概念。从这方面说，新函数远不如前面递归定义的函数。

但在计算时间上，新函数却有很大优越性。这个函数的主体就是一个循环，所需时间基本上由循环次数确定，而循环体的执行次数基本上等于参数 n 的大小（当 n 大于 0 时），计算 fib(100)的值不过大约 100 次循环，在计算机上根本察觉不到运行的时间。与前面函数完成同样计算需要的时间以万年计相比，新的函数显然具有绝对的优越性（这里没有考虑表示范围，long 类型通常无法表示 fib(100)的值）。

最后还要提醒读者：不是所有问题都像计算 Fibonacci 数这样简单，上面迭代函数可以看着消除原递归函数的重复计算的结果。有些问题的计算确实需要很长时间。人们已经发现了许多很有实际价值的问题，对于这些问题，已经找到的计算方法都非常耗时。但是对于这些问题到底有没有更快的计算方法至今也没有结论。寻找完成计算的好算法一直是计算机科学及其应用中一个非常重要的研究领域。

4.3.4 最大公约数

写出函数 long gcd(long, long)，它有两个长整数参数，求出这两个数的最大公约数（greatest common divisor，gcd）。存在着完成这一工作的许多方法：

解法 1

最直接的方法是检查一些整数，直到找到能同时整除两个参数的最大数。设给定参数分别是 m 和 n，为取一些数检查，就需要一个辅助变量 k，以它的值为各次检查所用。下面问题是 k 如何取值。一种简单办法是让 k 顺序取值，用一个循环实现重复试验。这样，k 的取值问题就变成了：如何取初值，如何更新，如何判断结束。

可能方法 1：令 k 从 1 开始递增，结束条件是 k 大于 m 或 n 中的某一个，因为最大公约数绝不会大于两个数中的任何一个。

下一问题是如何得到所需结果。m 和 n 可能有许多公约数，变量 k 的最后值也不会是 m 和 n 的公约数，因为它已大于两个数之一，这样就需要记录循环过程中找到的公约数，需要为此引进新的变量。在这个具体问题里，我们只需一个变量记录已找到的最大的那个公约数，下面用变量 d。初值设为 1，因为 1 是任何数的约数，一旦遇到 m 和 n 的新公约数（显然它大于 d），就把它记入 d 中。这样，更新 d 的条件可以写为：

```
if (m % k == 0 && n % k == 0)
    d = k;  /* k为新找到的一个公约数 */
```

确定了 d 及其初值，不难看出令 k 从 2 开始循环就可以了。函数的定义已经很清楚了：

```
long gcd (long m, long n) {
    long d = 1, k = 2;
    for ( ; k <= m && k <= n; ++k)
        if (m % k == 0 && n % k == 0)
            d = k;
    return d;
}
```

如果 m 和 n 没有公约数，变量 d 的初值 1 就会留下来，也能给出正确结果。

如果考虑参数的各种可能情况，就会发现还有许多特殊情况需要处理：

1. 如果 m 和 n 都为 0，此时不存在最大公约数。对这种情况要确定一种处理方法，例如，可以令函数返回值 0，指明遇到了这种特殊情况；
2. 如果 m、n 之一为 0，最大公约数就是另一个数。上面函数在遇到这种情况时返回值不对。这些可以直接判断和处理；
3. m、n 可能是负数，此时上面函数返回 1，可能是错的，也应当处理。

综合起来，在上面函数的循环之前应加上下面几个语句：

```
if (m == 0 && n == 0) return 0;
if (m < 0) m = -m;
if (n < 0) n = -n;
if (m == 0) return n;
if (n == 0) return m;
```

这个示例程序里用到一个辅助变量 d，记录计算过程中的中间值。引入这类辅助变量是程序设计中一种常用的技术。另一个值得注意的问题是，许多计算问题都有一些需要处理的特殊情况，应当仔细分析，分别处理。需要做的一件事就是分析参数的各种可能情况，看看程序是否都能给出正确结果，必要时设法改造程序，常常是在程序前面加一些条件判断，也可能需要另外的处理方法。

可能处理方法 2：令 k 从某个大数开始递减取值，这样找到的第一个公约数就是最大公约数。这时可以将 k 的初值取为 m 和 n 中较小的一个，在没遇到公约数的情况下递减。结束条件是：或者 k 值已经达到 1，或者找到了公约数。由于 1 总是两个数的公约数，因此，前一个条件被第二个条件覆盖。这样就可以写出下面的循环体：

```
for (k = (m > n ? n : m);
     m % k != 0 || n % k != 0; --k)
    ;      /* 空循环体 */
return k; /* 循环结束时，k总是最大公约数 */
```

采用这种方法写出的程序更简单些。

上面两个解法的共同点是采用重复测试方法，用一个个的数去检查条件成立与否。这种方法的缺点是效率较低，如果被检查的数比较大，循环就需要反复执行许多次。一般说，如果能够找到解决问题的特殊方法，就不应该选用这种“通用”方法。

解法 2

对于求最大公约数的问题，古代的人们已经提出了更有效的计算方法，这就是很著名的欧几里德算法（欧氏算法），即*辗转相除法*。下面介绍欧几里德算法的程序实现。首先，不难用递归形式给出辗转相除法求最大公约数的定义：

$$\gcd(m,n)=\begin{cases} n & m \bmod n = 0 \\ \gcd(n, m \bmod n) & m \bmod n \neq 0 \end{cases}$$

函数定义 1（递归方式）：先假定函数的第二个参数非 0，而且两个参数都不小于 0。下面函数直接对应于辗转相除法的递归定义：

```
long gcd1 (long m, long n) {
    return m % n == 0 ? n : gcd1(n, m % n);
}
```

这个函数很简单，数学中对欧几里德算法的研究保证了这个程序的计算能结束。这个函数的计算速度很快，一般来说远远快于顺序检查。

有关各种特殊情况的处理可以另写一个函数，其主体是对上面函数的调用，这是程序设计中常用的一种方法。下面是求最大公约数的主函数：

```
long gcd(long m, long n) {
    if (m < 0) m = -m;
```

```
    if (n < 0) n = -n;
    return n == 0 ? m : gcd1(m, n);
}
```

函数定义 2（循环方式）：辗转相除也是一种重复性工作，做的是反复地求余数，这种计算也可以通过循环实现。在写出程序前，先分析一下有关循环应当如何写：

1) 循环开始时 m 和 n 是求最大公约数的出发点；
2) 每次循环判断 m % n 的值是否为 0。如果为 0，那么 n 就是最大公约数，计算结束；
3) 否则做变量更新：更新后的 m 取原来 n 的值，而 n 取原来 m % n 的值。为保证更新操作正确，必须另用一个辅助变量 r，下面是正确的更新操作序列：

```
r = m % n; m = n; n = r;
```

这样，循环就可以写为：

```
for (r = m % n; r != 0; r = m % n) {
    m = n;
    n = r;
}
```

C 语言里的函数参数可直接作为局部变量用（参数实际上就是局部变量）。下面是一个函数定义，其中假定参数值均不小于 0。增加对负数处理的工作请读者完成。

```
long gcd2 (long m, long n) {
    long r;
    if (n == 0) return m;
    for (r = m % n; r != 0; r = m % n) {
        m = n;
        n = r;
    }
    return n;
}
```

m 的值为 0 以及 m 和 n 的值都为 0 没有作为特殊情况，因为这些处理已经包含在程序里，读者很容易弄清楚。这个程序里也有一个较复杂的循环。按前面说法，循环中总需要维持某一种不变关系，以保证循环的正确性。上面这个循环中保持的是变量间一种什么关系呢？这一点也请读者考虑。

4.3.5 河内塔（梵塔）问题

这里讨论一个用递归解决很简单，但不容易用循环程序解决的问题：河内（hanoi）塔问题。问题来自古印度。某神庙里有三根细圆柱，共有 64 个大小不等、中心有圆孔的金盘套在柱上，这就是梵塔。僧侣们日夜不息地将圆盘从一根柱移到另一根，规则是每次只能移一个圆盘，而且不允许将大盘放到小盘上。开始时所有圆盘都按从大到小顺序套在一根柱上，据说当所有圆盘都搬到另一根柱时，世界就要毁灭。图 4.3 的细柱上放着 5 片圆盘。

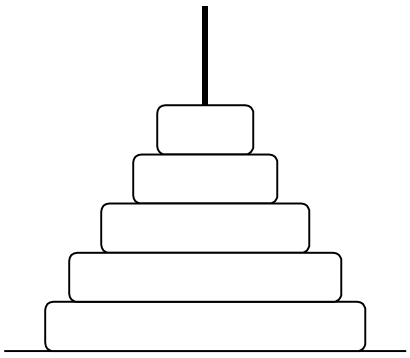

图 4.3 河内塔（梵塔）

现在考虑如何写一个程序来模拟搬圆盘的过程。当然，这一程序不能真正去搬动圆盘，我们希望它能打印出一系列搬动指令，说明应该将圆盘从哪根圆柱搬到哪根圆柱。为了方便起见，现在分别将三根圆柱命名为 a、b 和 c，假定开始时所有圆盘都在 a 圆柱上，而目标是将它们都搬到 b 圆柱，而且要保持从大到小的顺序。

初看起来，这个问题好像没什么规律。首先只能将最小的圆盘搬到另一圆柱，例如圆柱 b。然而这时已经不能再向圆柱 b 搬圆盘了，因为将任何圆盘搬过去，都将违反“不能把大圆盘放到小圆盘上”的规则。这时只能考虑将圆柱 a 上的次小圆盘搬到圆柱 c。下一步只能

将柱 b 上那个最小的圆盘搬到柱 c 的次小圆盘上面。这样考虑问题，事情显得很杂乱无章，实在看不出写出程序的线索。

对于这个问题，突破的要点在于换一种看法，反过来看问题：假设能将上面的 63 个圆盘从圆柱 a 搬到圆柱 c（其间可以利用圆柱 b 作为辅助位置），那么，下一步就可以将柱 a 上最后一个圆盘（最大圆盘）直接搬到 b，剩下的问题就是将 c 柱上的 63 个圆盘搬到 b 里。这样分析就抓住了问题的基本结构：实际上，可以把搬动 64 个圆盘（借助于另一个圆柱）的问题分解为三个子问题：

1. 借助于圆柱 b 将 63 个圆盘从 a 搬到 c；
2. 从 a 搬一个圆盘到 b；
3. 借助于圆柱 a 将 63 个圆盘从 c 搬到 b。

应该看到，搬 63 个圆盘的问题与搬 64 个圆盘完全一样，只是问题的规模减小了一点。从这一分析中，我们已经看到了采用递归方法解决问题的所有特征：这里有一种基本情况：如果某圆柱上只有一个需要搬的圆盘，那么就直接搬它；如果需要搬的圆盘多于一个，那就是递归情况，需要借助于另一根圆柱，将这一问题分解为三个部分，…。

为写出解决这一问题的递归函数，先将此函数命名为 henoi。首先考虑这个函数的类型特征，考虑它应该有哪些参数。显然函数应该有一个记录需要搬动的圆盘数（问题规模）的整参数，每次递归调用时这个参数减一，参数为 1 时可以直接搬（函数直接返回），否则就需要进一步递归。还需知道应该从哪个圆柱向哪个圆柱搬，以哪个圆柱作为辅助。假设我们用字符类型的值表示圆柱，这个递归函数的类型特征应该是：

```
void henoi(int n, char from, char to, char by)
```

我们想做：从圆柱 from 借助于 by 搬 n 个圆盘到圆柱 to。

至此，写出 henoi 函数的递归定义已经不困难了。下面是一种写法：

```
void moveone (char from, char to) {
    printf("%c -> %c\n", from, to);
}

void henoi (int n, char from, char to, char by) {
    if (n == 1) moveone(from, to);
    else {
        henoi(n-1, from, by, to);
        moveone(from, to);
        henoi(n-1, by, to, from);
    }
}
```

这里单独定义了一个 moveone 函数，以便将与输出有关的功能集中到一起。如果我们需要修改输出形式，现在就只要修改这个函数了。例如，如果有适当的图形显示系统，可以考虑将有关搬圆盘的动作用更形象的方式显示出来。如果用这个程序去控制一个机械手，真正地移动圆盘，那么 moveone 就应该去调用控制机械手的功能。下面的调用将输出把 4 个圆盘从圆柱 a 移动到圆柱 b 的各个移动步骤：

```
henoi(4, 'a', 'b', 'c');
```

这个函数虽然不长，但其执行时间随 n 值增大的速度却非常快。对 n 的值应用数学归纳法，很容易证明函数 moveone 的调用次数等于 2^n-1。请读者通过计时来检验这一理论结论。当然，由于输出的速度（与内部计算相比）非常慢，因此，那些输出大量信息的程序（例如 henoi 函数），执行中的许多时间都用在输出上。

我们完全可能采用循环结构实现这个程序，有很多可能方法，具体情况这里就不介绍了，读者可以设法自己做一做，或者去查找有关资料。

4.4 基本输入输出

程序总需要输入和输出。C 语言的输入输出都是通过标准库函数提供的，前面已经介绍过输出函数 printf 的基本使用方法，本节介绍另外三个常用的输入输出函数：scanf、getchar 和 putchar。这些函数都在文件 stdio.h 里说明（与 printf 一起），如果要使这些函数，程序前面需要写：

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

4.4.1 字符输入输出函数

getchar 是个没有参数的函数（简称“无参函数”），它从标准输入读一个字符，返回该字符的编码值。对于无参函数，在定义函数或描述函数类型特征时，应在参数表里写一个 void，指明函数没有参数的情况。因此，getchar 的类型特征应描述为：

```
int getchar(void)
```

标准库将函数 getchar 的返回值定义为 int 类型，而不是定义为 char 类型，这个问题将在下面解释。

调用无参函数时应在函数名后写一对空括号。getchar 的典型使用是：

```
n = getchar();
```

这个语句将从标准输入读一个字符，并把字符的编码赋给变量 n。默认情况下标准输入连到键盘，因此，当程序执行到这个 getchar 调用时，该函数将从键盘取字符。如果没有已经键入的字符，程序就会等待，直到人通过键盘输入字符，并按过换行（Enter）键之后，函数 getchar 才能得到结果，语句完成后程序继续运行下去。

函数 putchar 把一个字符送到标准输出。如果我们在程序里写：

```
putchar('O'); putchar('K');
```

两个字符“OK”将被送到标准输出。标准输出的默认连接通常是计算机显示器，因此，执行这两个语句的效果使字符在计算机屏幕上显示出来。

例：下面程序把由输入得到的一个字符输出，并换一行：

```
#include <stdio.h>
int main () {
    int c;
    c = getchar();
    putchar(c);
    putchar('\n');
    return 0;
}
```

这个程序执行时立即进入等待状态，等待输入字符。由键盘输入一个字符并按 Enter 键后，程序将把输入的字符重新输出来，换一行（因为它输出一个换行符）并结束。

4.4.2 字符输入中的问题

输入一系列字符

假设现在需要写一个程序，把由标准输入读到一系列字符送到标准输出。显然，该程序里需要反复读字符和输出字符，基本部分应写成一个循环。该循环的形式是：

```
while (....) {
    c = getchar();
    putchar(c);
}
```

现在的问题是怎样写循环条件，或者说，循环应该在什么情况下终止？我们想通过这个简单例子探索一个一般性的问题：如何处理由输入得到的信息。循环里的 putchar(c)可以它换成任意程序片段，去做希望程序做的其他事情。

这种循环结束有两种可能方式：第一，循环的终止由程序内部确定，与外界输入的字符无关。这种方式很简单，前面已经学过的方法足以解决问题了。例如在程序里用一个计数器，读入字符达到规定数量后循环立即结束。这种情况下没有什么新问题。

例：写一个程序，它由标准输入读 10 个字符，输出各字符的编码。程序很简单：

```
#include <stdio.h>
int main () {
    int c, n;
    for (n = 0; n < 10, ++n) {
        c = getchar();
        printf("%d ", c);
    }
    return 0;
}
```

这里将字符的值按整数输出，我们就可以看到各字符的编码值了。

第二种方式，程序根据输入的情况决定是结束循环还是继续循环。对于这种考虑，写程序时就需要根据客观情况，确定输入的结束条件，这种循环结束条件将成为写程序的人与使用程序的人之间的一种约定：当执行到达这个循环时，程序处理输入；当外界提供的输入满足了某些条件时，这一循环结束，程序继续做后面的事情。

例：写一个程序，它由标准输入读一行字符，输出这行字符的个数。

这个程序也很简单：

```
#include <stdio.h>
int main () {
    int c, n = 0;
    c = getchar();
    while (c != '\n') {
        ++n;
        c = getchar();
    };
    printf("%d characters\n", n);
    return 0;
}
```

利用 C 语言中赋值表达式也有值的性质，熟悉 C 语言的人通常将上面程序写成：

```
#include <stdio.h>
int main () {
    int c, n = 0;
    while ((c = getchar()) != '\n') ++n;
    printf("%d chars\n", n);
    return 0;
}
```

由于赋值运算符的优先级低，程序里循环条件中的括号是必不可少的。

一旦这样写出了程序，其中的循环体的执行次数就完全由程序执行时外部提供的输入确定了。对于上面这个例子，循环体将要执行多少次，就看人通过键盘提供多少个字符之后给了一个换行符。我们可以推广这一方式，连续做一段输入的循环可以写成遇到换行符结束，自然也可以写成遇到其他字符而结束，遇到什么字符结束就是我们（作为写程序的人）和为程序提供输入的人（可能也是我们自己）之间的一种约定。这也就是前面所说的，“循环结束条件将成为写程序的人与使用程序的人之间的一种约定”。

缓冲式输入

如果程序运行中要求从标准输入取得信息，假设是执行 `getchar`。我们通过键盘输入

字符后，需要按 Enter 键后程序才能得到有关输入。造成这种情况的原因是操作系统通常采用缓冲式输入方式。人通过键盘输入的字符临时保存在操作系统的“输入缓冲区”（系统管理下的一块内存区域），直至人按 Enter 键，操作系统才把位于缓冲区的的输入字符送给相应的 C 程序，此后 getchar 函数才能读到这些字符。

处理所有可输入字符

上面介绍的方法有一个大缺陷：这里总要选定一个作为结束判断用的字符。而这样做之后，该字符就不能再作为输入中的正常字符使用了。假如程序需要处理键盘能输入的所有字符，或需要处理所有可能的字符（在二进制编码范围 0～255 内的所有字符），上面的方法就行不通了。此时外部如何通知程序要处理的字符都处理完了呢？

标准库的输入函数都为这个问题提供了处理方案。标准库定义了一个符号常量 EOF（意为 End Of File，文件结束），getchar 在读字符时如果遇到文件结束，就返回 EOF 的值，说明已经没有输入了（下面将看到，其他输入函数也采用了类似的方式）。这样，如果把标准输入定向到系统中的某个文件，getchar 就会从该文件里一个个地读字符，在用完了所有字符后它就返回符号常量 EOF 的值。这样，程序里就可以通过判断返回值是否 EOF，确定被处理的文件是否读完。由此产生的一种循环写法是：

```
while (1) {
    c = getchar();
    if (c == EOF) break;
    ... ... /* 对输入的实际处理 */
}
```

这个循环的条件是 1，就是说条件一直成立。当输入数据全部用晚时，getchar 返回的 EOF 值将导致 if 语句的条件成立，此时执行 break 退出循环。

有些读者可能想知道 EOF 到底是什么，一般系统里把 EOF 的值定义为-1。但实际上这并不重要，因为程序里只需判断某个值是否与 EOF 的值相同，知道这个标识符就足够了。这里唯一的要求是：EOF 值必须与任何字符的编码值都不同（否则就会产生混乱）。正是为了处理这一情况，C 标准库里把 getchar 的返回类型定义为 int（而不是 char），因为 int 类型除了包含 char 类型所有的值之外，还可以有位置表示 EOF 信息。由于同样原因，调用 getchar 时也应当用 int 类型的变量接收其返回值，这样才能保证 EOF 信息不丢失，保证文件结束判断的正确性。

如果程序由标准输入读字符，默认情况下该程序将从键盘读入。我们怎样在键盘输入一个文件结束信息呢？不同系统的情况可能不同，但通常都可以用组合键输入这个信息。许多系统里可以用 Ctrl-Z 或 Ctrl-D 送入文件结束信息。在程序运行时，如果人按 Ctrl-Z 键，getchar 读到这里时返回的就是 EOF 值，从而导致循环结束。

例：写一个程序，从标准输入中读入一个个字符，并将读入的字符复制到标准输出，直到遇到文件结束：

```
#include <stdio.h>
int main () {
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

一个文件也可能没有内容，一开始就是文件结束，因此采取上面写法。还要请读者注意，这里必须有括住赋值表达式的括号。不写这对括号程序是什么意思？请读者分析。

例：写一个程序，统计由标准输入得到的文件中字符的个数。这个程序可以写为：

```
#include <stdio.h>
int main () {
    int c; long n = 0;
    while ((c = getchar()) != EOF)
        ++n;
    printf("%ld\n", n);
    return 0;
}
```

4.4.3 格式输入函数 scanf

写简单程序时，使用更多的可能是标准库函数 scanf，这个函数的使用形式和性质都比 getchar 复杂得多。scanf 是与 printf 对应的输入函数，它从标准输入读取信息，按照格式描述把读入的信息转换为指定数据类型的数据（的内部形式），并把这些数据赋给指定的程序变量。scanf 的使用形式也与 printf 类似：

scanf(格式描述串, &变量名, ...)

格式描述串的形式与函数 printf 类似，其中可以包含一个或几个转换描述（同样以 % 开头），这些转换描述说明输入的形式和数据的转换方式。格式串之后可以有若干个其他参数，参数个数应与格式串中转换描述一致。这些参数指明了接受输入数据的程序变量，其书写形式是在变量名前面加 & 符号。应特别注意，这个 & 符号必须写，不写 & 符号可能引起严重问题，例如导致程序执行中出错，甚至破坏系统（死机等）。这种写法的实际意义将在后面章节里介绍。

前面说过，调用 printf 时要注意两方面的一致性：格式串中的转换描述与对应位置的表达式（最简单情况下是普通变量）的类型必须一致，否则表达式的值就可能按错误方式转换，产生不正确的输出。使用函数 scanf 要求三方面一致：格式串中的转换描述、对应参数变量的类型，以及程序实际运行中人提供给程序的数据形式。

下表列出了最常用的转换描述，其对应参数变量类型和实际要求的输入间的关系：

转换描述	参数变量的类型	要求的实际输入
%d	int	十进制数字序列
%ld	long	同上
%f	float	十进制数，可以有小数点及指数部分
%lf	double	同上
%Lf	long double	同上

注意，scanf 的转换描述虽然在形式上与 printf 的转换描述相同，对转换对象的规定却有些差异。scanf 规定%f 对应的参数是 float；而%lf 对应的参数是 double。这些与 printf 的规定不同。请读者对照一下。

假设已经有了下面变量定义：

```
int n; double x; float y;
```

那么，语句

```
scanf("%d %lf %f", &n, &x, &y);
```

要求读入三个形式合适的数，scanf 将把读入的数按照指定类型进行转换，并把转换结果顺序赋给三个变量。正如前面所说，如果程序执行时人提供的输入（用键盘输入的字符序列）不符合这个语句的需要，输入工作也不可能正常完成。对于上面语句，输入的数值之间应该用空格或其他空白字符分隔。例如输入：

```
234 2252.18 220.4
```

这样 scanf 就会将 234 的值存入变量 n，将 2252.18 存入 x，将 220.4 存入 y。

在需要读入几个数值时，上例中转换描述间的空格并不必须。上面语句也可以写成：

```
scanf("%d%lf%f", &n, &x, &y);
```

注意，如果 sacnf 格式串的转换描述之间没有字符或者只有空格，输入的数据间也只能有

空白字符，不能出现其他字符。人们一般不在格式描述串里写转换描述之外的东西。有些初学者为 scanf 写如 "%d, %lf, %f" 一类的格式串，这实际上对输入提出了更多要求，输入时不注意就会导致错误，致使数据不能正常读入。在没有弄清这类复杂形式的意义之前最好不要采用。有关 scanf 格式串的细节问题在第九章有详细介绍。

scanf 返回 int 值：遇到文件结束时返回 EOF 值；一般情况下返回正确完成了的转换个数。假设某个 scanf 调用要求读入一个整数，执行时 scanf 会跳过开始遇到的空白字符，从遇到的第一个非空白字符开始处理。

例：写程序读入一系列数值，把每个数据作为圆盘半径，分别算出圆盘面积并输出。

由于这里要对一系列无法确定个数的数据做同样计算，显然需要写一个循环。由于不知道输入数据的项数，只能利用函数 scanf 的返回值控制循环。写出的程序如下：

```
#include <stdio.h>

void pc_area (double r) {
    if (r < 0)
        printf("Input error: %f\n", r);
    else
        printf ("r = %f, S = %f\n", r, 3.14159265 * r * r);
}

int main () {
    double x;
    while (scanf("%lf", &x) == 1) pc_area(x);
    return 0;
}
```

这里用 scanf 返回值为 1 作为继续循环的条件。由上面介绍可知，scanf 返回 1 表示它正确地读入了一个数据项，这里就是一个整数。如果文件结束，这一循环将如所希望的那样终止。如果在执行中遇到了错误数据，例如 scanf 在希望读入整数时遇到字母 m，这时由于无法把 m 解释为一个数，scanf 的转换工作失败并返回值 0（因为它正确完成了 0 个转换），因此循环也结束了。

例：写一个程序，它读入一系列数值，统计数据个数，确定其中的最小值和最大值，并计算出所有数据的平均值。

显然需要通过循环读入数据并完成其他工作。这里可以用两个变量记录已处理数据中的最小值和最大值，这样输入结束时就有了所有数据的最小值和最大值。每读入一个数值时都要考虑这两个变量的更新问题，使它们始终存着已读入数据的最小值和最大值（这是循环中需要保持的不变性质）。为了求平均值，还需要记录数据个数以及数据之和，为此还需要两个变量，并要在循环中对它们正确更新（也是循环的不变性质）。

这里出现的一个问题是最大值变量和最小值变量的初始值。下面对这个问题采取一种简单处理方式，假定输入数据集至少有一项。在这种假定下，程序就可以先输入一项数据，用它设置上述两个变量的初值（因为所有数据中最大的数据至少有这么大，最小的数据至少有这么小）。这样做出的程序如下：

```
#include <stdio.h>

int main () {
    double sum = 0.0, biggest, smallest, x;
    int count = 1;
    scanf("%lf", &sum);
    biggest = smallest = sum;

    while (scanf("%lf", &x) == 1) {
        sum += x;
```

```
            ++count;
            if (x > biggest) biggest = x;
            if (x < smallest) smallest = x;
        }

        printf("Count of numbers: %d\n", count);
        printf("Biggest: %f, Smallest: %f, Average: %f\n",
               biggest, smallest, sum / count);

        return 0;
    }
```

这个程序要求至少要求有一个输入数据项。如果改变问题的要求，允许没有数据项，这一程序应该怎么修改呢？这个问题请读者考虑。

4.4.4 输入函数的返回值及其作用

输入函数的返回值很重要，现在讨论一些有关的问题。程序可以分为两类，一类程序启动后自己运行，可能提供一些输出后结束；另一类程序执行中需要不断与外界打交道，需要外界提供信息，这类程序称为*交互式程序*（有输入的程序都可以看作交互式程序）。交互式程序要处理输入，但它们无法保证外界能提供正确的信息，无法保证交互中不出现异常情况，甚至无法保证外界不是故意捣乱。那么，我们在写这种交互式的程序时应当怎样考虑和处理输入的问题呢？这里的基本原则是设法处理可能出现的情况，因此，程序设法获取有关输入情况的信息就是非常重要的。

如果我们简单地写：

```
printf("Please enter an integer: ");
scanf("%d", &n);
for (i = 0; i < n; ++i) ...
```

如果当时读入整数的操作失败（用户输入不是合法整数），随后的循环会做什么就完全没有保证了。标准库的设计时考虑的交互式程序的这种需要，因此它提供的每个输入函数都用返回值说明函数执行中遇到的情况，包括正常情况和非正常情况。例如，scanf 返回一个 int 值，在遇到文件结束时返回 EOF 值，一般返回所完成的数据转换个数。通过检查这个返回值，程序就可以确定实际输入是否正确，从而决定下面应当如何处理。

在上面这类情形中，程序必须在得到一个合法整数的情况下，继续运行下去才有意义，采用上述简单写法是不能得到这种保证的。然而我们在写程序时确实不能保证用户的实际行为方式，因此只能做一些假设。首先可以假设用户的合作的，也就是说，他们希望使用这个程序，而不是想故意捣乱。此外再假设用户在看到提示之后就是想输入一个整数，但其输入中却可能出现错误，因此需要检查，在发生错误时要求用户重新输入。还有一个情况也值得考虑：当用户实际输入无法转换为整数时，scanf 一定是遇到了非数字非正负号的字符。在输入失败时这个字符仍然留在输入流里。如果再次调用 scanf 读入整数，首先遇到的还是这个字符，scanf 还会失败。因此在再次调用 scanf 之前必须将输入流中的一些东西丢掉。一种合理方式是丢掉当前输入行的所有内容，要求用户重新输入。采取这种策略，我们可以将上述程序片段重新写为：

```
printf("Please enter an integer: ");
while (scanf("%d", &n) != 1) {
    while (getchar() != '\n')  ;
    printf("Input incorrect. An integer again.");
}
for (i = 0; i < n; ++i) ...
```

这样就可以保证在执行 for 语句时，变量 n 中确实有了一个由输入得到的整数。如果对于 n 的值还有进一步要求（例如可能要求它大于 10 并小于 100），也可以写在条件中，在 n 值不满足要求时请用户重新输入。

作为另一个例子，如果程序某处要求输入三个整数，一种可能写法是：

```
for (i = 0; i < 3 && scanf("%d %d %d", &a, &b, &c) != 3; ++i) {
    while (getchar() != '\n')  ;
    printf("Ask for 3 integers. Please input again.");
}
if (i == 3) {
    /* 没有得到正常输入时的处理 */
}
```

这里的循环条件中判断了是否实际读到三个整数，在出现问题时就要求用户重新输入。在打印出错信息后，还用了一个循环把直到行结束的字符都读入并丢掉。这样做的想法是：输入出错的最常见情况是输入形式不对，因此无法进行正常的转换。如果不把这些形式错误的字符丢掉，它们就会阻塞输入流（它们不能被转换，永远也无法被使用，这又使它们一直呆在输入序列的最前面）。上面采取的写法允许使用者出 3 次错。

标准输入与文件

操作系统一般都允许对标准输入的重新定向，因此我们就可以把任何实际的文件定向到标准输入去，使其成为 getchar 或 scanf 的输入源。考虑到这种情况，我们在编程时可以不区分实际输入究竟是来自键盘，还是来自文件。如果需要处理的是一组连续输入，这两者就没有什么区别。在本书后面的讨论中，我们经常直接说程序从文件读入等等，而实际写的是从标准输入读入。对于标准输出的情况也类似。有些程序需要从明确给定的命名文件读入信息，或者向特定的命名文件输出信息。这称为文件输入和输出，标准库也提供了这方面的专用函数。有关情况将留到后面讨论文件处理的一章里介绍。

4.4.5 输入循环

现在对输入循环的控制方式做一个总结。如果在程序里需要输入一批数据，可以根据情况，采用不同的方式控制循环的进行和结束。下面通过实例总结一下这方面的技术。

通过计数器控制循环

假设在编程时已经知道输入数据的项数，我们就可以写一个计数循环，用计数器的值达到或者超过限定值控制循环的结束。

例如，假设现在要写一个程序求出一年 12 个月的总降雨量，提供给它的是每个月的降雨量。这一程序就可以通过一个计数的输入循环完成：

```
#include <stdio.h>

int main()
{
    double x, sum;
    int n;
    for (sum = 0, n = 0; n < 12; ++n) {
        printf("Enter next data: ");
        scanf("%lf", &x);
        sum += x;
    }
    printf("Annual Precipitation: %f\n", sum);

    return 0;
}
```

这是最基本的控制循环的方式。实际上，这种循环在编程时已知重复执行的次数，因此完全可以不用循环，而是采用重复写出相应的语句的方式实现。采用循环结构的优势在于用一段较短的描述代替了很长一段繁琐重复的描述。

用结束标志控制的循环

假定事先并不知道需要通过循环输入的数据的确切项数，采用计数循环的路就走不通了。在写简单程序时，另一种常见方式是用特殊“结束标志”控制的循环。看下面例子：

假定需要计算一批货物的总值，每次输入的是货物单价和数量。由于并不知道需要算的货物种类，因此无法使用计数循环。可以考虑用一种特殊“结束标志”通知程序所有数据都已输入完。例如用单价为 0 作为结束标志，就可以写出下面的程序：

```
#include <stdio.h>

int main()
{
    double price = 1, amount, sum = 0;
    while (price != 0) {
        printf("Enter next data (price amount): ");
        scanf("%lf%lf", &price, &amount);
        sum += price * amount;
    }
    printf("Total price: %f\n", sum);
    return 0;
}
```

这里先给 price 初值 1 是为了防止进入循环时条件不成立。最后一次循环多做了一次累计工作，由于 price 是 0，不会对结果造成影响。也可以采用条件语句和 break 语句，在循环的内部判断输入数据是否为 0 和退出循环。

第三种方式前面已经多次使用，以后的程序实例中还会频繁使用，那就是通过输入函数返回值传递信息，控制循环的结束。

输入流和输入操作

我们可以把标准输入看成一个绵延不断的字符序列（字符流）。程序里调用输入函数，就是想用掉该序列最前面的一个或几个字符。例如，getchar 用掉最前面的一个字符。scanf 要求用掉序列前面的几个字符（如果它们符合转换要求）。

输入序列中的字符用掉一个就少一个，未用的字符则仍然留在序列中。如果 scanf 工作中某个指定转换失败，输入流中的字符序列就将保持在当时的状态。特别的，如果第一个转换失败，函数 scanf 将返回 0，输入流保持不变。

作为一个例子，现在考虑把求圆盘面积程序中的循环条件改为：

```
while (scanf("%lf", &x) != EOF) ...
```

如果执行中输入了字母 m，这一程序将进入无穷循环。请读者根据上面讨论，设法弄清其中的原因。

4.5 控制结构和控制语句

C 语言还提供了另外两种控制结构和几种控制程序流程的语句。理论上说这些语句都不是必需的，它们的功能完全可以用前面已经介绍的结构实现。提供这些东西是为了使人书写程序更方便，有时使用这些语句可以使程序写得更简单，或者看起来更清晰。

4.5.1 do-while 循环结构

do-while 循环结构的形式是：

```
do 语句 while (条件);
```

其执行过程可以用图 4.2 的流程图描述：首先执行语句，然后判断条件。注意，这里的条件也是循环继续的条件，在条件成立（值不是 0）时继续循环；条件不成立循环结束。还请注

意围绕条件的括号和最后的分号，这些都是 do-while 语法上必不可少的成分。

与 while 循环的不同，在这里，作为循环体的语句执行在前而判断在后，所以循环体至少执行一次。对 while 结构而言，如果开始时循环条件不成立，循环体将一次也不执行。在一般程序里，do-while 结构的使用比 while 结构少得多。

下面是用 do-while 语句重写的立方根函数，请与前面用 while 写的函数比较一下：

```
double cbrt (double x) {
    double x1, x2 = x;
    if (x == 0.0) return 0.0;
    do  {
        x1 = x2;
        x2 = (2.0*x1 + x / (x1*x1)) / 3.0;
    } while (fabs((x2 - x1) / x1) >= 1E-6);
    return x2;
}
```

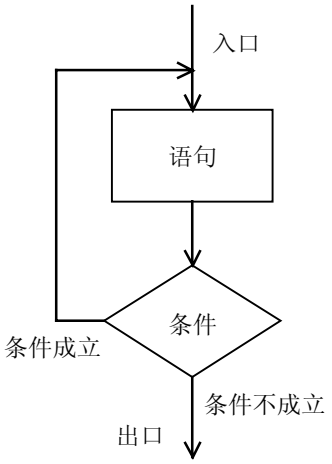

图 4.2　do-while 结构的执行流程

至此读者已看到了 C 语言的全部三种循环结构。在需要写循环时，可以根据情况选择使用。相对而言，while 和 do-while 循环结构比较简单，while 循环用得很多。另外，C 语言的 for 功能更强大，实际覆盖了 while 的功能。for 循环的结构比较复杂，成分较多，但它的设计确实反映了循环的典型特征，在 C 程序里使用非常多。在许多情况下，用 for 结构实现的循环往往比用 while 结构实现的循环更简洁清晰，特别是因为 for 结构把与循环控制有关的部分都集中在语句的头部，有利于人的阅读和理解。

4.5.2 流程控制语句

C 语言提供了三个流程控制语句：break 语句，continue 语句和 goto 语句。break 语句已经在前面介绍，现在介绍另外两个语句。实际上，只使用顺序结构（复合结构）、选择结构和循环结构，已经可以写出所有可能的程序了，语言中提供其他控制语句的主要目的是为了写程序方便，有时也是为了满足人们已经养成的编程习惯。这几个语句的形式都很简单。

```
while (...) {
    ... ...
    ... ...
    ... break;
    ... ...
    ... ...
    ... continue;
    ... ...
    ... ...
}
```

图 4.3　Break 和 continue 语句引起的控制转移

continue 语句

continue 语句的形式也很简单：

```
continue;
```

这个语句只能用在循环里，它使最内层循环体的本次执行结束，立即进入下一次循环。对于 while 和 do-while 循环，continue 执行之后的动作是条件判断；对于 for 循环，随后的动作是变量更新。

请注意 break 语句和 continue 语句的差别。break 语句导致最内层循环终止，使程序执行跳到这个循环语句之后；而 continue 引起的则是循环内部的一次控制转移，使执行控制跳到循环体的最后，相当于跳过循环体里这个语句后面那些语句。图 4.3 说明 break 语句和 continue 语句引起的控制转移的情况。

goto 语句

goto 语句又称*转移语句*或*转跳语句*，是计算机指令直接支持的控制动作，也是程序语言

里历史最悠久的控制语句。随着人们对程性质的认识发展，随着程序语言里其他控制结构和语句的出现，今天 goto 语句在程序语言里已经变得越来越不重要了。但是由于历史原因，许多语言仍然提供了 goto 语句，C 语言就是这样。

goto 语句需要与程序里的标号协同工作，其作用是实现在一个函数体内部的任意控制转移。标号可以写在任何语句前面，其形式是：

```
标号名：
```

标号的形式就是一个标识符后跟一个冒号，*标号名*可以是任意的标识符。一个标号表示了程序中的一个位置，作为 goto 语句转移的目标。goto 语句的形式是：

```
goto 标号名;
```

当程序执行到一个 goto 语句时，执行控制将立即转移到程序中相应标号所在的位置，从那里继续执行下去。

goto 语句的实现基础是 CPU 的分支转移指令。仔细想想，不难发现 break、continue 实际上语句都是受限制的转移语句，它们实现的是方式固定、意义更清晰的控制转移。

最早的高级程序语言里就有 goto 语句，那时它几乎是语言中唯一的控制语句。在早期程序实践中，人们无节制地随意使用 goto，写出了许多很“巧妙”的程序，其中的一些程序不但十分费解，有的还是看起来正确，其实带有隐藏很深、难以发现的错误。1968 年，荷兰计算机科学家 Dijkstra 写了一篇文章：“goto 是有害的”，引起计算机界的强烈反响，导致了一场持续大约六年的大辩论。这场辩论的结果是计算机领域的“结构程序设计”革命：新设计的各种语言都引进了几种标准的控制结构，计算机教育和实践中大力提倡结构化程序设计，尽量使用结构化的控制结构和函数（过程）抽象机制来写程序。对于 goto 语句的共同认识是：应当不用或尽量少用。

实际上，使用 goto 的大部分情况是在构造条件执行或者循环。典型的情况例如：

1）向前转跳	2）向后转跳

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

第 1 种形式在许多情况下构成了循环；第 2 种形式在许多情况下是条件执行。用循环控制结构或条件控制结构改写程序，可以使得到的程序更清晰易读，也更不容易有错。

有些读者过去接触过计算机，学过程序设计。其中有些人可能养成了随意使用 goto 语句的习惯。今天应当认识到，无节制地使用 goto 语句是一种不良程序设计习惯，基本上说明了一个人对要做的程序的结构没有清楚的认识。有人说，一个人写程序的水平与其在程序中使用 goto 语句的数量成反比。这话虽然未必准确，但也有一定道理。无论如何，含有大量 goto 的程序既难写正确，又难理解。我们绝不要养成这种习惯。本书所有的例子中都没有使用 goto 语句，不但解决了所提出的问题，结果程序也清楚简单，这至少也能说明一点问题。实际上，程序中不合理的 goto 表示写程序的人对问题的分析不清楚，没有做好程序流程分解，函数抽象等等，写出的是不成熟的程序。

由于提供了比较充分的控制结构，C 程序里真正“需要”使用 goto 语句的地方很少。这方面唯一合理的例子是需要从多重循环内部直接退出，一个可能的方法是用 goto 语句直接转移的循环语句之后，其大致形式是：

```
        for (......)
            for (......) {
                ....
                if (....) goto label;
                ....
            }
    label: ....
```

```
    ....
```

虽然也可以通过加入控制变量处理这种情况（并从而消除 goto 的出现），但那样做产生的程序并不更清晰易懂。因此，在这里使用 goto 是可取的。要注意，这里只是把 goto 语句作为退出循环的手段，转移的目标就是循环结束位置，随便转移仍是不可取的。

4.5.3 开关语句

开关语句（switch 语句）是 C 语言的另一种分支结构，，这是一种多分支结构，根据一个整型表达式的值从多个分支中选择执行。开关语句的形式较复杂，其一般形式是：

```
switch (整型表达式) {
    case 整型常量表达式: 语句序列
    case 整型常量表达式: 语句序列
    .... ....
    default: 语句序列
}
```

常量表达式指在编译时可以确定值的表达式，常用整型或字符型的文字量。default 部分可以没有；各 case 后的语句序列也可以没有（如果有，可包含一个或几个语句）。这里的“case 整型常量表达式:”当作标号看待，后面的表达式称为 case 表达式。

开关语句的执行：先求出整型表达式的值，然后用该值与各个 case 表达式的值比较。如果遇到相等的值，程序就从那里执行下去；如果找不到，而这一开关语句有 default 部分，那么就从“default:”之后执行；如果没有 default 部分，那么整个开关语句的执行结束。开关语句的一个基本要求是各 case 表达式的值互不相同。

例如，下面程序段按 n 的值确定分支，值为 1 和 2 时单独处理，其他情况统一处理：

```
switch (n) {
    case 1: ... ...
        break;
    case 2: ... ...
        break;
    default: ... ...
        break;
}
```

人们习惯在每个分支最后写一个 break 语句，包括最后一个分支，使语句序列执行完后能从开关语句里跳出。这样做的目的是防止修改程序时不慎引起错误。例如在最后增加了一个分支，但它前面没写 break 语句，就可能引进一个不容易发现的问题。

按 C 语言的规定，如果一个 case 分支的最后没有 break，它的语句序列执行完成后程序将进入下一分支的语句序列，这种情况导致一些分支的程序片段被共享。一般认为，除非多个分支都用完全一样的语句序列（也就是说，一些 case 标号后面没有语句序列，和后面标号使用同一语句序列），否则不提倡这种代码共享的方式。部分代码共享导致程序片段间的依赖关系，不利于程序的修改。

例：写一个程序，分别统计输入文件中的空格、行、数字、花括号（作为四个类）以及其他所有字符（放在一起作为一个类）。最后输出各项统计的结果。

显然这里应该有一个循环，逐个处理输入字符。循环体里需要分别处理各种情况，可以用 if 语句写。因为这里情况比较多，区分情况的依据是读入的字符，字符具有整数值，因此，人们常用开关语句处理这类情况。这样写出的程序如下：

```
#include <stdio.h>
int main () {
    int c;
    int nd = 0, nb = 0, nl = 0, nc = 0, nn = 0; /*5个计数器*/
    while ((c = getchar()) != EOF)
```

```
            switch (c) {
                case ' ':
                    ++nb; break;
                case '1': case '2': case '3': case '4': case '5':
                case '6': case '7': case '8': case '9': case '0':
                    ++nd; break;
                case '\n':
                    ++nl; break;
                case '{':
                case '}':
                    ++nc; break;
                default:
                    ++nn; break;
            }
        printf("spaces: %d, lines: %d, digits: %d", nb, nl, nd);
        printf("{ and }: %d, others: %d\n", nc, nn);
        return 0;
    }
```

在上面例子里可以看到多个分支（标号）共用一段代码的情况。

4.6 程序设计实例

本节讨论两个稍微复杂一点的编程示例，也是为了帮助读者总结前几章所学到的内容。

4.6.2 一个简单计算器

现在考虑一个简单的交互式计算器。假定它从键盘读入如下形式的输入行：

```
128+365
254+1438
10313+524
```

程序每读入一个形式正确的表达式后就计算并输出其结果，直至用户要求结束。

很容易想到，这个程序应该有一个基本循环：

```
while (还有输入) {
    取得数据
    计算并输出
}
```

由于需要输入和处理的是整数，简单的结束办法就是令程序在遇到非数字时结束。下面程序读入并计算表达式，如果第一个非空白字符不是数字（包括遇到文件结束）就退出：

```
#include <stdio.h>

int main () {
    int left, right;
    printf("Small Calculator\n");
    printf("Any no-digit character to stop.\n");
    while (scanf("%d", &left) == 1) {
        if (getchar() != '+' || scanf("%d", &right) != 1) {
            printf("Format incorrect. Format: nnnn+mmmm\n ");
            while (getchar() != '\n') // 丢掉本行剩余字符
                ;
            continue;
        }
        printf("%d + %d = %d\n", left, right, left+right);
    }

    return 0;
}
```

正如前面所说，如何结束循环是一种约定。

对这一程序可以做许多改进和扩充。本章后面的练习中提出了一些改进扩充工作，请读

者设法完成。也请自己设计一些进一步的扩充。

4.6.3 定义枚举常量

现在考虑在程序里定义“常量”的问题，即定义那种具有标识符形式，可以用在程序中不同地方，代表同一个常数的东西。定义“常量”的一种方式是写一个 enum 定义（枚举定义），其中给出要定义常量的名字（标识符）和希望这些标识符表示的常数值，可以很方便地定义出一组符号形式的常量。例如：

```
enum { NUM = 10, LEN = 20};
```

有了这个定义后，标识符 NUM 就代表 10，LEN 就代表 20。使用 enum 定义常量的限制是只能定义代表整数的常量，这样定义的常量称为*枚举常量*。第五章还介绍了常量的其他定义方式。有关 enum 的详细讨论在第九章，目前我们只将 enum 作为定义符号常量的一种机制。

下面是重写的完成摄氏华氏温度对照表的程序，程序的最前面定义了几个符号常量，程序里使用了所定义的符号常量，各常量分别代表被定义的常数值：

```
#include <stdio.h>

enum { START = 0, END = 300, STEP = 20 };

int main () {
    int c;
    for (c = START; c <= END; c += STEP)
        printf("C = %d, F = %d\n", c, c * 5/9 + 32);
    return 0;
}
```

这个程序很简单，但从中也可以看到使用符号常量的优点。首先，符号形式的描述比直接用数值好，写程序时可适当选择所用标识符，使其文字含义能有助于理解程序。如果程序里有两个 0，对阅读而言它们完全一样，而它们实际代表的意义可能完全不同。符号常量对提高程序可读性有很明显的作用。

此外，把程序里用的常数定义为符号常量，并在所有需要之处一致地使用符号常量，已被实践证明是一种很好的方法。这样写出的程序更容易修改。如果把常数直接写在程序里，修改常数值时就需要翻阅整个程序，仔细考虑每个数是否应该改。定义符号常量并一致地使用之后，修改常数就只需要修改符号常量的定义了。对于短小的程序，这种做法的收获还不明显。设想你需要在一个 100 万行的程序中把某个原来采用数值 0 的常数改为 1（这个常数可能在程序里到处使用），就可以理解采用符号常量的优越性了。

前面程序里多次使用的 EOF 就是标准库里定义的一个符号常量，不过它是采用另一种方式定义的，有关情况在第五章讨论。

4.6.4 单词计数问题

现在举一个很有价值的例子，在解决这一问题的过程中，我们采用了一种有效的分析和描述问题的方法。读者不仅应该注意这个例子本身，还应当注意其中的方法。本章及后面章节的许多练习题都可能利用这种方法，它能帮助我们比较容易把问题分析清楚，把程序写得更简洁和正确。

问题：一个正文文件可以看成一个字符的序列。在这个序列中，有效字符被各种空白字符（包括空格' '、制表符'\t'、换行字符'\n'等）分隔为一个个单词。现在要写一个程序统计文件中单词的个数，最后给出统计结果。

显然程序里需要一个计数器变量，处理中每遇到一个词时就将计数器加一。由于不知道被处理文件的内容，我们可以考虑用 getchar 读入。主要部分的框架可写为：

```
while (文件未结束)
    遇到一个词时计数器加一;
打印统计信息;
```

用函数 getchar 输入，判断文件结束很简单。剩下的问题就是如何确定“遇到一个词”，只要能完成这一工作，整个问题就迎刃而解了。

按照题目的要求，在逐个读入和检查字符时，只需要区分当前字符是否空格。不是空格的字符总是某个单词的一部分，空格的作用就是分隔单词。易见，在顺序输入中，只要能确定当前字符是单词的首字符，就知道又遇到新单词，这时应把词计数器加一。问题是：即使当前字符不是空格，它是不是新词的开始还依赖于前一字符是否空格。可见这里的处理步骤不能孤立进行，处理方式需要依赖于前面的历史情况。或者说，处理中需要记录遇到的情况，以便后面工作中参考。

通过分析可以总结出读入过程中的前后关系：1）当前字符是空格，这时应该通告后面步骤：如果遇到了非空格字符，那么就遇到了新词；2）当前字符不是空格，说明此时正处在一个单词的处理过程中，后面无论再遇到什么都不需要计数。

在计算机领域里，人们常把这类情况看作处理过程的不同状态，可以说这个读入过程有两种不同的状态：1）读入过程当时正处在单词之外（如果遇到非空格字符，那就是新词）；2）当时正处在某单词的内部（不会遇到新词）。在读入字符的过程中，程序的读入状态也在根据情况不断转换。这种过程在计算中很典型，计算机工作者们提出了一种形象的方式描述这种动态过程，它可以清楚地反映一个“系统”的状态、状态转换的条件、转换时的动作等等。这种抽象模型称为**自动机**。如果用 IN 和 OUT 作为读入过程当时处在词内部或外部的状态标志，程序读入字符的动态过程可以用图 4.4（自动机的图形表示方式）形象地描述。

现在我们可以用状态和状态转换的语言描述问题了：1）当读入处在 IN 状态时，读到非空格字符时状态不转换，读到空格将转到 OUT 状态。（2）当过程处在 OUT 状态时，读到空格时不转换，读到非空格字符将转换 IN 状态。不难看到，出现最后一种情况时就是遇到了新单词。

现在需要把分析清楚的过程用程序语言写出来。实现上述处理过程需要记录工作状态，为此设立一个状态变量 state，它的值可以是 IN 或 OUT，表示读入的当时状态。IN 和 OUT 用符号常量表示，这里只需要这两个值不同（程序不能同时处在两种状态）。这样，对当前字符（假设它存放在变量 c 中）的处理可以用下面程序片段描述：

```
if (c == ' ' || c == '\t' || c == '\n') /* 遇到空格等 */
    if (state == IN)
        state = OUT;
    else            /* state为OUT */
        state = OUT;
else                    /* 不是空格 */
    if (state == IN)
        state = IN;
    else {          /* state为OUT */
        state = IN;
        ++count;
    }
```

count 是计数器变量，这一程序片段直接对应于前面的自动机描述。不难看出，上面的直接翻译方式引进了一些冗余动作，在下面完整程序里，对于代码中的一些部分进行了可能的简化：

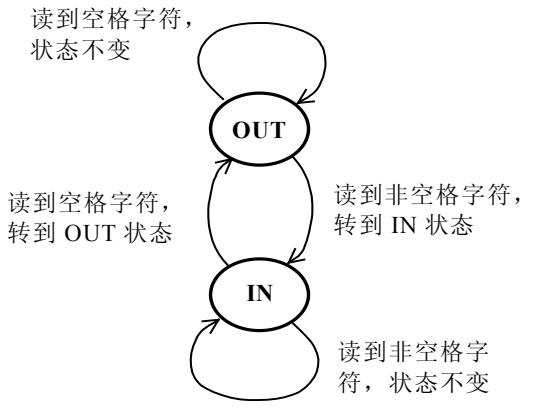

图 4.4　描述读入状态转换的自动机

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

```
enum { OUT = 0, IN = 1 };

int main () {
    int c, count = 0, state = OUT;
    while ((c = getchar()) != EOF)
        if (c == ' ' || c == '\t' || c == '\n')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++count;
        }
    printf("word count: %d\n", count);
    return 0;
}
```

当然，我们也完全可能用其他方式实现这一自动机模型。例如，下面程序采用另一种实现方式，请读者自己分析理解：

```
#include <stdio.h>

int main () {
    int c = ' ', count = 0;
    while (c != EOF) {
        while ((c=getchar()) != EOF && (c==' ' || c=='\t' || c=='\n'))
            ;
        if (c == EOF) break;
        ++count;
        while ((c=getchar()) != EOF && c!=' ' && c!='\t' && c!='\n')
            ;
    }
    printf("word count: %d\n", count);
    return 0;
}
```

这里用空格字符给 c 赋初值，只是为了保证循环开始时不会出问题。

这里用的分析方法和描述方法都很重要。本章后面有几个练习题，也可采用类似技术描述和处理。自动机是一类抽象计算模型，这里讨论的是最简单的有限状态自动机。自动机理论是计算机科学的一个重要领域，既有理论价值也有实用价值。这方面的进一步情况可以参考其他书籍。人们在计算机应用和程序设计实践中还提出了许多模型。这些模型一方面能作为工具，应用于实践，作为分析问题和描述问题的工具。另外，许多模型还有深刻的理论价值，能帮助我们进一步认识计算过程、程序、程序设计的规律性。在对计算机领域知识的进一步学习中，希望读者能有意识地关注这些模型和理论的作用及其重要性。

4.7 程序测试和排错

在写好一个程序，通过加工（编译和连接）产生可执行程序之后，就需要去运行它，提供一些数据去试验它，以便确认该程序确实能满足我们的需要，能完成预期的工作。程序测试（即试验性地运行程序）的工作应该如何做？应该提供什么样的数据去运行它？发现错误时应该如何应对？什么工具能帮我们检查程序里的错误？本节简单讨论这些问题。

我们当然希望程序一经写出就完好无缺。初学者在遇到程序有问题时，也常认为一定是“系统”有问题，或是计算机有毛病，自己在其中毫无过错，只是不够幸运，遇到其他东西捣乱。有经验的人们则知道，绝大部分情况下是自己的问题，需要设法排除的正是自己前段工作中所犯的错误。读者经过一段实践，应已看到做这方面的一些情况了。

测试和排错是程序开发过程中必不可缺少的阶段，人们对此积累了许多实践经验，也进行了许多研究，有不少这方面的论文和专著。本节将介绍在程序测试和排错的一些最基本技

术，希望进一步了解有关情况的读者，可以阅读 Kernighan 等的《程序设计实践》一书，其中有关这一方面的精辟讨论，也可以参考其他有关的书籍。

所谓测试（testing），就是要在完成一个程序或者程序中的一部分之后，通过一些试验性的运行，通过仔细检查运行效果，设法确认这个程序确实完成了我们所期望的工作。也可以反过来说：测试就是设法用一些特别选出的数据去挖掘出程序里的错误，直至无法发现进一步的错误为止。排错（debugging）则是在发现程序有错的情况下，设法确认产生有关错误的根源，而后通过修改程序，排除这些错误的工作过程。

显然，有些排除错误的工作出现在测试之前。我们在测试程序之前，首先要将开发出的源程序加工为可执行程序。在这个加工过程中就可能发现程序里的一些语法错误和上下文关系错误，只有排除了这类错误之后才能进入进一步的测试和排错阶段。

4.7.1 测试

测试中考虑的基本问题就是怎样运行程序，提供它什么样的数据，才可能最大限度地将程序中的缺陷和错误挖掘出来。通过思考，我们可以看到两类基本的确定测试数据的方式：

1. 根据程序本身的结构确定测试数据。这一做法相当于将程序打开，看着它的内部考虑如何去检查它，以便发现其中的问题。这种方式通常称为白箱测试。
2. 根据程序所解决的问题本身去确定测试过程和数据数据，并不考虑程序内部如何解决问题。这一做法相当于将程序看作一个解决问题的“黑箱”，因此称为黑箱测试。

下面分别简单介绍这两种测试方法。

4.7.2 白箱测试

白箱测试的基础是考察程序的内部结构和由此而产生的执行流程，设法选择数据，使程序在试验性运行中能通过“所有”可能出现的执行流程。这一做法的基本想法是：如果通过每种执行流程的计算都能给出正确结果，那么这个程序的正确性就比较有保证了。

几种基本控制结构所产生的可能执行流并不难理解：

1. 复合结构。复合结构只有一条执行流，从第一个语句开始，到最后一个语句结束。
2. 条件结构。“`if` (条件) 语句”有两条可能的执行流：条件不成立时语句不执行而直接结束；当条件成立时，它在执行了语句之后结束。因此，如果被测试程序包含一个这种形式的条件语句，我们就应设法通过提供的测试数据，检验程序在这两种执行流程的情况下都能正确完成工作。“`if` (条件) 语句$_1$ `else` 语句$_2$”的情况与此类似。
3. while 循环结构。循环结构的执行流情况更复杂些。从本质上说，循环：

   ```
   while （条件） 循环体
   ```

   可能产生无穷多条执行流：循环体不执行（第一次条件检测就失败），循环体执行一次，循环体执行两次，……。这意味着我们无法完全地测试一个循环的所有可能执行流程，这也正是循环结构比较复杂，理解和书写都比较困难的内在根源。为测试程序中的循环，常用方法是选择测试数据，检查循环的某些典型情况，包括循环体执行 0 次、1 次、2 次的情况，有时再选择若干其他典型情况。如果能确认在这些情况下程序的执行效果都如我们所期，我们对这个循环的正确性就比较有信心了。

控制结构可以嵌套，由此形成的执行流也就会复合起来。一段程序的所有可能执行流的数量可能很多，在测试它的时候，就需要系统化地一步步地做。例如，语句：

```
if (x > 0) ...
else if (y > 0) ...
else ...
```

存在着三条可能的执行流：`x > 0` 时，`x <= 0` 且 `y > 0` 时，和 `x <= 0` 且 `y <= 0` 时，

程序的执行将分别通过这三条执行流。

通过分析弄清了程序的可能执行流程，接着的工作就是设计测试数据，设法时程序通过各个执行流。这里不仔细讨论细节了。下面有关如何做黑箱测试的技术也可以参考。

4.7.3 黑箱测试

在做黑箱测试时，我们考虑的不是程序的内部结构（也可以假设它并不可知，即使可以知道，也可能因为是别人做的，或者因为它极其复杂，因此理解其所有的执行流非常困难），而只是考虑程序所解决问题的各种情况。

举例说，如果某个函数只有一个 int 类型的参数，那么我们就可以考虑检查在参数为 0 时这个函数能否得到正确结果，在参数为 1 和 -1 时，2 和 -2 怎么样，如此等等。如果对函数的性质有所了解，还应该考查那些已知结果的明显情况。

例如，对于计算立方根的函数：

```
double cbrt (double x) {
    if (x == 0.0) return 0.0;
    double x1, x2 = x;
    do {
        x1 = x2;
        x2 = (2.0 * x1 + x / (x1 * x1)) / 3.0;
    } while (fabs((x2 - x1) / x1) >= 1E-6);
    return x2;
}
```

我们应该测试该函数对于 0、1、-1、8、27 等等的情况。在确认这些情况都能正确工作后，再检查一些一般性的情况。

为测试 cbrt，可以写一个简单的主函数，其中包含一些调用 cbrt 做计算和输出结果的语句。但是，简单地做几次试验未必能使人确信所定义的函数完全正确（实际上也确实无法保证）。为了换一批数据等等再做试验，常常需要去修改源程序，重新编译后再运行。这样工作的的一个缺点是效率不高，而且还可能因为不小心而将新错误引进源程序。

换一种想法，可以考虑尽可能让程序帮助做事情，至少可以写程序提供一个试验 cbrt 的环境。实际上，常常只需写不多代码就可以做出一个试验工具。例如，为测试 cbrt 函数，我们可以写出如下的主函数：

```
int main () {
    double x, y;
    while (scanf("%lf", &x) == 1)
        printf("cbrt(%f) = %f\n", x, cbrt(x));
    return 0;
}
```

这个主函数使人可以方便地做一系列试验。我们首先应该试验一些特殊情况和容易判断正误的情况，而后再检查一些其他情况。下面是一些试验和输出：

```
0
cbrt(0.000000) = 0.000000
1
cbrt(1.000000) = 1.000000
27
cbrt(27.000000) = 3.000000
1000
cbrt(1000.000000) = 10.000000
100
cbrt(100.000000) = 4.641589
... ...
```

这种函数可以看作是一个针对 cbrt 的测试平台，有了它，随后的测试就很容易做了。

如果觉得这样反复输入并检查输出也很令人讨厌，那么就可以考虑让计算机帮助我们做更多的事情。例如将 main 改写为：

```
int main () {
    double x, y;

    for (x = -10.0; x <= 10.0; x += 0.11) {
        y = cbrt(x);
        if (fabs(x - y*y*y) > 1E-10)
            printf("Check error. x = %f, cbrt(x) = %f\n", x, y);
    }
    printf("Test finished.\n");
    return 0;
}
```

修改其中循环的起点、终点和步长，就可以让计算机去做许许多多的测试。一旦遇到某个计算结果的误差超过限制，程序就会将这个数据输出来，使人可以进一步检查。如果一切正常，这个程序将只输出一行信息就结束。

请注意这里的方法，对于许多计算函数（它们可能很复杂），常常有相对更简单的检查方式。例如这个程序里用乘法检查求立方根。这种想法常常很有用。

这种为检查有用的程序部分而写的虚拟主函数通常被叫做*驱动程序*，或者*测试平台*。在开发复杂程序的过程中，人们往往需要写许多这种代码段，以便一部分一部分地检查所开发的程序，为最后装配出完整的系统做准备。随着学习的进展，我们写出的程序也会变得越来越长、越来越复杂，因此也常需要写一些驱动程序，以便做一些测试。

请注意，这些测试方式并不一定要在计算机上做。在我们写出程序之后，先借助自己的手和眼睛，查看一下程序对典型数据的工作情况，常能发现一些很表面的常见错误。

4.7.4 排除程序里的错误

通过对程序的加工和试验性运行，我们可能发现了程序中的错误。发现之后就需要设法排除错误，这两方面的工作常常是交替进行的。检查和更正程序错误也是一种只有通过实践才能学会的技术。

有关排除语法错误的问题在第 1 章已有所论述，这里就不再重复了。如果我们的程序具有良好的格式，良好的组织结构，排除语法错误通常不难完成。在加工过程发现错误后，我们应仔细阅读系统产生的错误信息报告，根据它们设法确定错误的根源并更正之。下面考虑程序运行中产生出不正确行为或者结果的情况。

在测试中发现程序错误时，要排除错误，首先需要确定错误的根源。而确定错误的根源又需要考虑程序的执行流，分析确定所检查的错误是通过什么样的执行流产生的，需要考察程序对什么样的数据出现错误。

首先应设法保证程序对某些最基本的情况能够正确工作。如果连这一点也无法做到，那么就应该将注意力集中到这里。例如，如果程序里有循环，在循环体根本不执行的情况下程序结果对不对？程序对于数据 0 能否工作？在一个数据都没有的情况下（如果要处理的是一批数据，例如输入一系列字符或者数据时）能否正确工作。

特殊情况通常导致一些比较容易确定的执行流，我们应先解决这些简单情况下的程序问题。而后再考虑更一般的更复杂的情况。

如果程序对某个（或者某组）数据产生错误，那么就应该考虑试验一些相关的情况，看看出错的数据是否有某种规律性，它们所导致的执行流是否有规律性。而后检查执行流所通过的语句。逐步缩小范围，最终确定错误的原因。

还有一种常见错误是执行进入了死循环，程序在启动之后就再也不结束了。这种情况下，需要检查的就是程序里的各个循环，考察那个循环可能无止境地兜圈子。例如，各个循环的

循环变量是否正确更新等等。

检查程序错误的一种最常见技术就是在程序里插入适当的输出语句，将一些变量在程序计算过程中的值输出来，以便仔细检查。例如在输入语句之后存入语句，打印接受输入值的变量，可以帮助我们看到程序是否正确得到输入。如果程序出现死循环，插入其中的输出语句就会反复输出，也可以帮助我们标示死循环的范围。

许多程序开发环境里带有功能强大的动态调试和查错系统（称为 debugger），利用它们可以较方便地跟踪程序执行过程。但是我们发现，在初学程序设计的早期就养成依靠查错系统的习惯，往往会阻碍人对程序正确感觉，诱使初学者养成在没有想清问题的情况下就随便进入编程和查错的坏习惯。这样学习程序设计的人，后来编出的程序质量也比较差。因此我们建议，在学习基本程序设计的阶段不要依赖于系统的查错系统。

最后还应该指出，测试程序、排除程序错误的最重要工具就是我们的眼睛和头脑，这是任何其他东西兜不能代替的，也是别人不能代替的。在发现程序有错之后，应仔细阅读检查程序的相关部分，许多错误实际上是很明显的。为了有助于检查，程序的正确格式就愈发显得重要。作者经常遇到学生拿来自己写好后找不出错误的程序，而只要将程序的格式整理清楚，错误立刻就被发现了。这种情况也请读者注意，其教训就是：*如果你看不到程序里的错误，请先将程序的格式整理好。*

文中问题的解释

1） (4.3.4) 保证变量 m、n 的最大公约数仍是两个实参的最大公约数。

2) (4.4.2) 赋值中出现类型转换。由于 EOF 不是 char 类型所能表示的，给 char 类型的变量赋值时转换结果无定义。在随后与符号常量 EOF（int 类型）比较时，存入字符变量的值又转换回 int，但已不可能是原来的值了。所以文件结束的信息丢失了。

3) (4.4.3) 例如判断第一个输入是否遇到文件结束。

本章讨论的重要概念

循环控制变量，累积变量，递推变量，浮点计算的误差，迭代公式（递推公式），素数，斐波那契序列（Fibonacci 序列），程序计时，循环不变关系，最大公约数，欧几里德算法，do-while 循环，流程控制语句，break 语句，continue 语句，goto 语句，标号，结构化程序设计，字符输入输出，文件结束与标准常量 EOF，格式化输入函数 scanf，开关语句（switch），常量表达式，定义符号常量，状态与转换，自动机

练习

1. 1）写出通过递推方式求 200 之内的完全平方数的程序；2）写出只使用加法的求完全平方数的程序；3）写出求 1000 之内的完全立方数的程序，请参考书中实例的写法和上面的两种写法，分别写出相应的求立方数的版本。
2. 试验正文中乌龟旅行的实例，看看在你所用的 C 系统上得到什么样的结果。从数学教科书中找出有关调和级数的理论结论，并将它与我们的试验做一个比较。
3. 写一个程序，计算并输出 Fibonacci 序列中一系列的相邻项之比。确定一个范围，观察输出的结果，能够得到什么结论（这个比的序列可能有极限吗？极限是什么）。查阅有关资料，了解有关的理论结果。
4. 写函数计算 1! + 2! + ... + *k*!。用主函数试验函数对一系列 *k* 值计算出的结果。你写出的函数对 1 到 10 计算结果都正确吗？如果出现错误，弄清楚是什么原因。这个程序能对 *k* = 30 得到正确结果吗？（另外，你能只用一重循环完成函数的定义吗？）
5. 写函数计算：$f(x) = 1+1/(1+1/(\cdots+1/(1+1/x)\cdots))$，公式中有 *n* 层嵌套。利用这

个函数打印 `x = 1.0、2.0、…、20.0`，`n = 10` 时的函数值表。

6. 实现书中讨论的验证哥德巴赫猜想的程序，用不同的 *n* 对 6～*n* 的范围试验该程序。去掉程序中的打印输出语句，增加计时功能，对不同的 *n* 运行程序，考察程序的运行时间，画出一条曲线，说明运行时间与 *n* 的关系。
7. 设法（从文献中）找到其他更有效的素数判断方法并实现对应函数。在一个数值比较大的整数区间试验书上给出的函数和你写的其他函数，利用它们打印出这一区间中的所有素数。你所试验的几种方法在工作效率上有明显差异吗？（为程序计时）
8. 定义函数：`void prt_factors(int)`，它对正整数实参输出其所有的因子。
9. 定义函数：`void prt_pfactors(int)`，它对正整数实际参数，输出其所有的素因子（多重因子重复输出）；对于负参数，首先输出-1，然后输出所有因子。
10. 已知 $\sum_{n=1}^{\infty}\frac{1}{n^2}=\frac{\pi^2}{6}$，利用该公式编程序求 π 的近似值，看用这个和式的前多少项求出的近似值与 3.14159165 的误差小于 10^{-5}，令程序输出三项数据：计算得到的和，由这个和求出的 π 的近似值，求得该和所用得项数。把 10^{-5} 改为 10^{-6}、10^{-7} 并重新试验，用计时方式总结出误差减小与执行时间之间的关系。
11. 修改书中计算 sin 值的函数，使之能输出计算中循环执行的次数。用不同的数值（一个比一个大）试验这一函数，观察出现的情况。你看到出现溢出的情况了吗？（为试验方便，你应该写一个适用的驱动程序）
12. 已知 $\sinh^{-1}x=x-\frac{1}{2}\cdot\frac{x^3}{3}+\frac{1\cdot3}{2\cdot4}\cdot\frac{x^5}{5}-\frac{1\cdot3\cdot5}{2\cdot4\cdot6}\cdot\frac{x^7}{7}+\cdots(x<1)$，写函数 `double asinh(double)` 计算 $\sinh^{-1}x$ 的近似值。
13. 考虑不用函数（例如 `isprime`）直接写出对 6～200 的偶数验证哥德巴赫猜想的程序（利用循环、条件、`break` 语句等，不使用 `goto` 语句）。将这样写出的程序与用定义函数的方式写出的程序比较。例如考察两个函数的行数、字节数，其中各种控制结构的嵌套深度，控制结构使用的个数等。
14. 辗转相减求最大公约数的递归定义是（其中 $m>0,n>0$）：

$$\gcd(m,n)=\begin{cases}m & m=n\\ \gcd(n,m) & m>n\\ \gcd(m-n,n) & m>n\end{cases}$$

利用这个定义，用递归和循环方式各写出一个求最大公约数的函数。

15. 对一些 n 值试验河内塔程序，给它们计时。据此估计你所用的计算机搬完 64 个金盘需要多长时间。如果僧侣们 1 秒钟搬金盘 1 次，搬完 64 个金盘需要多少时间？将这一时间与科学家对宇宙的估计寿命做个比较，据此评价僧侣们的说法。
16. 一个三位的十进制整数，如果它的三个数位数字的立方和等于这个数的数值，那么它就被称为一个“水仙花数”。定义函数判断一个整数是水仙花数，并利用这个函数打印出所有的水仙花数。
17. 对一个整数，如果其所有因子（包括因子 1 在内）之和正好等于这个数，那么就称它为“完全数”。因子之和小于自身的数称为“亏数”；因子之和大于自身的数称为“盈数”。写一个函数，当其参数是亏数时返回负值，是完全数时返回 0，是盈数时返回正值。利用这个函数求出 1000 以内的所有完全数（实际上只有 1、6、28、496）。为这个程序计时：从 100 开始每隔 100 做一次计算，写一个循环，输出各次计算花的时间。再从 1000 开始，每隔 1000 做一次计算直到 10000 为止，输出对程序执行计时的值。利用所定义的函数对一段区间的整数做一个分类，输出其中各个数所属的类。
18. 写程序由标准输入得到一系列三个一组的数，把每组数作为三角形的三边长，计算三角形的面积。注意在程序里检查输入数据，对不能构成三角形的情况给出错误信息。仔细

分析自己的程序，能否检查出所有不合理数据。用不同数据运行试验。

19. 写一个程序，它读入一系列整数，最后输出其中最大的两个数。
20. 写一个程序，它输出所读入的一系列整数的平均值。假定给它的第一个数并不是数据，而是用于说明数据的项数。
21. 假设程序由输入得到的一系列正实数是一条折线在 x 等于 0，1，2，…的对应值（数据的数目事先并未确定），请求出这一折线与 x 轴之间区域的面积。
22. 能够组成直角三角形三个边的最小一组整数是 3、4、5。写程序求出在一定范围里所有可以组成直角三角形三个边的整数组，输出三个一组的整数。设法避免重复的组。
23. 改造本章正文中讨论的计算器程序：
    1）增加计算一行后的处理，使出现在正确表达式之后的任意字符序列都不会影响下一表达式的计算（抛弃正确表达式之后的字符）。
    2）修改程序，使之在数值、加号之间出现任意个空格时都能正确计算。
    3）修改程序，使之不仅能处理加法，还能处理其他算术运算符（提示：记录运算符，用 `if` 或者 `switch` 判断、计算和输出）。
24. 仿照本章正文中的单词统计程序，写一个统计 C 程序里标识符个数的程序。在程序里可以利用标准库提供的字符分类函数：

    `int isalpha(int c)`　当 `c` 是字母的编码时，返回非 `0` 值；

    `int isdigit(int c)`　当 `c` 是数字的编码时，返回非 `0` 值。

    使用这两个函数时，应在程序文件前部写 `#include <ctype.h>`。
25. 定义下面的计算规则：遇到 1 就终止；如果遇到 1 之外的奇数，求出它乘 3 加 1 的值作为下一个数；如果遇到偶数，求出它除以 2 的值作为下一个数。问题是：对每个正整数，反复使用这套规则，最终是否都能得到 1。也就是问，按照这个规则定义的递推序列是否对每个正整数都终止（这个著名问题至今也没有结论）。请按上面的规则定义一个函数，它返回对参数计算直到终止所经历的计算步数。对一系列参数试验这个函数。注意：序列中的数可能变得很大，请设法检查溢出，遇到溢出时打印信息。
26. 考察目前银行对各种期限的储蓄利率。写一个函数，它能够计算出对给定的时间（例如 8 年，10.5 年等，以半年为基本单位）如何分段将能够得到最大的收益。
27. 为本章正文中的某些程序实例写出好用的测试驱动程序。为你所做的某些练习写出好用的测试驱动程序，并借助于它们仔细检查你所定义的函数。
28. 用你认为最准确的方式描述本章中各个实例里的每个循环的循环不变关系。
29. 为你自己做的各个程序写出循环不变关系。设法检查你程序里循环中的语句，看它们是不是能维持有关的不变关系。