第 2 章　循环结构程序设计

学习目标

- ☑ 掌握 for 循环的使用方法
- ☑ 掌握 while 和 do-while 循环的使用方法
- ☑ 学会使用计数器和累加器
- ☑ 学会用输出中间结果的方法调试
- ☑ 学会用计时函数测试程序效率
- ☑ 学会用重定向的方式读写文件
- ☑ 学会用 fopen 的方式读写文件
- ☑ 了解算法竞赛对文件读写方式和命名的严格性
- ☑ 记住变量在赋值之前的值是不确定的
- ☑ 学会使用条件编译指示构建本地运行环境
- ☑ 学会用编译选项-Wall 获得更多的警告信息

第 1 章的程序虽然完善，但并没有发挥出计算机的优势。顺序结构程序自上到下只执行一遍，而分支结构中甚至有些语句可能一遍都执行不了。换句话说，为了让计算机执行大量操作，必须编写大量的语句。能不能只编写少量语句，就让计算机做大量的工作呢？这就是本章的主题。基本思路很简单：一条语句执行多次就可以了。但如何让这样的程序真正发挥作用，可不是一件容易的事。

2.1　for 循环

考虑这样一个问题：打印 1, 2, 3,…, 10，每个占一行。本着“解决问题第一”的思想，很容易写出程序：10 条 printf 语句就可以了。或者也可以写一条，每个数后面加一个“\n”换行符。但如果把 10 改成 100 呢？1000 呢？甚至这个重复次数是可变的：“输入正整数 n，打印 1, 2, 3,…, n，每个占一行。”又怎么办呢？这时可以使用 for 循环。

程序 2-1　输出 1,2,3,…,n 的值

```
1  #include<stdio.h>
2  int main()
3  {
4    int n;
5    scanf("%d", &n);
6    for(int i = 1; i <= n; i++)
7      printf("%d\n", i);
```

```
8    return 0;
9  }
```

暂时不用考虑细节，只要知道它是“让 i 依次等于 1, 2, 3,…, n，每次都执行 printf("%d\n", i);”即可。这个“依次”非常重要：程序运行结果一定是 1,2,3,…,n，而不是别的顺序。

提示 2-1：for 循环的格式为：for(初始化; 条件; 调整)　循环体;

在刚才的例子中，初始化语句是“int i = 1”。这是一条声明+赋值的语句，含义是声明一个新的变量 i，然后赋值为 1。循环条件是“i≤n”，当循环条件满足时始终进行循环。调整方法是 i++，其含义和 i = i+1 相同——表示给 i 增加 1。循环体是语句“printf("%d\n", i);”，这就是计算机反复执行的内容。注意循环变量的妙用：尽管每次执行的语句相同，但是由于 i 的值不断变化，该语句的输出结果也是不断变化的。

提示 2-2：尽管 for 循环反复执行相同的语句，但这些语句每次的执行效果往往不同。

为了更深入地理解 for 循环，下面给出了程序 2-1 的执行过程。

当前行：5。scanf 请求键盘输入，假设输入 4。此时变量 n=4，继续。
当前行：6。这是第一次执行到该语句，执行初始化语句 int i=1。条件 i≤n 满足，继续。
当前行：7。由于 i=1，在屏幕输出 1 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=2，n=4，条件 i≤n 满足，继续。
当前行：7。由于 i=2，在屏幕输出 2 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=3，n=4，条件 i≤n 满足，继续。
当前行：7。由于 i=3，在屏幕输出 3 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=4，n=4，条件 i≤n 满足，继续。
当前行：7。由于 i=4，在屏幕输出 4 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=5，n=4，条件 i≤n 不满足，跳出循环体。
当前行：8。程序结束。

这个执行过程对于理解 for 循环非常重要：语句是一条一条执行的。强烈建议教师在课堂上演示单步调试的方法，并打开 i 和 n 的 watch 功能，以帮助学生掌握如何用实验验证上面所介绍的执行过程。观察执行过程时应留意两个方面：“当前行”的跳转（在 IDE 中往往高亮显示），以及变量的变化。这二者也是编码、测试和调试的重点。根据实际情况，教师可以用 IDE（如 Code::Blocks）或者文本界面的 gdb 进行演示。gdb 的简明参考见附录 A。

提示 2-3：编写程序时，要特别留意“当前行”的跳转和变量的改变。

上面的代码里还有一个重要的细节：变量 i 定义在循环语句中，因此 i 在循环体内不可见，例如，在第 8 行之前再插入一条“printf("%d\n", i);”会报错[①]。有经验的程序员总是尽量缩小变量定义的范围，当写了足够多的程序之后，这样做的优点会慢慢表现出来。

提示 2-4：建议尽量缩短变量的定义范围。例如，在 for 循环的初始化部分定义循环变量。

[①] Visual C++ 6.0 等早期编译器允许在循环体之后访问 i，但这样，如果再写一个“for(int i = 0; i < n; i++)”则会出现 i 重定义的错误。

有了 for 循环，可以解决一些简单的问题。

例题 2-1　aabb

输出所有形如 aabb 的 4 位完全平方数（即前两位数字相等，后两位数字也相等）。

【分析】

分支和循环结合在一起时功能强大：下面枚举所有可能的 aabb，然后判断它们是否为完全平方数。注意，a 的范围是 1~9，但 b 可以是 0。主程序如下：

```
for(int a = 1; a <= 9; a++)
  for(int b = 0; b <= 9; b++)
    if(aabb 是完全平方数) printf("%d\n", aabb);
```

请注意，这里用到了循环的嵌套：for 循环的循环体自身又是一个循环。如果难以理解嵌套循环，可以用前面介绍的方法——在 IDE 或 gdb 中单步执行，观察“当前行”和循环变量 a 和 b 的变化过程。

上面的程序并不完整——“aabb 是完全平方数”是中文描述，而不是合法的 C 语言表达式，而 aabb 在 C 语言中也是另外一个变量，而不是把两个数字 a 和两个数字 b 拼在一起（C 语言中的变量名可以由多个字母组成）。但这个“程序”很容易理解，甚至能让读者的思路更加清晰。

这里把这样“不是真正程序”的“代码”称为伪代码（pseudocode）。虽然有一些正规的伪代码定义，但在实际应用中，并不需要太拘泥于伪代码的格式。主要目标是描述算法梗概，避开细节，启发思路。

提示 2-5：*不拘一格地使用伪代码来思考和描述算法是一种值得推荐的做法。*

写出伪代码之后，我们需要考虑如何把它变成真正的代码。上面的伪代码有两个“非法”的地方：完全平方数判定，以及 aabb 这个变量。后者相对比较容易：用另外一个变量 n = a*1100 + b*11 存储即可。

提示 2-6：*把伪代码改写成代码时，一般先选择较为容易的任务来完成。*

接下来的问题就要困难一些了：如何判断 n 是否为完全平方数？第 1 章中用过“开平方”函数，可以先求出其平方根，然后看它是否为整数，即用一个 int 型变量 m 存储 sqrt(n)四舍五入后的整数，然后判断 m^2 是否等于 n。函数 floor(x)返回不超过 x 的最大整数。完整程序如下：

程序 2-2　7744 问题（1）

```
#include<stdio.h>
#include<math.h>
int main()
{
  for(int a = 1; a <= 9; a++)
    for(int b = 0; b <= 9; b++)
    {
      int n = a*1100 + b*11; //这里才开始使用 n，因此在这里定义 n
```

```
        int m = floor(sqrt(n) + 0.5);
        if(m*m == n) printf("%d\n", n);
    }
    return 0;
}
```

读者可能会问：可不可以这样写？if(sqrt(n) == floor(sqrt(n))) printf("%d\n", n)，即直接判断 sqrt(n)是否为整数。理论上当然没问题，但这样写不保险，因为浮点数的运算（和函数）有可能存在误差。

假设在经过大量计算后，由于误差的影响，整数 1 变成了 0.9999999999，floor 的结果会是 0 而不是 1。为了减小误差的影响，一般改成四舍五入，即 floor(x+0.5)[①]。如果难以理解，可以想象成在数轴上把一个单位区间往左移动 0.5 个单位的距离。floor(x)等于 1 的区间为[1,2)，而 floor(x+0.5)等于 1 的区间为[0.5, 1.5)。

提示 2-7：*浮点运算可能存在误差。在进行浮点数比较时，应考虑到浮点误差。*

另一个思路是枚举平方根 x，从而避免开平方操作。

程序 2-3　7744 问题（2）

```
#include<stdio.h>
int main()
{
    for(int x = 1; ; x++)
    {
        int n = x * x;
        if(n < 1000) continue;
        if(n > 9999) break;
        int hi = n / 100;
        int lo = n % 100;
        if(hi/10 == hi%10 && lo/10 == lo%10) printf("%d\n", n);
    }
    return 0;
}
```

此程序中的新知识是 continue 和 break 语句。continue 是指跳回 for 循环的开始，执行调整语句并判断循环条件（即“直接进行下一次循环”），而 break 是指直接跳出循环[②]。

这里的 continue 语句的作用是排除不足四位数的 n，直接检查后面的数。当然，也可以直接从 x=32 开始枚举，但是 continue 可以帮助我们偷懒：不必求出循环的起始点。有了 break，连循环终点也不必指定——当 n 超过 9999 后会自动退出循环。注意，这里是“退出循环”

[①] 这样做，小数部分为 0.5 的数也会受到浮点误差的影响，因此任何一道严密的算法竞赛题目中都需要想办法解决这个问题。后面还会讨论这个问题。

[②] 逻辑与“&&”似乎也没有出现过，但假设读者在学习后已经翻阅了相关资料，或者教师已经给学生补充了这个运算符。如果确实没有学过，现在学也来得及。

而不是“继续循环”（想一想，为什么），可以把 break 换成 continue 加以验证。

另外，注意到这里的 for 语句是“残缺”的：没有指定循环条件。事实上，3 部分都是可以省略的。没错，for(;;) 就是一个死循环，如果不采取措施（如 break），就永远不会结束。

2.2 while 循环和 do-while 循环

例题 2-2 3*n*+1 问题

猜想[①]：对于任意大于 1 的自然数 n，若 n 为奇数，则将 n 变为 $3n+1$，否则变为 n 的一半。经过若干次这样的变换，一定会使 n 变为 1。例如，3→10→5→16→8→4→2→1。

输入 n，输出变换的次数。$n \leqslant 10^9$。

样例输入：

```
3
```

样例输出：

```
7
```

【分析】

不难发现，程序完成的工作依然是重复性的：要么乘 3 加 1，要么除以 2，但和 2.1 节的程序又不太一样：循环的次数是不确定的，而且 n 也不是“递增”式的循环。这样的情况很适合用 while 循环来实现。

程序 2-4 3*n*+1 问题（有 bug）

```
#include<stdio.h>
int main()
{
  int n, count = 0;
  scanf("%d", &n);
  while(n > 1)
  {
    if(n % 2 == 1) n = n*3+1;
    else n /= 2;
    count++;
  }
  printf("%d\n", count);
  return 0;
}
```

上面的程序有好几个值得注意的地方。首先是“=0”，意思是定义整型变量 count 的同时初始化为 0。接下来是 while 语句。

① http://en.wikipedia.org/wiki/3n+1。

提示 2-8：while 循环的格式为“while(条件) 循环体;”。

此格式看上去比 for 循环更简单，可以用 while 改写 for。“for(初始化; 条件; 调整) 循环体;”等价于：

```
初始化;
while(条件)
{
  循环体;
  调整;
}
```

建议读者再次利用 IDE 或者 gdb 跟踪调试，看看执行流程是怎样的。

n /=2 的含义是 n = n / 2，类似于前面介绍过的 i++。很多运算符都有类似的用法，例如，a *= 3 表示 a = a * 3。

count++的作用是计数器。由于最终输出的是变换的次数，需要一个变量来完成计数。

提示 2-9：当需要统计某种事物的个数时，可以用一个变量来充当计数器。

这个程序是否正确？先来测试一下：输入“987654321”，看看结果是什么。很不幸，答案等于 1——这明显是错误的。题目中给出的范围是 $n \leqslant 10^9$，这个 987654321 是合法的输入数据。

提示 2-10：不要忘记测试。一个看上去正确的程序可能隐含错误。

问题出在哪里呢？若反复阅读程序仍然无法找到答案，就动手实验吧！一种方法是利用 IDE 和 gdb 跟踪调试，但这并不是本书所推荐的调试方法。一个更通用的方法是：输出中间结果。

提示 2-11：在观察无法找出错误时，可以用“输出中间结果”的方法查错。

在给 n 做变换的语句后加一条输出语句 printf("%d\n", n)，将很快找到问题的所在：第一次输出为-1332004332，它不大于 1，所以循环终止。如果认真完成了前面的所有探索实验，读者将立刻明白这其中的缘由：乘法溢出了。

下面稍微回顾一下数据类型的大小。在第 1 章中，通过实验得出了 int 整数的大小——很可能是-2147483648~2147483647，即-2^{31}~$2^{31}-1$。为什么叫“很可能”呢，因为 C99 中只规定了 int 至少是 16 位，却没有规定具体值[①]。是不是感觉有些别扭？的确如此，所以 C99 规定了一些固定长度的整数，例如 int32_t、uint32_t[②]。

好在算法竞赛的平台相对稳定，目前几乎在所有的比赛平台上，int 都是 32 位整数。

提示 2-12：C99 并没有规定 int 类型的确切大小，但在当前流行的竞赛平台中，int 都是 32 位整数，范围是-2147483648~2147483647。

① 在笔者中学时期，int 一般是 16 位的，即-32768~32767。
② uint32_t 表示无符号 32 位整数，范围是 0~4294967296。

回到本题。本题中 n 的上限 10^9 只比 int 的上界稍微小一点，因此溢出了也并不奇怪。只要使用 C99 中新增的 long long 即可解决问题，其范围是 -2^{63}~$2^{63}-1$，唯一的区别就是要把输入时的%d 改成%lld。但这也是不保险的——在 MinGW 的 gcc[①]中，要把%lld 改成%I64d，但奇怪的是 VC2008 里又得改回%lld。是不是很容易搞错？所以如果涉及 long long 的输入输出，常用 C++的输入输出流或者自定义的输入输出方法，本书将在后面的章节对其进行深入讨论。

提示 2-13：long long 在 Linux 下的输入输出格式符为%lld，但 Windows 平台中有时为%I64d。为保险起见，可以用后面介绍的 C++流，或者编写自定义输入输出函数。

最后给出 long long 版本的代码，它避开了对 long long 的输入输出，并且成功算出 n=987654321 时的答案为 180。

程序 2-5　3n+1 问题

```
#include<stdio.h>
int main()
{
  int n2, count = 0;
  scanf("%d", &n2);
  long long n = n2;
  while(n > 1)
  {
    if(n % 2 == 1) n = n*3+1;
    else n /= 2;
    count++;
  }
  printf("%d\n", count);
  return 0;
}
```

例题 2-3　近似计算

计算 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$，直到最后一项小于 10^{-6}。

【分析】

本题和例题 2-2 一样，也是重复计算，因此可以用循环实现。但不同的是，只有算完一项之后才知道它是否小于 10^{-6}。也就是说，循环终止判断是在计算之后，而不是计算之前。这样的情况很适合使用 do-while 循环。

程序 2-6　近似计算

```
#include<stdio.h>
int main() {
  double sum = 0;
```

[①] 这并不是 MinGW 引起的，而是因为 Windows 的 CRT（C Runtime）。

```
    for(int i = 0; ; i++) {
      double term = 1.0 / (i*2+1);
      if(i % 2 == 0) sum += term;
      else sum -= term;
      if(term < 1e-6) break;
    }
    printf("%.6f\n", sum);
    return 0;
}
```

提示 2-14：do-while 循环的格式为“do { 循环体 }while(条件);”，其中循环体至少执行一次，每次执行完循环体后判断条件，当条件满足时继续循环。

2.3　循环的代价

例题 2-4　阶乘之和

输入 n，计算 $S = 1! + 2! + 3! + \cdots + n!$的末 6 位（不含前导 0）。$n \leqslant 10^6$，$n!$表示前 n 个正整数之积。

样例输入：

```
10
```

样例输出：

```
37913
```

【分析】

这个任务并不难，引入累加变量 S 之后，核心算法只有“for(int i = 1; i <= n; i++)S+= i!”。不过，C 语言并没有阶乘运算符，所以这句话只是伪代码，而不是真正的代码。事实上，还需要一次循环来计算 i!，即“for(int j = 1; j <= i; j++)　factorial *= j;”。代码如下：

程序 2-7　阶乘之和（1）

```
#include<stdio.h>
int main()
{
  int n, S = 0;
  scanf("%d", &n);
  for(int i = 1; i <= n; i++)
  {
    int factorial = 1;
    for(int j = 1; j <= i; j++)
      factorial *= j;
```

```
    S += factorial;
  }
  printf("%d\n", S % 1000000);
  return 0;
}
```

注意累乘器 factorial（英文“阶乘”的意思）定义在循环里面。换句话说，每执行一次循环体，都要重新声明一次 factorial，并初始化为 1（想一想，为什么不是 0）。因为只要末 6 位，所以输出时对 10^6 取模。

提示 2-15：*在循环体开始处定义的变量，每次执行循环体时会重新声明并初始化。*

有了刚才的经验，下面来测试一下这个程序：n=100 时，输出−961703。直觉告诉我们：乘法又溢出了。这个直觉很容易通过“输出中间变量”法得到验证，但若要解决这个问题，还需要一点数学知识。

提示 2-16：*要计算只包含加法、减法和乘法的整数表达式除以正整数 n 的余数，可以在每步计算之后对 n 取余，结果不变。*

在修正这个错误之前，还可以进行更多测试：当 n=10^6 时输出什么？更会溢出不是吗？但是重点不在这里。事实上，它的速度太慢！下面把程序改成“每步取模”的形式，然后加一个“计时器”，看看究竟有多慢。

程序 2-8　阶乘之和（2）

```
#include<stdio.h>
#include<time.h>
int main()
{
  const int MOD = 1000000;
  int n, S = 0;
  scanf("%d", &n);
  for(int i = 1; i <= n; i++)
  {
    int factorial = 1;
    for(int j = 1; j <= i; j++)
      factorial = (factorial * j % MOD);
    S = (S + factorial) % MOD;
  }
  printf("%d\n", S);
  printf("Time used = %.2f\n", (double)clock() / CLOCKS_PER_SEC);
  return 0;
}
```

上面的程序再次使用到了常量定义，好处是可以在程序中使用代号 MOD 而不是常数

1000000，改善了程序的可读性，也方便修改（假设题目改成求末 5 位正整数之积）。

这个程序真正的特别之处在于计时函数 clock()的使用。该函数返回程序目前为止运行的时间。这样，在程序结束之前调用此函数，便可获得整个程序的运行时间。这个时间除以常数 CLOCKS_PER_SEC 之后得到的值以“秒”为单位。

提示 2-17：*可以使用 time.h 和 clock()函数获得程序运行时间。常数 CLOCKS_PER_SEC 和操作系统相关，请不要直接使用 clock()的返回值，而应总是除以 CLOCKS_PER_SEC。*

输入“20”，按 Enter 键后，系统瞬间输出了答案 820313。但是，输出的 Time used 居然不是 0！其原因在于，键盘输入的时间也被计算在内——这的确是程序启动之后才进行的。为了避免输入数据的时间影响测试结果，可使用一种称为“管道”的小技巧：在 Windows 命令行下执行 echo 20 | abc，操作系统会自动把 20 输入，其中 abc 是程序名[①]。如果不知道如何操作命令行，请参考附录 A。笔者建议每个读者都熟悉命令行操作，包括 Windows 和 Linux。

在尝试了多个 n 之后，得到了一张表，如表 2-1 所示。

表 2-1　程序 2-8 的输出结果与运行时间表

n	20	40	80	160	1600	6400	12800	25600	51200
答案	820313	940313	940313	940313	940313	940313	940313	940313	940313
时间	<0.01	<0.01	<0.01	<0.01	0.05	0.70	2.70	11.08	43.72

由表 2-1 可知：第一，程序的运行时间大致和 n 的平方成正比（因为 n 每扩大 1 倍，运行时间近似扩大 4 倍）。甚至可以估计 $n=10^6$ 时，程序大致需要近 5 个小时才能执行完。

提示 2-18：*很多程序的运行时间与规模 n 存在着近似的简单关系。可以通过计时函数来发现或验证这一关系。*

第二，从 40 开始，答案始终不变。这是真理还是巧合？聪明的读者也许已经知道了：25！末尾有 6 个 0，所以从第 5 项开始，后面的所有项都不会影响和的末 6 位数字——只需要在程序的最前面加一条语句“if(n>25)　n = 25;”，效率和溢出都将不存在问题。

本节展示了循环结构程序设计中最常见的两个问题：算术运算溢出和程序效率低下。这两个问题都不是那么容易解决的，将在后面章节中继续讨论。另外，本节中介绍的两个工具——输出中间结果和计时函数，都是相当实用的。

2.4　算法竞赛中的输入输出框架

例题 2-5　数据统计

输入一些整数，求出它们的最小值、最大值和平均值（保留 3 位小数）。输入保证这些数都是不超过 1000 的整数。

[①] Linux 下需要输入“echo | ./abc”，因为在默认情况下，当前目录不在可执行文件的搜索路径中。

样例输入：

```
2 8 3 5 1 7 3 6
```

样例输出：

```
1 8 4.375
```

【分析】

如果是先输入整数 n，然后输入 n 个整数，相信读者能够写出程序。关键在于：整数的个数是不确定的。下面直接给出程序：

程序 2-9　数据统计（有 bug）

```
#include<stdio.h>
int main()
{
  int x, n = 0, min, max, s = 0;
  while(scanf("%d", &x) == 1)
  {
    s += x;
    if(x < min) min = x;
    if(x > max) max = x;
    n++;
  }
  printf("%d %d %.3f\n", min, max, (double)s/n);
  return 0;
}
```

看看这个程序多了些什么内容？scanf 函数有返回值？对，它返回的是成功输入的变量个数，当输入结束时，scanf 函数无法再次读取 x，将返回 0。

下面进行测试。输入“2 8 3 5 1 7 3 6”，按 Enter 键，但未显示结果。难道程序速度太慢？其实程序正在等待输入。还记得 scanf 的输入格式吗？空格、TAB 和回车符都是无关紧要的，所以按 Enter 键并不意味着输入的结束。那如何才能告诉程序输入结束了呢？

提示 2-19：在 Windows 下，输入完毕后先按 Enter 键，再按 Ctrl+Z 键，最后再按 Enter 键，即可结束输入。在 Linux 下，输入完毕后按 Ctrl+D 键即可结束输入。

输入终于结束了，但输出却是“1 2293624 4.375”。这个 2293624 是从何而来？当用-O2 编译（读者可阅读附录 A 了解-O2）后答案变成了 1 10 4.375，和刚才不一样！换句话说，这个程序的运行结果是不确定的。在读者自己的机器上，答案甚至可能和上述两个都不同。

根据“输出中间结果”的方法，读者不难验证下面的结论：变量 max 在一开始就等于 2293624（或者 10），自然无法更新为比它小的 8。

提示 2-20：变量在未赋值之前的值是不确定的。特别地，它不一定等于 0。

解决的方法就很清楚了：在使用之前赋初值。由于 min 保存的是最小值，其初值应该是一个很大的数；反过来，max 的初值应该是一个很小的数。一种方法是定义一个很大的常数，如 INF = 1000000000，然后让 max=-INF，而 min=INF，另一种方法是先读取第一个整数 x，然后令 max = min = x。这样的好处是避免了人为的“假想无穷大”值，程序更加优美；而 INF 这样的常数有时还会引起其他问题，如“无限大不够大”，或者“运算溢出”，后面还会继续讨论这个问题。

上面的程序并不是很方便：每次测试都要手动输入许多数。尽管可以用前面讲的管道的方法，但数据只是保存在命令行中，仍然不够方便。

一个好的方法是用文件——把输入数据保存在文件中，输出数据也保存在文件中。这样，只要事先把输入数据保存在文件中，就不必每次重新输入了；数据输出在文件中也避免了“输出太多，一卷屏前面的就看不见了”这样的尴尬，运行结束后，慢慢浏览输出文件即可。如果有标准答案文件，还可以进行文件比较[①]，而无须编程人员逐个检查输出是否正确。事实上，几乎所有算法竞赛的输入数据和标准答案都是保存在文件中的。

使用文件最简单的方法是使用输入输出重定向，只需在 main 函数的入口处加入以下两条语句：

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

上述语句将使得 scanf 从文件 input.txt 读入，printf 写入文件 output.txt。事实上，不只是 scanf 和 printf，所有读键盘输入、写屏幕输出的函数都将改用文件。尽管这样做很方便，并不是所有算法竞赛都允许用程序读写文件。甚至有的竞赛允许访问文件，但不允许用 freopen 这样的重定向方式读写文件。参赛之前请仔细阅读文件读写的相关规定。

提示 2-21：*请在比赛之前了解文件读写的相关规定：是标准输入输出（也称标准 I/O，即直接读键盘、写屏幕），还是文件输入输出？如果是文件输入输出，是否禁止用重定向方式访问文件？*

多年来，无数选手因文件相关问题丢掉了大量分数。一个普适的原则是：详细阅读比赛规定，并严格遵守。例如，输入输出文件名和程序名往往都有着严格规定，不要弄错大小写，不要拼错文件名，不要使用绝对路径或相对路径。

例如，如果题目规定程序名称为 test，输入文件名为 test.in，输出文件名为 test.out，就不要犯以下错误。

错误 1：程序存为 t1.c（应该改成 test.c）。

错误 2：从 input.txt 读取数据（应该从 test.in 读取）。

错误 3：从 tset.in 读取数据（拼写错误，应该从 test.in 读取）。

错误 4：数据写到 test.ans（扩展名错误，应该是 test.out）。

错误 5：数据写到 c:\\contest\\test.out（不能加路径，哪怕是相对路径。文件名应该只有

[①] 在 Windows 中可以使用 fc 命令，而在 Linux 中可以使用 diff 命令。

8个字符：test.out）。

提示 2-22：*在算法竞赛中，选手应严格遵守比赛的文件名规定，包括程序文件名和输入输出文件名。不要弄错大小写，不要拼错文件名，不要使用绝对路径或相对路径。*

当然，这些错误都不是选手故意犯下的。前面说过，利用文件是一种很好的自我测试方法，但如果比赛要求采用标准输入输出，就必须在自我测试完毕之后删除重定向语句。选手比赛时一紧张，就容易忘记将其删除。

有一种方法可以在本机测试时用文件重定向，但一旦提交到比赛，就自动“删除”重定向语句。代码如下：

程序 2-10　数据统计（重定向版）

```
#define LOCAL
#include<stdio.h>
#define INF 1000000000
int main()
{
#ifdef LOCAL
  freopen("data.in", "r", stdin);
  freopen("data.out", "w", stdout);
#endif
  int x, n = 0, min = INF, max = -INF, s = 0;
  while(scanf("%d", &x) == 1)
  {
    s += x;
    if(x < min) min = x;
    if(x > max) max = x;
/*
    printf("x = %d, min = %d, max = %d\n", x, min, max);
*/
    n++;
  }
  printf("%d %d %.3f\n", min, max, (double)s/n);
  return 0;
}
```

这是一份典型的比赛代码，包含了几个特殊之处：

- 重定向的部分被写在了#ifdef 和#endif 中。其含义是：只有定义了符号 LOCAL，才编译两条 freopen 语句。
- 输出中间结果的 printf 语句写在了注释中——它在最后版本的程序中不应该出现，但是又舍不得删除它（万一发现了新的 bug，需要再次用它输出中间信息）。将其注释的好处是：一旦需要时，把注释符去掉即可。

上面的代码在程序首部就定义了符号 LOCAL，因此在本机测试时使用重定向方式读写文件。如果比赛要求读写标准输入输出，只需在提交之前删除#define LOCAL 即可。一个更好的方法是在编译选项而不是程序里定义这个 LOCAL 符号（不知道如何在编译选项里定义符号的读者请参考附录 A），这样，提交之前不需要修改程序，进一步降低了出错的可能。

提示 2-23：*在算法竞赛中，有经验的选手往往会使用条件编译指令并且将重要的测试语句注释掉而非删除。*

如果比赛要求用文件输入输出，但禁止用重定向的方式，又当如何呢？程序如下：

程序 2-11　数据统计（fopen 版）

```
#include<stdio.h>
#define INF 1000000000
int main()
{
  FILE *fin, *fout;
  fin = fopen("data.in", "rb");
  fout = fopen("data.out", "wb");
  int x, n = 0, min = INF, max = -INF, s = 0;
  while(fscanf(fin, "%d", &x) == 1)
  {
    s += x;
    if(x < min) min = x;
    if(x > max) max = x;
    n++;
  }
  fprintf(fout, "%d %d %.3f\n", min, max, (double)s/n);
  fclose(fin);
  fclose(fout);
  return 0;
}
```

虽然新内容不少，但也很直观：先声明变量 fin 和 fout（暂且不用考虑 FILE *），把 scanf 改成 fscanf，第一个参数为 fin；把 printf 改成 fprintf，第一个参数为 fout，最后执行 fclose，关闭两个文件。

提示 2-24：*在算法竞赛中，如果不允许使用重定向方式读写数据，应使用 fopen 和 fscanf/fprintf 进行输入输出。*

重定向和 fopen 两种方法各有优劣。重定向的方法写起来简单、自然，但是不能同时读写文件和标准输入输出；fopen 的写法稍显繁琐，但是灵活性比较大（例如，可以反复打开并读写文件）。顺便说一句，如果想把 fopen 版的程序改成读写标准输入输出，只需赋值“fin =

stdin; fout = stdout;”即可，不要调用 fopen 和 fclose[①]。

对文件输入输出的讨论到此结束，本书剩余部分的所有题目均使用标准输入输出。

例题 2-6　数据统计 II

输入一些整数，求出它们的最小值、最大值和平均值（保留 3 位小数）。输入保证这些数都是不超过 1000 的整数。

输入包含多组数据，每组数据第一行是整数个数 n，第二行是 n 个整数。n=0 为输入结束标记，程序应当忽略这组数据。相邻两组数据之间应输出一个空行。

样例输入：

```
8
2 8 3 5 1 7 3 6
4
-4 6 10 0
0
```

样例输出：

```
Case 1: 1 8 4.375

Case 2: -4 10 3.000
```

【分析】

本题和例题 2-5 本质相同，但是输入输出方式有了一定的变化。由于这样的格式在算法竞赛中非常常见，这里直接给出代码：

程序 2-12　数据统计 II（有 bug）

```
#include<stdio.h>
#define INF 1000000000
int main()
{
  int x, n = 0, min = INF, max = -INF, s = 0, kase = 0;
  while(scanf("%d", &n) == 1 && n)
  {
    int s = 0;
    for(int i = 0; i < n; i++) {
      scanf("%d", &x);
      s += x;
      if(x < min) min = x;
      if(x > max) max = x;
    }
    if(kase) printf("\n");
```

[①] 有读者可能试过用 fopen("con", "r")的方法打开标准输入输出，但这个方法并不是可移植的——它在 Linux 下是无效的。

```
    printf("Case %d: %d %d %.3f\n", ++kase, min, max, (double)s/n);
  }
  return 0;
}
```

聪明的读者，你能看懂其中的逻辑吗？上面的程序有几个要点。首先是输入循环。题目说了n=0为输入标记，为什么还要判断scanf的返回值呢？答案是为了鲁棒性(robustness)。算法竞赛中题目的输入输出是人设计的，难免会出错。有时会出现题目指明以 n=0 为结束标记而真实数据忘记以 n=0 结尾的情形。虽然比赛中途往往会修改这一错误，但在ACM/ICPC 等时间紧迫的比赛中，如果程序能自动处理好有瑕疵的数据，会节约大量不必要的时间浪费。

提示 2-25：*在算法竞赛中，偶尔会出现输入输出错误的情况。如果程序鲁棒性强，有时能在数据有瑕疵的情况下仍然给出正确的结果。程序的鲁棒性在工程中也非常重要。*

下一个要点是 kase 变量的使用。不难看出它是“当前数据编号”计数器。当输出第 2 组或以后的结果时，会在前面加一个空行，符合题目“相邻两组数据的输出以空行隔开”的规定。注意，最后一组数据的输出会以回车符结束，但之后不会有空行。不同的题目会有不同的规定，请读者仔细阅读题目。

像本题这样“多组数据”的题目数不胜数。例如，ACM/ICPC 总决赛就只有一个输入文件，包含多组数据。即使是 NOI/IOI 这样多输入文件的比赛，有时也会出现一个文件多组数据的情况。例如，有的题目输出只有 Yes 和 No 两种，如果一个文件里只有一组数据，又是每个文件分别给分，一个随机输出 Yes/No 的程序平均情况下能得 50 分，而一个把 Yes 打成 yes, No 打成 no 的程序却只有 0 分[①]。

接下来是找 bug 时间。上面的程序对于样例输入输出可以得到正确的结果，但它真的是正确的吗？在样例输入的最后增加第 3 组数据：1 0，会看到这样的输出：

Case 3: -4 10 0.000

相信读者已经意识到问题出在哪里了：min 和 max 没有“重置”，仍然是上个数据结束后的值。

提示 2-26：*在多数据的题目中，一个常见的错误是：在计算完一组数据后某些变量没有重置，影响到下组数据的求解。*

解决方法很简单，把 min 和 max 定义在 while 循环中即可，这样每次执行循环体时，会新声明和初始化 min 和 max。细心的读者也许注意到了另外一个问题：为什么第 3 个数（累加和）是对的呢？原因在于：循环体内部也定义了一个 s，把 main 函数里定义的 s 给“屏蔽”了。

提示 2-27：*当嵌套的两个代码块中有同名变量时，内层的变量会屏蔽外层变量，有时会引起十分隐蔽的错误。*

[①] 也不总是如此。有些比赛会善意地把这种只是格式不对的结果判成“正确”。可惜这样的比赛非常少。

这是初学者在求解“多数据输入”的题目时常范的错误，请读者留意。这种问题通常很隐蔽，但也不是发现不了：对于这个例子来说，编译时加一个-Wall就会看到一条警告：warning: unused variable 's' [-Wunused-variable]（警告：没有用过的变量's'）。

提示2-28：用编译选项-Wall编译程序时，会给出很多（但不是所有）警告信息，以帮助程序员查错。但这并不能解决所有的问题：有些“错误”程序是合法的，只是这些动作不是所期望的。

2.5 注解与习题

不知不觉，本章已经开始出现一些挑战了。尽管难度不算太高，本章的例题和习题已经出现了真正的竞赛题目——仅使用简单变量和基本的顺序、分支与循环结构就可以解决很多问题。在继续前进之前，请认真总结，并且完成习题。

2.5.1 习题

习题2-1 水仙花数（daffodil）

输出100~999中的所有水仙花数。若3位数ABC满足$ABC=A^3+B^3+C^3$，则称其为水仙花数。例如$153=1^3+5^3+3^3$，所以153是水仙花数。

习题2-2 韩信点兵（hanxin）

相传韩信才智过人，从不直接清点自己军队的人数，只要让士兵先后以三人一排、五人一排、七人一排地变换队形，而他每次只掠一眼队伍的排尾就知道总人数了。输入包含多组数据，每组数据包含3个非负整数a,b,c，表示每种队形排尾的人数（$a<3$，$b<5$，$c<7$），输出总人数的最小值（或报告无解）。已知总人数不小于10，不超过100。输入到文件结束为止。

样例输入：

```
2 1 6
2 1 3
```

样例输出：

```
Case 1: 41
Case 2: No answer
```

习题2-3 倒三角形（triangle）

输入正整数$n\leqslant 20$，输出一个n层的倒三角形。例如，$n=5$时输出如下：

```
#########
 #######
  #####
   ###
    #
```

习题 2-4　子序列的和（subsequence）

输入两个正整数 $n<m<10^6$，输出 $\frac{1}{n^2}+\frac{1}{(n+1)^2}+\cdots+\frac{1}{m^2}$，保留 5 位小数。输入包含多组数据，结束标记为 $n=m=0$。提示：本题有陷阱。

样例输入：

```
2 4
65536 655360
0 0
```

样例输出：

```
Case 1: 0.42361
Case 2: 0.00001
```

习题 2-5　分数化小数（decimal）

输入正整数 *a*, *b*, *c*，输出 *a*/*b* 的小数形式，精确到小数点后 *c* 位。$a,b\leqslant 10^6$，$c\leqslant 100$。输入包含多组数据，结束标记为 $a=b=c=0$。

样例输入：

```
1 6 4
0 0 0
```

样例输出：

```
Case 1: 0.1667
```

习题 2-6　排列（permutation）

用 1,2,3,…,9 组成 3 个三位数 *abc*，*def* 和 *ghi*，每个数字恰好使用一次，要求 *abc*:*def*:*ghi*=1:2:3。按照“abc def ghi”的格式输出所有解，每行一个解。提示：不必太动脑筋。

下面是一些思考题。

题目 1。假设需要输出 2, 4, 6, 8,…, 2*n*，每个一行，能不能通过对程序 2-1 进行小小的改动来实现呢？为了方便，现把程序复制如下：

```
1  #include<stdio.h>
2  int main()
3  {
4    int n;
5    scanf("%d", &n);
6    for(int i = 1; i <= n; i++)
7      printf("%d\n", i);
8    return 0;
9  }
```

任务 1：修改第 7 行，不修改第 6 行。

任务 2：修改第 6 行，不修改第 7 行。

题目2。下面的程序运行结果是什么？“!=”运算符表示“不相等”。提示：请上机实验，不要凭主观感觉回答。

```
#include<stdio.h>
int main()
{
  double i;
  for(i = 0; i != 10; i += 0.1)
    printf("%.1f\n", i);
  return 0;
}
```

2.5.2 小结

循环的出现让程序逻辑复杂了许多。在很多情况下，仔细研究程序的执行流程能够很好地帮助理解算法，特别是“当前行”和变量的改变。有些变量是特别值得关注的，如计数器、累加器，以及“当前最小/最大值”这样的中间变量。很多时候，用 printf 输出一些关键的中间变量能有效地帮助读者了解程序执行过程、发现错误，就像本章中多次使用的一样。

别人的算法理解得再好，遇到问题时还是需要自己分析和设计。本章介绍了“伪代码”这一工具，并建议“不拘一格”地使用。伪代码是为了让思路更清晰，突出主要矛盾，而不是写“八股文”。

在程序慢慢复杂起来时，测试就显得相当重要了。本章后面的几个例题几乎个个都有陷阱：运算结果溢出、运算时间过长等。程序的运行时间并不是无法估计的，有时能用实验的方法猜测时间和规模之间的近似关系（其理论基础将在后面介绍），而海量数据的输入输出问题也可以通过文件得到缓解。尽管不同竞赛在读写方式上的规定不同，熟练掌握了重定向、fopen 和条件编译后，各种情况都能轻松应付。

再次强调：编程不是看书看会的，也不是听课听会的，而是练会的。本章后面的上机编程习题中包含了很多正文中没有提到的内容，对能力的提高很有好处。如有可能，请在上机实践时运用输出中间结果、设计伪代码、计时测试等方法。