

5.5 for 语句

`while` 和 `do-while` 语句往往用在循环次数未知的情况下。当循环次数确定的时候，可以使用 `for` 语句。

5.5.1 for 语句的语法

`for` 循环仍然是一种当型循环，其语法如下：

```
for (初始化表达式; 条件表达式; 增量表达式)
{
    循环体
}
```

其中

- 初始化表达式：一般为赋值表达式，为循环控制变量赋初值。初始化表达式只被计算一次。
- 条件表达式：一般为关系表达式或逻辑表达式，作为控制循环测试条件。当结果为真时，循环继续，否则，表达式循环终止。
- 增量表达式：一般为赋值表达式或自增、自减表达式，为循环变量增量或减量。增/减量值一般称为“步长”。

`for` 语句的流程图 5-6 所示。

对例 5-1，使用 `for` 语句可以使程序更加简洁：

```
int i, sum=0;
for (i = 1; i <= 10000; ++i) sum += i;
```

或者：

```
int i, sum;
for (i = 1, sum = 0; i <= 10000; ++i) sum += i; //初始化表达式是一个逗号表达式
```

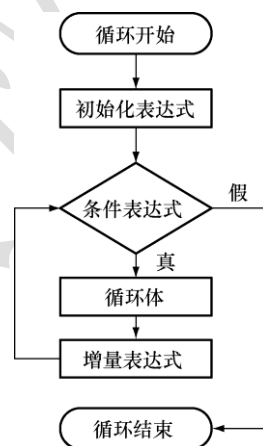


图 5-6 for 语句的流程图

5.5.2 for 语句的变体形式

`for` 语句的使用非常灵活，其中的 3 个表达式根据实际情况都是可以省略的，但每一个“;”都不能省略。省略了表达式的 `for` 语句有以下几种变体形式：

(1) 省略初始化表达式。如果在 `for` 语句前面给循环控制变量赋了初始值，那么初始化表达式可以省略。例如：

```
i=1;
for (;i<=10000;++i) sum += i;
```

(2) 省略增量表达式。如果 `for` 语句的循环体中有改变循环控制变量的操作，那么增量表达式也是可以省略。例如：

```
for (i=1;i<=10000;)
{
    sum +=i;
    ++i;
}
```

需要提醒读者，遗忘改变循环控制变量可能造成死循环。

(3) 省略条件表达式。如果循环体中包含有测试循环是否结束的代码，那么条件表达式可以省略。此时，测试条件为永真。例如：

```
for (i = 1;; ++i) //相当于 for(i= 1; 1; ++i)
{
    if (i<=10000) //这个测试保证了循环可以在有限次数后结束
```

```
        sum += i;
    else
        break;    //break 是强制退出循环语句，后面进一步讲(5.7.1)。
}
```

此例还演示了在循环语句中嵌入 `if` 语句的情况。可以看到，整条 `if` 语句都嵌入到 `for` 语句当中而不能交错。

(4) 省略所有 3 个表达式。例如：

```
for (;;) ...
```

不看循环体，那么这就是一个死循环。因此，必须保证在循环体内有退出循环的机制。

5.5.3 穷举法

在前面的章节中介绍了迭代法。但这种方法不能解决所有的循环问题。请看下面的例子。

【例 5-5】从键盘输入一个任意正整数，编程判断它是否是质数。若是，输出 “Yes”，否则输出 “No”。

【解题思路】在前面的章节里已经详细探讨过判断素数的方法，这里再来复习一下设计好的算法。

- (1) 给定数 p ，令 $q = \sqrt{p}$ ；
- (2) 令 $i = 2$ 。
- (3) 令 $r = p \% i$ 。
- (4) 如果 $r == 0$ ，则表明 p 不是质数，转到第 7 步；否则， $++i$ 。
- (5) 如果 $i \leq q$ ，那么转向第 3 步。
- (6) 否则， p 一定是质数，输出 “Yes”，转到第 8 步。
- (7) 输出 “No”。
- (8) 结束。

可以看到，这类算法与迭代法不同。迭代法试图找出级数中的递推关系，而这类算法试图在一定范围内将所有的可能都测试一遍，直到找到正确答案，或者发现无解为止。这种循环的模式称为“穷举法”(*enumerative method*)，也是一种常用的解决方案。

读者应该已经确定，上述算法的实现肯定要用到循环语句。但现在的问题是：步骤 6 和步骤 7 都是循环语句之后的步骤，并且只有一步能被执行。如何保证做到这一点呢？一个常用的技巧就是在得到结论后，先不急着重输出，而是用一个指示变量来保存结论的指示结果，然后在循环之外对这个指示变量进行测试，最后再输出结论。图 5-7 是本例的流程图。

```
//5-5.c
#include <stdio.h>
#include <math.h>

int main()
{
    int p, i, isPrime = 1; //isPrime 指示变量

    printf("请输入一个正整数:");
    scanf("%d", &p);
    for (i = 2; i <= sqrt(p); ++i)
    {
        if (p % i == 0)
        {
            isPrime = 0; //肯定不是质数
            break;
        }
    }
    printf("%s", isPrime == 1 ? "Yes" : "No");

    return 0;
}
```

运行结果如下。

请输入一个正整数:17✓

Yes

另一次运行结果为:

请输入一个正整数:24✓

No

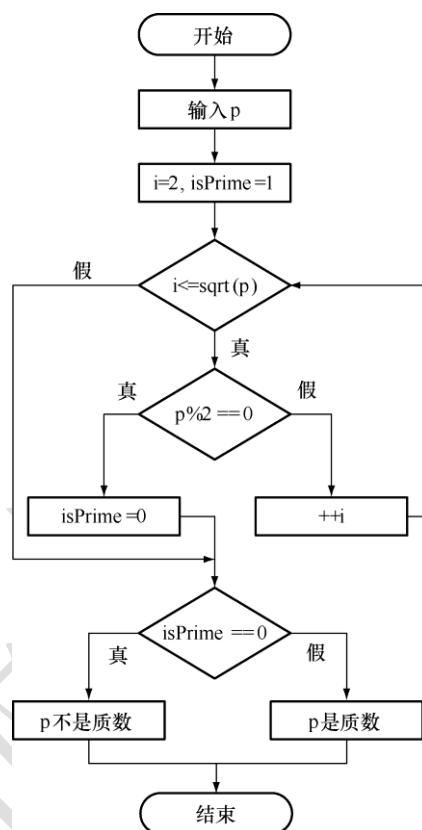


图 5-7 【例 5-5】的流程图



1. 初学者在使用 for 语句时也容易犯“分号病”。例如:

```
for (sum = 0, i = 0; i < 10000; ++i); //请注意这个分号的存在
sum += i;
```

最终的结果是 sum = 10001。原因很简单: for 语句后面的空语句将 sum += i; 这条语句排除在 for 循环的循环体之外, 因此 for 语句白白做了 10000 次循环, 循环结束后 i = 10001; 而 sum += i 只执行了一次, 因此 sum 的值就是 10001。

2. 多次修改循环控制变量。例如:

```
for (i = 0; i < 10; ++i) { ...; ++i; }
```

每一次循环都会使 i 加 2, 因此实际的循环次数没有到 10 次。

【例 5-6】求 1~10000 之内所有奇数之和。

【解题思路】在范围内最大的奇数是 9999, 那么累加数可以从这个数开始。由于累加数是奇数, 因此每一次的累加数都可以由上一次的数减 2 得到。在判断循环继续条件时, 只要累加数不是负数就继续, 否则就结束循环。

```
//5-6.c
#include <stdio.h>

int main()
{
    int sum = 0;
    int odd;

    for (odd = 9999; odd > 0; odd -= 2)
        sum += odd;

    printf("sum=%d", sum);
}
```

```
return 0;
}
```

程序的运行结果是：

```
sum=25000000
```

本例示意了两种情形：

- (1) 循环计数变量可以从大到小递减；
- (2) 循环控制变量的步长可以是能满足需要的任意值，不必总是 1。

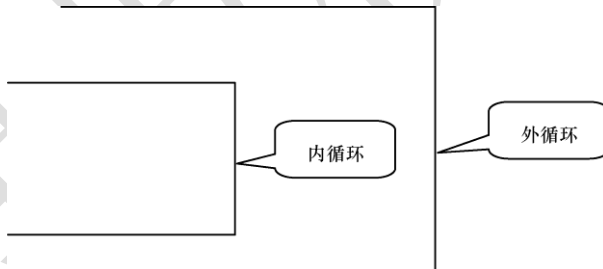
习题 5-3 请编程实现从键盘输入一个正整数，然后输出这个数的所有因子。

5.6 循环嵌套

一个循环语句的内部可以包含另一个循环语句，这样的结构称为“**循环的嵌套**”。while、do-while 和 for 这 3 种循环均可相互嵌套，即在它们的循环体内都可以再使用上述任一种循环结构。循环嵌套常用于解决矩阵运算、报表打印这类问题。

一般说来，嵌套循环中涉及几个循环结构就称之为几重循环。下例示意了 for 和 while 嵌套形成的二（双）重循环：

```
for (i=1;i<10;++i)
{
    while (j<10)
    {
        ...
        ++j;
    }
}
```



使用循环嵌套需要注意以下问题：

- (1) 尽量使用复合语句（即多用花括号）以保证逻辑上的正确性；
- (2) 内外层的循环变量名应不同，以免造成阅读困扰；
- (3) 不允许循环交叉，即内循环必须完全包含于外循环内；
- (4) 建议采用缩进的书写格式，使得层次清晰，易于检查。

【例 5-7】编程实现九九乘法表的计算和打印。

【解题思路】九九表是一张有 9 行 9 列的表格，其中第 i 行是 i 分别与 1~9 相乘的结果。据此可以发现，打印 9 行需要一个循环，而打印某一行的 9 列也需要一个循环。因此采用双重循环来实现计算和打印。为了保证打印的结果美观整齐，还需要考虑如下情况。

(1) 输出的结果有 1 位数，也有 2 位数。如果仅用空格分开各项，那么肯定会出现数据不能对齐的情况。使用输出域宽控制可以很好地解决这个问题。

(2) 当每打印完一行数据后，如果没有特别处理，那么后续的输出将会在同一行上继续，从而不能形成一张表格。因此，需要在完成一行的输出后额外打印一个换行。

以下是程序编码。

```
//5-7.c
#include <stdio.h>

int main()
{
    int row, col;

    for (row = 1; row < 10; ++row)          //外循环表示行
    {
        for (col = 1; col < 10; ++col)      //内循环表示列
```

```

        printf("%4d", row * col);    // %4d 的宽度为了统一对齐
        putchar('\n');              // 输出一行后换行
    }

    return 0;
}

```

运行结果如下:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

这里分析一下外循环（简称 row 循环）和内循环（简称 col 循环）的执行情况。

（1）row 循环的 for 语句执行 1 次，其循环体中的 col 循环的 for 语句和 putchar() 都要执行 9 次。

（2）每一次 row 循环中，col 循环的 for 语句执行 1 次，其循环体中的 printf() 要执行 9 次。那么，9 次 row 循环后，printf() 共要执行 $9 \times 9 = 81$ 次。

（3）在每一次 row 循环中，循环控制变量 row 保持不变；而 col 循环的控制变量 col 要从 1 计数到 9。表 5-1 就说明了这种情况。

表 5-1 双重循环中循环控制变量 row 和 col 的变化规律

外循环序号	row 的值	col 的计数规律	row * col 的变化规律
1	1	1, 2, 3, ...9	$1 \times 1, 1 \times 2, 1 \times 3 \dots 1 \times 9$
2	2	1, 2, 3, ...9	$2 \times 1, 2 \times 2, 2 \times 3 \dots 2 \times 9$
...
9	9	1, 2, 3, ...9	$9 \times 1, 9 \times 2, 9 \times 3 \dots 9 \times 9$

【例 5-8】示意了内层循环的循环次数是固定的情形。而有些双重循环的内层循环次数是不固定的，并且会依赖于外层循环的循环控制变量的当前值。【例 5-8】示意了这种情况。

【例 5-9】打印如图 5-8 所示的由 “*” 组成的三角形。

【解题思路】凡是涉及行列的问题一般都会用到双重循环，此例也不例外。通过观察题目给出的三角形图案，可以得出：第 i 行（i 从 1 开始计）有 i 个 “*” 号。假设外层循环控制变量为 k，从 1 计数到 7；内层循环控制变量为 j，那么它的计数范围就应该是从 1 到 k。

```

//5-8.c
#include <stdio.h>

int main()
{
    int k=1,j;

    while (k<=7)
    {
        for (j=1; j<=k; ++j) //每行输出 k 个 “*”
            putchar('*');

        putchar('\n');
        ++k; //这一行很重要!
    }
    return 0;
}

```

图 5-8 *组成的三角形

习题 5-4 将【例 5-4】扩展成:找出指定范围内的所有质数。指定范围由用户输入。

5.7 break 和 continue 语句

循环的定义是只要条件满足,循环体就会反复不停地执行,直到条件不满足为止。到目前为止,能终止循环的情况只有一种,即条件表达式值为 0 (假)。但在解决某些问题时,需要在条件表达式还未达到假之前提前结束循环。这就需要执行无条件转移控制语句 **break** 或 **continue**。

5.7.1 break 语句

在前面的讲述中我们已经知道, **break** 可以终止 **switch** 语句中某个 **case** 标号的执行。此外, **break** 也可用于循环语句中,用以提前终止循环。例如:

```
while (...)
{
    ...
    break; //直接终止循环
    a = 0; //这条语句及其后的循环体内的语句永远不会被执行
    ...
}
```

在实际应用中, **break** 语句常常被放在一条 **if** 语句当中。例如,要求输入一串字符,长度不超过 50, 或者 '*' 时提前结束,其编码如下。

```
count=0;
while (count<50)
{
    c=getchar();
    if (c=='*') break;
    ++count;
}
```

在上例中,即使没有输入到 50 个字符,只要遇到 '*', 循环也会提前终止。

break 语句虽然具有终止循环的功能,但却只能终止直接包含它的那条循环语句。如果包含 **break** 的循环是个内层循环,那么该 **break** 不能终止外层循环。例如:

```
for (i=0;...)
    for (j=0;...)
        if (...) break; //只能终止 j 循环。此后返回到 i 循环处继续执行。
```

如果要从嵌套很深的多重循环中直接跳出来,那么必须使用一些技巧,或者使用 **goto** 语句。由于 **goto** 语句的争议较大,因此这里就不再讨论了。

5.7.2 continue 语句

另一种终止循环的语句是 **continue** 语句。不过,它不是终止整个循环,而是终止本次循环,即不执行 **continue** 后面的语句,直接进入该循环结构的下一次循环操作。执行流程如下。

```
while (...)
{
    ...
    continue; //本次循环在这里终止。流程返回到循环测试部分,准备进入下次循环
    ++a; //这条语句及其后的代码永远不会被执行
}
```



在 **for** 语句中, **continue** 会立即转向计算增量表达式,以改变循环变量,然后再判定条件表达式,以确定循环是否继续。

同样地, **continue** 语句一般也被包含在一条 **if** 语句中。

【例 5-10】输入一行字符，统计字符的个数，如果有同一个字符连续输入几次，则只统计一次。

【解题思路】如何判断本次输入的字符与上次输入的相同呢？这肯定需要保存上次输入的结果，然后将本次输入与暂存的上次结果进行比较，如果相同则不需要计数；否则，增加计数值，同时用本次输入替换上次暂存结果。

```
//5-9.c
#include <stdio.h>

int main()
{
    char ch_old, ch_new;

    int count=0;
    ch_old='\n';
    do
    {
        ch_new=getchar();
        if (ch_old==ch_new)
            continue;
        ch_old=ch_new;
        ++count;
    } while (ch_new!='\n');

    printf("共%d 种字符\n", count-1); // 最后一个回车符需从 count 中减去

    return 0;
}
```

运行结果如下：

```
abbccdeffghiijjkk✓
共 12 种字符
```

与 `break` 相似，`continue` 只会影响到直接包含它的那条循环语句。

如果在嵌套循环中要终止内层循环而进入外层循环，应该使用 `break` 而不是 `continue`。

5.8 循环的应用

读者已经学习了 3 种循环语句，了解它们的异同，以及如何终止循环。现在再通过一些综合例子来加深印象。

5.8.1 迭代法的应用

【例 5-11】已知 Fibonacci 数列的递推关系如下：

$$F_1 = F_2 = 1$$

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

求该数列第 20 项。

【解题思路】递推关系的存在暗示了要用到迭代来处理此类问题。但此问题的特殊之处在于：前面例子中都是单项递推，而此例却是两项递推，那么该如何处理它们的关系呢？

假设有变量 `Fn`、`Fn1`、`Fn2` 分别代表了 F_n 、 F_{n-1} 和 F_{n-2} ，那么很明显地，有：

$$Fn = Fn1 + Fn2$$

此后，当前的 `Fn` 应该成为下一次的 `Fn1`，而当前的 `Fn1` 应该成为下一次的 `Fn2`。这样，可以令

$$Fn2 = Fn1$$

$$Fn1 = Fn$$

然后再重复操作就可以了。当循环结束后，`Fn` 就是要求的项。

```
//5-10.c
#include <stdio.h>
```

```

int main()
{
    int Fn, Fn1 = 1, Fn2 = 1;
    int i;

    for (i = 2; i < 20; ++i)
    {
        Fn = Fn1 + Fn2;
        Fn2 = Fn1;
        Fn1 = Fn;
    }
    printf("F(20)=%d\n", Fn);

    return 0;
}

```

运行结果如下：

5.8.3 穷举法应用

【例 5-13】百鸡问题是中国古代一道著名的数学问题，其原文如下：今有鸡翁一，值钱伍；鸡母一，值钱三；鸡雏三，值钱一。凡百钱买鸡百只，问鸡翁、母、雏各几何？

【解题思路】解决这样的数学问题，往往需要为其找到数学模型。这道题的数学模型不难建立。假设公鸡、母鸡、小鸡分别有 x 、 y 、 z 只，那么有

$$\begin{aligned} x+y+z &= 100 \\ 5x+3y+z/3 &= 100 \\ x, y, z &\geq 0 \end{aligned}$$

成立。这种带约束条件的不定方程组在组合数学中称为线性规划，而题目就是求这个线性规划的解。

线性规划的求解有严格的标准化过程，很适合用程序求解。但其方法过于繁复，这里用一种更加简明直观的求解方式，就是穷举法。穷举的意思是，将所有可能解列举出来，然后一个一个地去测试这些解中哪些真正地符合题目的要求。在本例中，一个解 (x, y, z) 由 3 个分量构成，其中每一个分量的取值范围都是从 0 到 100。因此，就从第一个可能解 $(0, 0, 0)$ 开始，将各分量的值代入到不定方程组中，测试它们是否满足约束条件：如果满足，则这是一组解；否则，使 z 分量自增，然后再一次测试，直到 z 的值取到 100 为止。此后 z 分量归零， y 分量自增，然后再将 z 从 0 测试到 100。此后依此类推，将分量的每一种组合都测试到，直到组合达到上限 $(100, 100, 100)$ 为止。可以看出，这在程序实现上需要用到循环，而且由于每一个分量都要用到一个循环，因此程序的主体应该由 3 重循环构成的，每一重循环的次数都是 100。以下是算法的主体部分：

```

for (x = 0; x <= 100; ++x)
    for (y = 0; y <= 100; ++y)
        for (z = 0; z <= 100; ++z)
            if (x + y + z == 100 && 5 * x + 3 * y + z / 3 == 100)
                输出(x, y, z);

```

算法虽然有了，但却不是最优的，有几处是可以优化的。

(1) 效率问题。算法中的总循环次数是 $100 \times 100 \times 100 = 100$ 万次，显然太多了。从题目的要求可知，因为公鸡 5 钱一只，而总共只有 100 钱，所以公鸡的最大数目是 $100/5=20$ 只，也就是说， x 的取值范围是 $0 \sim 20$ 。同样的道理，母鸡的最大数目是 $100/3=33$ 只，小鸡的最大数目是 100 只。这样一来，循环的总次数就只有 $20 \times 33 \times 100 = 66000$ 次，从而使优化后的算法整体运行时间是原始算法的 6.6%，效率得到了极大的提升。

(2) 除法运算问题。原始算法中有 $z/3$ 这样的表达式。大家知道，除法运算的效率比乘法要低，所以在程序中尽量避免除法运算。在本例中，这个除法运算是可以消除的。在等式的两边都乘以 3，可以将其改变为

```
15 * x + 9 * y + z == 300
```

至此，完成了算法的设计，而其正确性是可以保证的，程序设计如下。

```
//5-12.c
```

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

```
int main()
```

```
{
```

```
int x, y, z;
```

```
for (x = 0; x <= 20; ++x)
```

```
for (y = 0; y <= 33; ++y)
```

```
for (z = 0; z <= 100; ++z)
```

```
if (x + y + z == 100 && 15 * x + 9 * y + z == 300)
```

```
printf("rooster = %d, hen = %d, chick = %d\n", x, y, z);
```

```
return 0;
```

```
}
```

运行结果如下：

```
rooster = 0, hen = 25, chick = 75
```

```
rooster = 4, hen = 18, chick = 78
```

```
rooster = 8, hen = 11, chick = 81
```

```
rooster = 12, hen = 4, chick = 84
```