第3单元　重复结构

重复结构也称循环结构，是在一定条件下，让一条或一段程序重复地执行多遍。

3.1　C语言重复结构基础

3.1.1　C语言的三种重复结构

与人工计算相比，计算机的优势在于不怕烦琐、运算速度快。能够充分发挥这种优势的算法是重复。所以重复结构(或称循环结构)是程序最常见的结构。C语言提供的重复结构有如下3种。

(1) while结构。while结构的格式如下。

```
while (循环条件表达式)
    循环体代码
```

这种结构的控制机理如下。

① 进入条件：循环条件表达式为真。

② 进入后重复执行循环体代码，每一次执行循环体代码后都要对循环条件表达式进行一次测试。

③ 退出条件：循环条件表达式为假。

(2) do-while结构。do-while结构的格式如下。

```
do {
    循环体代码
} while (循环条件表达式);
```

这种重复结构的控制机理如下。

① 无条件进入。

② 进入后重复执行循环体代码，每一次执行循环体代码后都要对循环条件表达式进行一次测试。

③ 退出条件：循环条件表达式为假。

(3) for结构。for结构的格式如下。

```
for (表达式1; 表达式2; 表达式3)
    循环体代码
```

这种重复结构的控制机理如下。

① 执行表达式1后，若表达式2为真进入。

② 进入后重复执行循环体代码。每一次执行循环体代码后，都要执行一次表达式 3，并对表达式 2 进行一次测试。

③ 退出条件：表达式 2 为假。

图 3-1～图 3-3 所示为 C 语言 3 种重复结构的流程图。

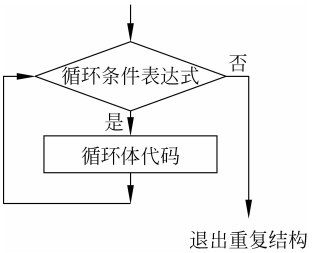

图 3-1　while 结构

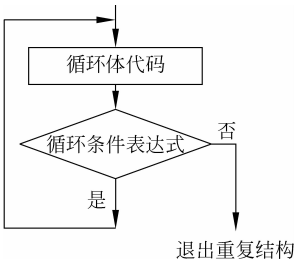

图 3-2　do-while 结构

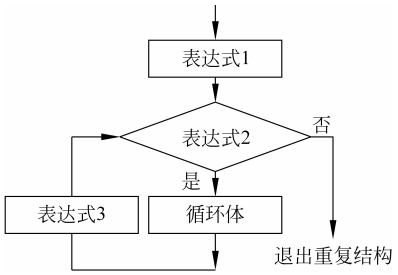

图 3-3　for 结构

3.1.2　累加器程序

设计一个程序，可以从键盘上输入 N 个学生的成绩，最后输出总成绩。

1. 问题分析

对于这个问题，首先可以粗略地给出下面的算法。

S1：重复地输入 N 个数据，并进行累加。

S2：输出总成绩。

由于要求和，就需要一个存放和的变量 sumScore。于是，S2 可以描述如下。

```
printf ("总成绩为:%f\n",sumScore);
```

这样，就只剩下如何进一步细化 S1 了。它执行的是重复地输入 N 个成绩，为此要使用重复结构，用 while 结构可以先粗略地描述如下。

```
int i=1;
while (i<=N) {
    S1.1:输入一个学生成绩;
    S1.2:累加学生成绩;
    i=i+1;                          /*计数一次,准备输入下个学生成绩          */
}
```

进一步可以描述如下。

```
int i=1;
float score,sumScore;
while (i<=N) {
    printf ("请输入一个学生成绩;");
    scanf ("%f",&score);
```

```
    sumScore+=score;                /*累加,相当于 sumScore=sumScore+score   */
    ++i;
}
```

在C语言中,可以把赋值表达式a=a+b简洁地写成a+=b,还可以进一步把表达式i=i+1写成更简洁的形式:++i。++称为增量(或自增)运算符。注意,增量运算符可以写在变量的前面(称为前缀形式),也可以写在变量的后面(称为后缀形式)。笼统地说,它们都是将一个变量的值增1,但细致地说,它们在实现过程上有所不同。请看下面的代码。

【代码3-1】 演示增量运算符的前缀形式和后缀形式的不同。

```
#include <stdio.h>
int main (void) {
    int a=5,b=5,c=5,d=5;

    printf ("a的初值:%d,",a);
    printf ("++a的立即值:%d,",++a);
    printf ("前缀自增后a的值:%d\n",a);

    printf ("b的初值:%d,",b);
    printf ("b ++的立即值:%d,",b ++);
    printf ("后缀自增后b的值:%d\n",b);

    return 0;
}
```

程序执行结果如下。

```
a的初值:5,++a的立即值:6,前缀自增后a的值:6
b的初值:5,b++的立即值:5,后缀自增后b的值:6
```

这个代码证明了自增运算符的前缀形式与后缀形式的不同:前缀自增是“立即增1”,即在当前的位置中增1,如上述a的值;后缀自增是“随后增1”,如上述b的值。至于后缀自增到何时才增,C标准没有规定,但在当前语句结束时一定是增1了。显然,后缀增量运算符会有这样的副作用。因此建议尽量使用前缀的增量运算符。

与增量运算符相对应的是减量(自减)运算符--,它每执行依次,值减1。

这里顺便要讨论一个问题:为什么代码3-1中对a和b都要分成3个语句呢?合在一个语句中行不行呢?这与函数参数执行的顺序有关。

【代码3-2】 验证函数参数执行的顺序。

```
#include <stdio.h>
int main (void) {
    int a=5,b=5;

    printf ("a的初值:%d,++a的立即值:%d,前缀自增后a的值:%d,",a,++a,a);
    printf ("下一条语句中的a值:%d\n",a);
```

```
    printf ("b 的初值:%d,b++的立即值:%d,后缀自增后 b 的值:%d,",b,b++,b);
    printf ("下一条语句中的 b 值:%d\n",b);

    return 0;
}
```

执行结果如下。

```
a的初值：6,++a的立即值：6,后缀自增后a的值：5,下一条语句中的a值：6
b的初值：5,b++的立即值：5,后缀自增后b的值：5,下一条语句中的b值：6
```

这里,为什么 a 自增前是 6,自增后反成了 5,而下一条语句又成了 6 呢?
这是因为,这里的编译器对函数参数的计算是从右向左进行的。

2. 程序代码

【代码 3-3】 用 while 结构实现的累加器代码。

```
#include <stdio.h>
#define N  8                                    //宏定义
int main (void) {
    int i=1;
    int i=1;
    float score,sumScore;

    while (i<=N) {
        printf ("请输入一个学生成绩:");
        scanf ("%f",&score);
        sumScore+=score;
        ++i;
    }

    printf ("总成绩为:%f\n",sumScore);
    return 0;
}
```

有以下几点说明。

(1) 程序中的“#define N 8”称为宏定义,N 称为宏。它与“#include <stdio.h>”等都是预处理命令。预处理命令是给编译器的命令,要求编译器在编译之前,首先完成某些处理。其中,宏定义命令要求编译器对程序中的某些字符串进行替换:用该命令中指出的后面的字符串替换前面的字符串。在本例中要求用 8 替换其程序中的 N。或者说,程序中的每个 N,在编译预处理时,都将被替换为 8。采用宏定义与在程序中直接写一个直接常量相比,优点在于修改便利,当程序的多个地方使用一个相同的值时,只修改宏即可。

进行宏定义要注意以下几点。

① 编译预处理命令后面没有分号,因为它们不是 C 语言的语句。

② 宏名要符合 C 语言标识符的命名规则。为了与变量相区别,一般使用大写。

③ 宏定义要放在函数的外面。

(2) 变量 i 的作用是控制重复的次数,称为循环变量。每重复一次,i 增 1,所以也称为计数器。

3. 测试用例设计——边值分析法

对于重复结构的程序进行路径覆盖测试往往是比较困难的。因为每一趟重复都是一个路径。若再有选择,将会使路径的数量更大幅度地增加。所以,对于重复结构的测试,一般要使用黑箱测试法。黑箱测试法主要进行程序外特性的测试。在 2.3.3 节中介绍了一种黑箱测试法——等价分类法。下面介绍另一种黑箱测试方法——边值分析法。它是在重复结构中应用比较多的一种测试方法。

边值分析法是对某些边界条件进行测试。所谓边界条件,是指输入等价类和输出等价类边缘上的数据。运用边值分析法应特别注意它与等价分类法的不同:边值分析不是从等价类中随便选一个例子作为代表,而是以等价类边界为测试的主要目标。因为大量经验证明,多数错误是发生在输入或输出范围的边界上,而不是发生在输入输出范围的内部。这种方法非常适合重复结构程序的测试。重复结构中的许多错误发生在循环次数的边界上,为此要注意下列边值条件。

(1) 重复初始边值条件,一般有以下几种。

① 零次重复,即不执行循环体。

② 一次重复,以便测试初始化方面的问题。

③ 二次重复,进一步揭露初始化方面的问题。

(2) 重复终止边值条件,一般有以下几种。

① 第 $n-1$ 次重复。

② 第 n 次重复。

③ 第 $n+1$ 次重复。

(3) 特殊重复次数,一般有以下几种。

① 属于给定循环次数之内的典型重复次数。

② 属于非正常情况下的典型重复次数。

本题的程序比较简单,它没有明确的重复终止边界条件,只有明确的重复初始条件。所以可以通过改变宏 N 的值,进行基于重复结构初始边界值的测试,即测试 N 为 0、1 和 2 的情况,从输入和输出对应的情况就可以发现有无错误。下面是 N 为 0 时的测试结果。

```
总成绩为：-107374176.000000
```

显然,本来应该是 0,却输出了一个莫名其妙的值。什么原因呢?

原来是由于变量 sumScore 没有初始化。一个变量没有初始化,其值是不确定的。这样后面在其基础上进行累加的结果也是不确定的,计算的平均值也是不确定的。因此,必须对其进行初始化。初始化为什么呢?对于累加器,初值一般为 0,对于累乘器初值一般为 1。

对于变量 score 来说,因为它只用来存放从键盘上输入的值。所以进不进行初始化没有什么影响。不过,一般来说,对变量进行初始化是程序员应当养成的好习惯。

【代码 3-4】 修改后的代码。

```
#include <stdio.h>
#define N  0
int main (void) {
    int i=1;
    float score=0,sumScore=0;            //变量初始化

    while (i<=N) {
        printf ("请输入一个学生成绩:");
        scanf ("%f",&score);
        sumScore+=score;
        ++i;
    }

    printf ("总成绩为:%f\n",sumScore);
    return 0;
}
```

重新测试结果如下。

```
总成绩为: 0.000000
```

下面再测试 N 为 1 和 2 的情形,结果如下。

```
请输入一个学生成绩:80.5
总成绩为: 80.500000
```

```
请输入一个学生成绩:80.5
请输入一个学生成绩:90.5
总成绩为: 171.000000
```

进行结果分析,没有发现错误。

4. 用 do-while 结构实现的累加器程序

【代码 3-5】 采用 do-while 结构的代码。

```
#include <stdio.h>
#define N  0
int main (void) {
    int i=1;
    float score=0,sumScore=0;            //变量初始化

    do {
        printf ("请输入一个学生成绩:");
        scanf ("%f",&score);
        sumScore+=score;
        ++i;
    } while (i<=N);

    printf ("总成绩为:%f\n", sumScore);
    return 0;
}
```

先用 N 为 0(即没有学生的情况)进行测试,得到结果如下。

```
请输入一个学生成绩:80.5
总成绩为: 80.500000
```

显然,这是不正确的。本来没有学生,不需要执行循环体,却有输入成绩的提示,用户于是输入了一个学生成绩。这是由于 do-while 结构要先无条件地执行一次循环体造成的。在这种情况下,使用 do-while 不合适。

5. 用 for 结构实现的累加器程序

【代码 3-6】 采用 for 结构的代码。

```
#include <stdio.h>
#define N  0
int main (void) {
    int i;
    float score=0,sumScore=0;            //变量初始化

    for (i=1; i<=N; ++i) {
        printf ("请输入一个学生成绩:");
        scanf ("%f",&score);
        sumScore+=score;
    }

    printf ("总成绩为:%f\n", sumScore);
    return 0;
}
```

测试结果略,没有发现错误。

6. 程序改进

在很多情况下,计数式的程序对用户往往会形成一种负担,要求用户时时刻刻要注意输入几个数据。这在数据量少时还可以,数据量多了就会很麻烦。如何可以不让用户老去计数呢?答案是可以采用结束标志。

【代码 3-7】 采用标志控制重复结构的结束,而不是用计数器。

```
#include <stdio.h>
int main (void) {
    float score=0,sumScore=0;

    do {
        sumScore+=score;
        printf ("请输入一个学生成绩(输入-1,程序结束):");
        scanf ("%f",&score);
    } while (score !=-1);                //用标志控制重复结构结束,而不是用计数器

    printf ("平均成绩为:%f\n", sumScore);
```

```
    return 0;
}
```

测试结果如下。

```
请输入一个学生成绩:80.5
请输入一个学生成绩:90.5
请输入一个学生成绩:70.5
请输入一个学生成绩:60.5
请输入一个学生成绩:-1
总成绩为: 302.000000
```

3.1.3 打印九九乘法表

打印一张如图 3-4 所示的九九乘法表。

1	2	3	4	5	6	7	8	9
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

图 3-4 一种九九乘法表

1. 问题分析

下面用逐步求精的方法分析本例的解法。

首先,把上述九九表分为三部分,表头(即 1～9 九个数字)、隔线、表体。于是,这个程序也可以分为如下三部分。

S1:打印表头。

S2:打印隔线。

S3:打印表体。

(1) 打印表头。

表头有 9 个数字 1,2,…,9。可以看成打印一个变量 i 的值,其初值为 1,每次加 1,直到 9 为止。这使用 for 结构最合适。设每个数字区占 4 个字符空间,则很容易写出 s1,代码如下。

```
for(i=1; i<=9;++i)
    printf ("%4d",i);
```

由于打印一行后,屏幕的光标会停留在一行的最后,为了使下一部分能从下一行开始显示,需要光标到下一行的起始位置。为此,在上面的程序段后需要增加一个换行操作。于是打印表头的代码修改如下。

```
for(i=1; i<=9; ++i)
    printf ("%4d",i);
printf ("\n");
```

(2) 打印隔线。

考虑隔线的总宽度与表头同宽,则可以用同样结构写出 s2,代码如下。

```
for(i=1; i <=36; ++i)
    printf ("%c", '-');
printf ("\n");
```

在上述两个程序段中,都使用 i 作循环变量。在 s2 中,i 只用于控制循环过程,称单纯

循环变量。在 s1 中,i 除用于控制循环过程外,还作为操作变量使用,即在循环体内还要用到它,对其进行操作,称为操作型循环变量。在使用单纯循环变量时,循环变量本身的具体值并不重要,重要的是通过循环变量控制循环执行的次数,只要循环变量的初值、终值和步长配合恰当即可。例如,打印一行隔线也可以写为如下形式。

```
for(i=101; i <=136; ++i)
    printf ("%c", '-');
```

还可以写为如下形式。

```
for (i=36; i>=1; --i)
    printf ("%c", '-');
```

或

```
for (i=10; i<=360; i+=10)
    printf ("%c", '-');
```

在众多的书写形式中,当然最易于理解的还是一开始写的那种形式。

(3) 打印表体。

表体共九行,所以首先考虑一个打印九行的算法 s3。

```
for(i=1; i <=9;++i)
    打印第 i 行
```

下面进一步考虑如何“打印第 i 行”。因为每行都有九个数字,故“打印第 i 行”可以写为如下形式。

```
for(j=1; j<=9; ++j)
    打印第 j 个数
printf ("\n");            //执行一个换行操作
```

“打印第 j 个数”即在第 i 行的第 j 列上打印一个数,大小为 i * j,占 4 个字宽。故可写为如下形式。

```
printf ("%4d", i * j);
```

至此,打印九九乘法表的算法已经全部细化为 C 代码了。

2. 程序代码与测试

【代码 3-8】 打印矩形九九乘法表程序。

```
#include <stdio.h>
int main (void) {
    int i,j;
    for (i=1; i<=9; ++i)
```

```
        printf ("%4d",i);
    printf ("\n");

    for (i=1; i<=36;++i)
        printf ("%c",'-');
    printf ("\n");

    for (i=1;i<=9;  ++i) {
        for (j=1; j<=9;++j)
            printf ("%4d",i*j);
        printf ("\n");
    }
    return 0;
}
```

这个程序的测试,可以采用结果分析法——看是否能打印出符合要求的九九乘法表。运行结果如下,符合要求。

```
   1   2   3   4   5   6   7   8   9
------------------------------------
   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
```

3. 打印三角形的九九乘法表

在上述矩形的九九乘法表中重复的内容比较多,一种简洁的九九乘法表是直角三角形的。三角形的九九乘法表有多种形式,采用哪种,就看打印表体时,变量 j 从什么位置开始,到什么位置结束,这两个位置与 1、变量 i 和 9 之间的关系。例如要打印从左上角的 1 到右下角的 81 画一条对角线,取其左下半部分形成的三角形,就是 j 从 1 开始到 i,即每一行从第 1 列开始打印 i 个数据,表体部分修改如下。

```
for (i=1;i<=9; ++i) {
    for (j=1; j<=i;++j)
        printf ("%4d",i*j);
    printf ("\n");
}
```

【代码 3-9】 打印三角形的九九乘法表程序。

```
#include <stdio.h>
int main (void) {
    int i,j;
    for (i=1; i<=9; ++i)
        printf ("%4d",i);
```