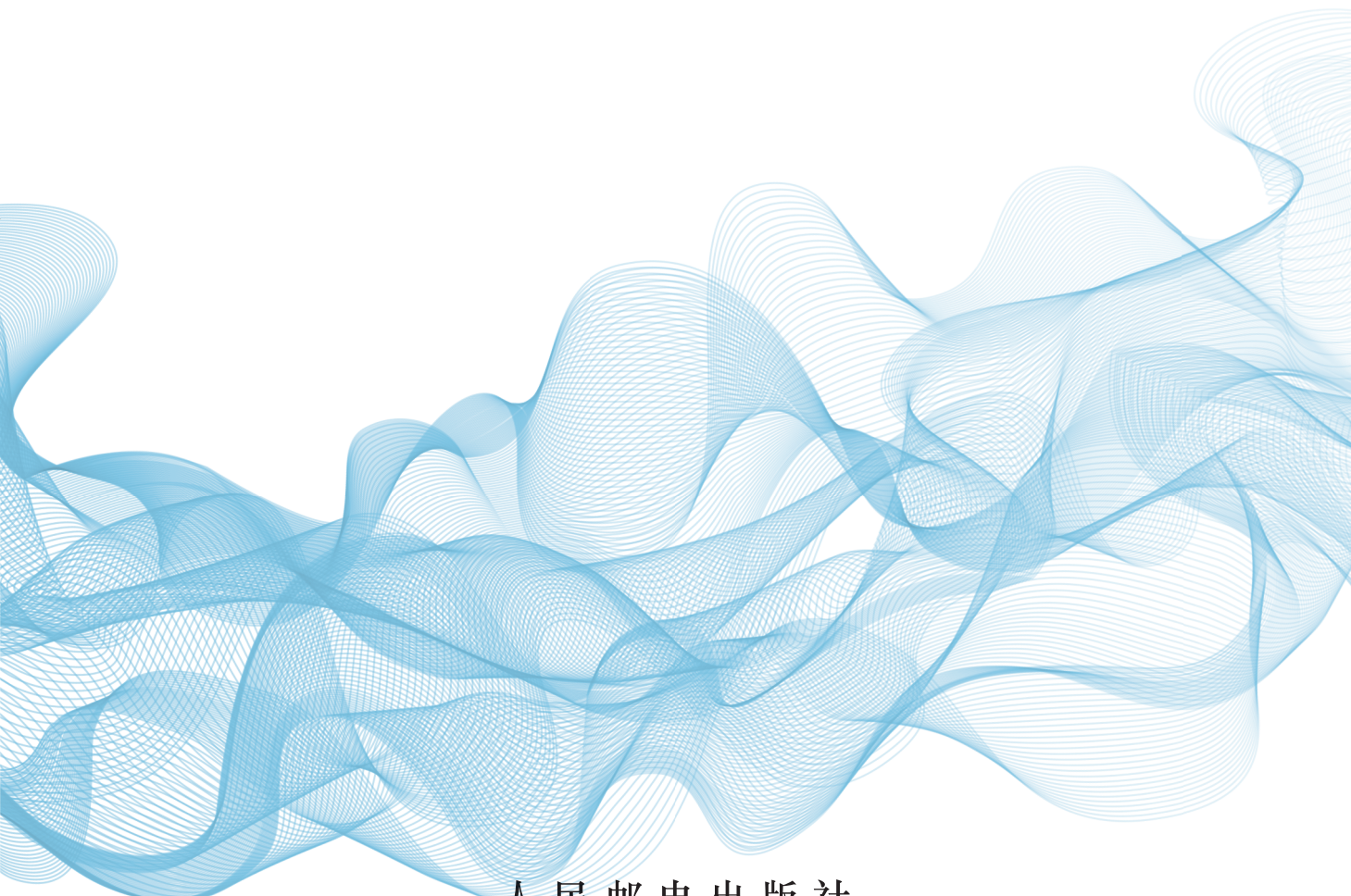
明解C语言

中级篇

[日] 柴田望洋 / 著　丁灵 / 译

TURING
图灵程序设计丛书



人民邮电出版社
北　京

图书在版编目（CIP）数据

明解C语言. 中级篇 /（日）柴田望洋著；丁灵译
. -- 北京：人民邮电出版社，2017.9
（图灵程序设计丛书）
ISBN 978-7-115-46406-4

Ⅰ. ①明… Ⅱ. ①柴… ②丁… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字（2017）第178307号

内 容 提 要

本书延续了《明解C语言：入门篇》图文并茂、示例丰富、讲解细致的风格，在结构上又独树一帜，每章都会带领读者编写一个游戏程序并逐步完善或加以变更，来讲解相关的C语言进阶知识。每章的程序都很简单有趣，而且包含着很多实用性的技巧，例如随机数的生成、数组的应用方法、字符串和指针、命令行参数、文件处理、接收可变参数的函数的生成方法、存储空间的动态分配与释放，等等。此外，还会讲解详细的语法规则、众多库函数的使用方法、算法等知识。

本书适合有一定C语言基础，想要掌握实际编程能力的读者阅读。

◆ 著　　[日] 柴田望洋
译　　丁　灵
责任编辑　杜晓静
执行编辑　刘香娣
责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷
◆ 开本：800×1000　1/16
印张：22
字数：520千字　　2017年9月第1版
印数：1－4 000册　　2017年9月北京第1次印刷
著作权合同登记号　图字：01-2016-3273号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字20170147号

前　言

大家好。

这本《明解 C 语言：中级篇》是为那些已经学完入门内容，想要掌握实际编程能力的读者编写的。

为了让大家能够从 C 语言编程的“新手”中毕业，踏实地在“中级者”的道路上前进，本书将带领大家一边接触众多的程序一边学习，这些程序的编写和运行都很有趣。

本书中选取的程序包括以下题材。

- 猜数游戏
- 兼具扩大视野的心算训练
- 字符的消除和移动（字幕显示等）
- 猜拳游戏
- 珠玑妙算
- 记忆力训练
- 日历显示
- 打字练习
- 英语单词学习软件……

上述程序都很简练，大家尝试之后可能会惊讶道：“这么短的程序居然这么有意思！”

当然这些程序不仅仅是有趣而已，每个程序中都包含着实用性的技巧，例如随机数的生成、数组的应用方法、包含汉字的字符串、字符串和指针、命令行参数、文件处理、生成接收可变参数的函数的方法、存储空间的动态分配与释放，等等。另外我们还将学习详细的语法规则、众多库的规范以及使用方法等。

希望大家通过阅读本书，争取从新手阶段完全毕业！

柴田望洋

2015 年 4 月

“明解”及“新明解”是三省堂股份有限公司的注册商标。
正文中的商品名称通常是各公司的商标或注册商标。
正文中没有明确写出 TM 和 ® 标识。

©2015　本书包含程序在内的所有内容均受到著作权法的保护。
未经作者及出版社允许，禁止擅自复制及盗印本书内容。

导　读

笔者迄今为止遇到过很多难以从 C 语言“新手”阶段毕业的人，他们似乎都抱有下面这样的烦恼。

——虽然能理解入门书中所写的程序，但换成自己写就写不出来了。

——虽然了解数组和指针等语法知识，但不知该如何在实际程序中使用。

——在新员工培训中学到的基础知识和实际工作中要求的相差甚远，或者在大学课堂上所学的内容跟毕业设计要求编写的程序难度大相径庭，因此不知如何是好。

事实上，这些烦恼在某种意义上也是无可奈何的。因为在学习编程语言的初级阶段，学习“语言”本身的基础知识是必需的，无暇顾及应用语言的“编程”。

当然，语言和编程两者也不是完全对立的。但是对新手而言，如果想要同时学习这两者，要记住和掌握的东西未免太多了。因此，初学阶段往往把重点放在“语言”上，很多入门书的结构都是如此。

本书的结构和一般图书不同，每章的标题不是“数组”“指针”这样的编程术语，而是像下面这样。

第 1 章　猜数游戏
第 2 章　专注于显示
第 3 章　猜拳游戏
第 4 章　珠玑妙算
第 5 章　记忆力训练
第 6 章　日历
第 7 章　右脑训练
第 8 章　打字练习
第 9 章　文件处理
第 10 章　英语单词学习软件

我们在每一章都会“开发程序”。在开发程序的过程中，逐渐学习相关的语法、库函数、算法以及编程知识。

我们要学习的程序清单总共有 111 个。

▶为了帮助大家理解，本书使用了大量简明易懂的图表（全书共有 152 张图表）。

下面总结了一些阅读本书时需要事先了解和注意的事项。

▪ 关于阅读本书所需的预备知识和本书的难易程度

本书是“明解 C 语言”系列的第二本书，在讲解《中级篇》的同时，也会带领大家一并复习《入门篇》中学过的内容。

►因此，学习内容和难易程度跟《入门篇》和同系列的第三本《实践篇》有部分重复。这主要考虑到有些读者在入门学习时采用的是非本系列《入门篇》的其他图书。

▪ 关于标准库函数的解说

大家将在本书中学到 ***random*** 函数、***srand*** 函数、***fopen*** 函数等众多 C 语言标准库函数（包括函数式宏共有 57 个）。这些函数的解说都是笔者基于 C 标准库的 JIS 标准文件改写而成的，为了传达严格的规范，表述可能会略显生硬。

▪ 关于源程序

大家可以从以下网站下载本书涉及的源程序。若是这些程序能为大家所用，笔者将感到万分荣幸。

http://www.ituring.com.cn/book/1810

目　录

第 1 章

猜数游戏

本章中要编写的是“猜数游戏”程序。我们先来做一个测试版本，这个测试版的程序只能比较玩家输入的数值和计算机准备的数值，之后再逐渐为其追加其他功能。

本章主要学习的内容

- if 语句的结构 / 效率 / 可读性
- do 语句（先循环后判断）
- while 语句（先判断后循环）
- for 语句（先判断后循环）
- break 语句
- 相等运算符和关系运算符
- 逻辑运算符
- 增量运算符（前置 / 后置）
- sizeof 运算符
- 表达式求值
- 德・摩根定律
- 随机数的生成与种子的变更
- 对象宏
- 数组
- 数组的遍历
- 数组元素的初始化
- 数组元素个数的设定和获取
- ⊙ rand 函数
- ⊙ srand 函数
- ⊙ RAND_MAX

1–1 猜数判定

本章中会编写一个“猜数游戏”的程序。首先我们要做的是一个测试版本，用来显示玩家从键盘输入的值和计算机事先准备好的“目标数字”的比较结果。

通过 if 语句实现条件分支

List 1-1 所示的程序是测试版的“猜数游戏”。

先运行程序。因为程序提示输入 0～9 的数值，所以我们就在键盘上键入数值，这样一来，程序就会把键入的数值和“目标数字”进行比较，并显示出比较后的结果。

List 1-1 chap01/kazuate1.c

```
/* 猜数游戏（其一：测试版） */
#include <stdio.h>

int main(void)
{
    int no;          /* 读取的值 */
    int ans = 7;     /* 目标数字 */

    printf("请猜一个0～9的整数。\n\n");

    printf("是多少呢：");
    scanf("%d", &no);

    if (no > ans)
        printf("\a再小一点。\n");
    else if (no < ans)
        printf("\a再大一点。\n");
    else
        printf("回答正确。\n");

    return 0;
}
```

if 语句

运行示例 1

```
请猜一个0～9的整数。

是多少呢：9↵
♪再小一点。
```

运行示例 2

```
请猜一个0～9的整数。

是多少呢：5↵
♪再大一点。
```

运行示例 3

```
请猜一个0～9的整数。

是多少呢：7↵
♪回答正确。
```

本游戏中的“目标数字”是 7，用变量 *ans* 表示，从键盘输入的值则用变量 *no* 表示。

程序通过阴影部分的 **if** 语句来判断 *no* 和 *ans* 两个变量值的大小关系，然后如 Fig.1-1 所示，根据判断结果显示“再小一点。”“再大一点。”“回答正确。”。

输出的字符串中包含两种**转义字符**。一个是我们很熟悉的 **\n**，表示**换行**；另一个是 **\a**，表示**警报**。在大多数环境下，一旦输出警报就会响起蜂鸣音，因此本书在运行示例中采用♪符号表示警报。

►关于转义字符，我们会在第 2 章中详细介绍。

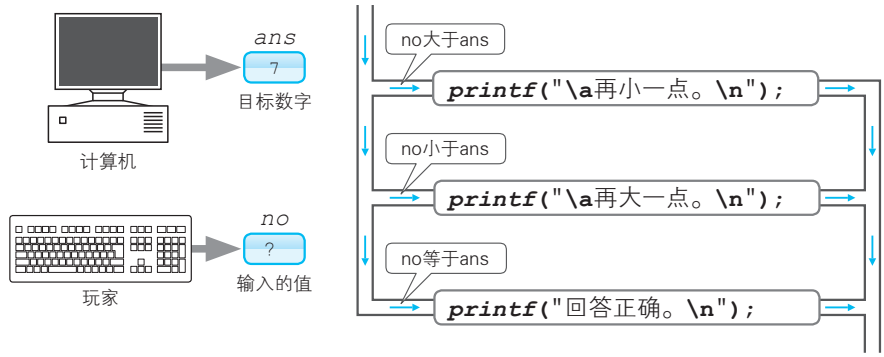


Fig.1-1 通过 if 语句实现程序流程分支

if 语句的嵌套

下面让我们来了解一下比较 *no* 和 *ans* 这两个变量值的 **if** 语句的结构。

if 语句是通过对名为**控制表达式**的表达式进行**求值**（**专栏 1-1**），再根据求值结果把程序流程分为不同的分支的语句，它包含两种语句结构，如右图所示。

if 语句的结构

- **if**（表达式）语句
- **if**（表达式）**else** 语句

控制表达式

▶（）中的**表达式**为控制表达式。

本程序中的 **if** 语句采用以下形式。

if（表达式） 语句 **else if**（表达式） 语句 **else** 语句

当然，这里**并不是**为了把程序流程分成三个分支才特意采用这种语句结构的。从字面意思可知，**if** 语句是一种语句，因此 **else** 控制的语句也可以是 **if** 语句。如 Fig.1-2 所示，程序采用了在 **if** 语句中**嵌套 if** 语句的结构。

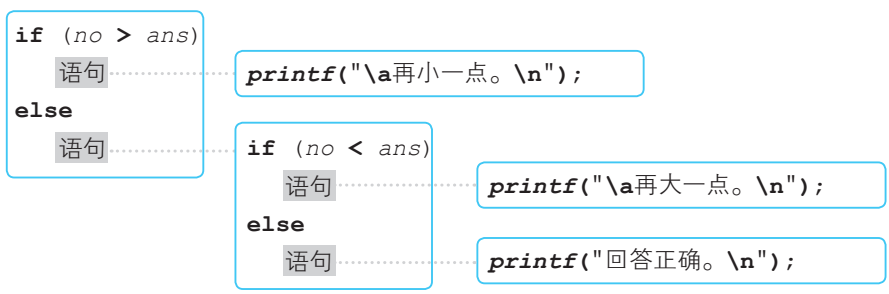


Fig.1-2 if 语句的嵌套

实现多分支的方法

为了跟本程序的 **if** 语句 1 达到同样效果，笔者编写了 2 和 3 中的 **if** 语句。

下面让我们一起来比较并讨论一下这三个程序，以加深对 **if** 语句的理解。

程序 2

最末尾的 **else** 语句后面追加了阴影部分的内容。只有当两个判断（*no* **>** *ans*）和（*no* **<** *ans*）都不成立，也就是 *no* 和 *ans* 相等时，程序才会运行这部分。

在阴影部分进行的判断是**肯定会成立的条件**。

程序 3

这里有三个 **if** 语句并列。无论变量 *no* 和 *ans* 之间的大小关系如何，程序都会进行这三个条件判断。

笔者根据变量 *no* 和 *ans* 的大小关系，对这三个程序中都进行了什么判断（对哪个控制表达式进行了求值）进行了总结，如 Table 1-1 所示。

1 List 1-1的if语句

```
if (no > ans)
    printf("\a再小一点。\n");
else if (no < ans)
    printf("\a再大一点。\n");
else
    printf("回答正确。\n");
```

2 在最后的else语句中追加if (no == ans)

```
if (no > ans)
    printf("\a再小一点。\n");
else if (no < ans)
    printf("\a再大一点。\n");
else if (no == ans)
    printf("回答正确。\n");
```

3 三个独立的if语句并列

```
if (no > ans)
    printf("\a再小一点。\n");
if (no < ans)
    printf("\a再大一点。\n");
if (no == ans)
    printf("回答正确。\n");
```

Table 1-1 三个程序所进行的判断

大小关系	*no* **>** *ans*时	*no* **<** *ans*时	*no* **==** *ans*时
1	①	① ②	① ②
2	①	① ②	① ② ③
3	① ② ③	① ② ③	① ② ③

① 判断（*no* **>** *ans*）
② 判断（*no* **<** *ans*）
③ 判断（*no* **==** *ans*）

比如，我们假设 *no* 大于 *ans*，则程序 1 和程序 2 都只会进行①的判断，即（*no* **>** *ans*）。

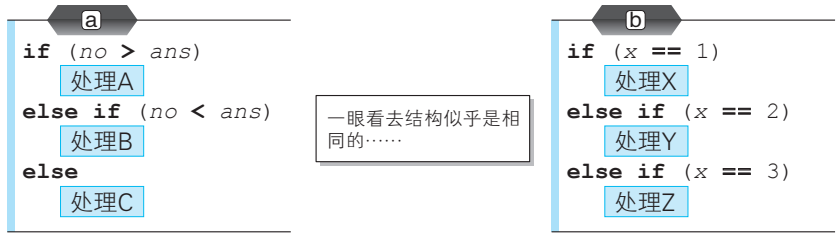
▶因为如果 *no* 大于 *ans*，那么在执行完 ***printf*("\a** 再小一点。**\n")** 后，整个 **if** 语句就执行完毕了。

而在程序 3 这种有三个独立的 **if** 语句并列的情况下，则会执行三次判断，即①的（*no* **>** *ans*）、②的（*no* **<** *ans*），还有③的（*no* **==** *ans*）。这种实现方法效率最低。

*

不管在何种条件下，判断次数最少的都是程序 1 的 **if** 语句。

程序 1 的 **if** 语句的优点不光只有判断次数少而已。为了让大家理解这一点，这里通过 Fig.1-3 来说明。



● Fig.1-3 看似相同但大相径庭的 if 语句分支

图a的 if 语句

图a的 **if** 语句跟程序1中的 **if** 语句结构相同，程序流程都分为三个分支。执行的不是“处理 A”就是“处理 B”，再不然就是“处理 C”。

▶不会出现没有执行任何处理或者执行了两项和三项处理的情况。

图b的 if 语句

图b的 **if** 语句是根据变量 *x* 的值进行分支的。

看上去程序似乎执行了“处理 X”“处理 Y”“处理 Z”三者中的一项处理，然而变量 *x* 的值如果不是 1、2、3，**那么程序就不会进行任何处理**。

如 Fig.1-4 所示，程序流程实质上分为四个分支。

如此，图b与图a的 **if** 语句的结构**完全不同**，因此**不能**省略最后的判断 **if** (*x* == 3)。

```
if (x == 1)
    处理X
else if (x == 2)
    处理Y
else if (x == 3)
    处理Z
else
    什么也不做
```

藏有看不见的分支

● Fig.1-4 图b的说明

▶如果省略了，那么即使 *x* 的值不是 3，而是 4 或 5 等，“处理 Z”也会被运行。

*

程序1的 **if** 语句的结构如图a，在末尾的 **else** 语句后面是没有 **if** 语句的，因此一看就明白**不存在更多分支**。

就程序的易读性而言，程序1也要优于程序2，因为程序2在末尾的 **else** 语句后放了个“多余”的判断。

▶如果一定要对程序的读者强调“当 *no* 等于 *ans* 时这么做”，则可以按照程序2那样来实现。

通常，编译器的优化技术会内部删除程序2的这种“多余”的判断，因此我们没必要太在意效率问题。

专栏 1-1　表达式和求值

▪ 表达式是什么

编程的世界里经常会使用**表达式**（expression）这个术语，表达式包括以下内容。

- 变量
- 常量
- 把变量和常量用运算符结合起来的式子

我们来看下面这个式子。

```
n + 52
```

变量 *n*、整数常量 52，以及用运算符号 **+** 将这两者连接起来的 `n + 52` 都是表达式。

再看下面这个式子。

```
x = n + 52
```

在这里，*x*、*n*、52、`n + 52`、`x = n + 52` 都是表达式。

一般情况下，用 × × 运算符连接的表达式就叫作 × × 表达式。例如用赋值运算符把 *x* 和 `n + 52` 连接的表达式 `x = n + 52` 就是**赋值表达式**（assignment expression）。

▪ 表达式的求值

原则上，所有的表达式都包含值（**void** 型这种特殊的类型除外）。在运行程序时可以计算这个值。

计算表达式的值就叫作**求值**（evaluation）。**程序通过逐一对各个表达式进行求值而得以运行。**

Fig.1C-1 是求值的具体示例（假设这张图中 **int** 型的变量 *n* 的值为 135）。

因为变量 *n* 的值为 135，所以 *n*、52、`n + 52` 各自求值后得到的值分别是 135、52、187。当然这三个值的类型都是 **int** 型。

本书采用电子体温计的图示来表示求值结果。左侧的小字是“类型”，右侧的大字是“值”。

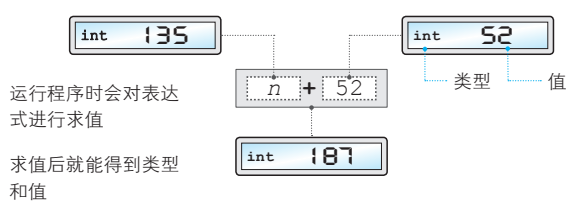
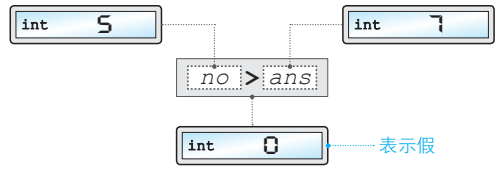


Fig.1C-1　表达式的求值（int 型 + int 型）

List 1-1 的 **if** 语句最初的控制表达式是 `no > ans`。如果变量 *no* 读取的值是 5，那么求值过程就会如 Fig.1C-2 所示的那样。

关系运算符用于判断两个操作数的值（求值结果）的大小关系（1-2 节）。此时因为判断条件不成立，所以对表达式 `no > ans` 求值得到的是表示假的“**int** 型的 0”。

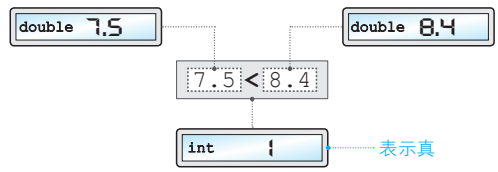


● Fig.1C-2 表达式的求值（int 型 > int 型）

如果 *no* 的值大于 7，那么就会得到表示真的“**int** 型的 1”。

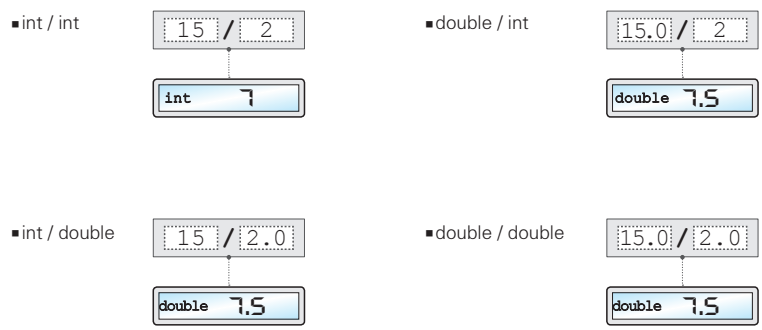
*

在这个示例中，运算对象（左边和右边的操作数）的类型是 **int** 型，求值后得到的类型也是 **int** 型。对关系运算符而言，就算操作数的类型不是 **int** 型，它也会生成 **int** 型。如 Fig.1C-3 所示，对比较 **double** 型的 7.5 和 8.4 的表达式 7.5 **<** 8.4 求值，得到的结果是 **int** 型的 1。



● Fig.1C-3 表达式的求值（double 型 < double 型）

作为运算对象的操作数的类型不一定是相同的。Fig.1C-4 中把 **int** 型的 15 和 **double** 型的 15.0 分别除以了 **int** 型的 2 和 **double** 型的 2.0（这张图中省略了对常量 15 和 15.0 的求值）。可见如果有一方的操作数是 **double** 型，那么运算结果就会是 **double** 型。



● Fig.1C-4 表达式的求值（对 int 型和 double 型做除法运算）

1–2 重复到猜对为止

如果“猜数游戏”只允许玩家输入一次数值，那未免太无趣了。我们把程序改良一下，让玩家可以一直重复输入直到猜对为止。

通过 do 语句循环

如果玩家只能输入一次数值，那么想要猜对数值的话，就需要不停地重启程序直到猜对为止，这样一来不只是没意思，还麻烦得要命。

下面我们把程序改良一下，让玩家能够反复输入数值直到猜对为止，改良后的程序如 List 1-2 所示。

List 1-2 chap01/kazuate2.c

```
/* 猜数游戏（其二：重复到猜对为止——利用do语句）*/
#include <stdio.h>

int main(void)
{
    int no;          /* 读取的值 */
    int ans = 7;     /* 目标数字 */

    printf("请猜一个0～9的整数。\n\n");

    do {
        printf("是多少呢：");
        scanf("%d", &no);

        if (no > ans)
            printf("\a再小一点。\n");
        else if (no < ans)
            printf("\a再大一点。\n");
    } while (no != ans);

    printf("回答正确。\n");

    return 0;
}
```

do语句

/* 重复到猜对为止 */

运行示例

```
请猜一个0～9的整数。

是多少呢：6
♪再大一点。
是多少呢：8
♪再小一点。
是多少呢：7
回答正确。
```

List 1-2 中删除了 List 1-1 中 **if** 语句的后半截，并在此基础上追加了阴影部分的 **do** 语句。

do 语句是通过**先循环后判断**（后述）重复进行处理的语句，其结构如右图所示。

▶和之前学习的 **if** 语句，以及接下来要学习的 **while** 语句和 **for** 语句等语句的结构不同，**do** 语句的末尾带有分号“;”。

do 和 **while** 围起来的语句叫作**循环体**。只要 () 中的表达式，也就是**控制表达式**的求值结

果不为 0，那么循环体就会被一直重复运行下去，直到控制表达式的求值结果为 0，才会结束重复运行。

下面参照 Fig.1-5 来理解如何通过本程序的 **do** 语句实现循环。

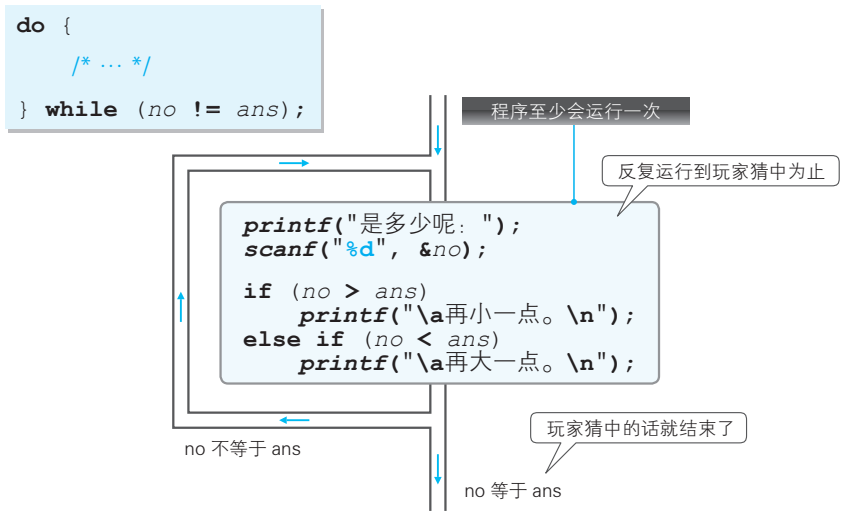


Fig.1-5　通过 do 语句来重复程序流程

do 语句的控制表达式为 *no* **!=** *ans*。

运算符 **!=** 对左边和右边的操作数的值是否**不相等**这一条件进行判断。如果这个条件成立，程序就会生成 **int** 型的 1，不成立则会生成 **int** 型的 0。

如果读取的值 *no* 和目标数字 *ans* 不相等，那么对控制表达式 *no* **!=** *ans* 进行求值，得到的值就是 1。因此需要通过 **do** 语句来重复运行程序，再次运行用 {} 括起来的代码块，也就是循环体。

当程序读取到的 *no* 和目标数字 *ans* 是同一个值时，控制表达式的求值结果就是 0，循环就结束了，此时画面显示“回答正确。”，程序运行结束。

相等运算符和关系运算符

相等运算符（equality operator）和**关系运算符**（relational operator）的判断条件成立的话就会生成 **int** 型的 1，不成立就会生成 **int** 型的 0。

▶ **int** 型的 1 表示“真”，0 表示“假”（**专栏 1-2**）。

相等运算符“==”“!=”

判断两个操作数是否相等。

关系运算符 “<” “>” “<=” “>=”

判断两个操作数的大小关系。

通过 while 语句循环

除了 **do** 语句外，C 语言的循环语句[①]还有 **while** 语句和 **for** 语句。

我们用**先判断后循环**的 **while** 语句试着写出之前的程序，如 List 1-3 所示。

List 1-3 chap01/kazuate2while.c

```
/* 猜数游戏[其二（另一种解法）：重复到猜对为止——利用while语句]*/

#include <stdio.h>

int main(void)
{
    int no;          /* 读取的值 */
    int ans = 7;     /* 目标数字 */

    printf("请猜一个0~9的整数。\n\n");

    while (1) {
        printf("是多少呢：");
        scanf("%d", &no);

        if (no > ans)
            printf("\a再小一点。\n");
        else if (no < ans)
            printf("\a再大一点。\n");
        else
            break;
    }

    printf("回答正确。\n");

    return 0;
}
```

while语句

break语句

运行示例

```
请猜一个0~9的整数。

是多少呢：6
♪再大一点。
是多少呢：8
♪再小一点。
是多少呢：7
回答正确。
```

while 语句的结构如右图所示。

while 语句的结构

```
while（表达式）语句
```

循环体

控制表达式

只要**控制表达式**的求值结果不为 0，那么作为循环体的**语句**就会永远重复运行下去。但是求值结果一旦为 0，就不再循环了。

因为本程序的 **while** 语句的控制表达式是 1，所以**循环会永远进行下去**。这样的循环一般称为“无限循环”。

break 语句

一直重复的话，程序会永无止境。为了强制跳出循环语句，我们在本程序中使用了 **break** 语句。

因为 **break** 语句是在 *no* 和 *ans* 相等时运行的，所以通过 **while** 语句进行的循环会被强制中断。

① 声明一组要反复执行的命令，直到满足某些条件为止。——译者注

▶使用 **break** 语句的程序往往不容易读也不容易理解，我们只在“某个特殊的条件成立时，因为某种原因想强制结束循环语句”的情况下使用 **break** 语句就好。这里所举的“猜数游戏”的循环结构很简单，因此实现这个程序不需要用到 **break** 语句，像 List 1-2 那样通过 **do** 语句（不使用 **break** 语句）就能实现。

while 语句和 do 语句

很难看出程序中的 **while** 是属于 **do** 语句的一部分还是属于 **while** 语句的一部分，下面我们结合右图中所示的程序来思考一下。

首先把 0赋给变量 *x*，然后通过 **do** 语句对变量 *x* 的值进行增量操作，直到 *x* 等于 5 为止。

接下来，在 **while** 语句中对 *x* 的值进行减量操作，并显示其结果。

▶关于增量（increment）运算符 **++** 和减量（decrement）运算符 **--**，在 1–4 节中会详细为大家讲解。

如右图所示，我们把 { } 括起来的代码块当作 **do** 语句的循环体。

do 语句的 while

while 语句的 while

```
x = 0;
do
    x++;
while (x <= 5);
while (x >= 0)
    printf("%d ", --x);
```

```
x = 0;
do {
    x++;
} while (x <= 5);
while (x >= 0)
    printf("%d ", --x);
```

这样一来，**只要看每一行的开头就能区分 while 属于哪一部分了**。

> **while:** 如果开头是 **while**，则属于 **while** 语句的开头部分。
> } **while:** 如果开头是 }，则属于 **do** 语句的结尾部分。

不管是 **do** 语句还是 **while** 语句抑或是 **for** 语句，只要其循环体是单一的语句，那么就没必要特意导入代码块。

话虽如此，对 **do** 语句来说，其循环体如果是单一的语句，导入代码块则会增加程序的易读性。

先判断后循环和先循环后判断

根据何时判断是否继续处理，循环可以分为两种。

先判断后循环（while 语句和 for 语句）

在进行处理前，先判断是否要继续处理。会出现循环体一次也没有运行的情况。

先循环后判断（do 语句）

在进行处理后，再判断是否要继续处理。循环体至少会被运行一次。

1-3 随机设定目标数字

在前面的“猜数游戏”中，“目标数字”都是事先在程序里设置好的，所以我们事先是知道答案的。为了提升游戏的趣味性，我们来让这个值自动变化。

rand 函数：生成随机数

为了每次游戏时都能改变“目标数字”，我们需要一个随机数。用于生成随机数的就是 ***rand*** 函数，如下所示。

	rand
头文件	**#include** <stdlib.h>
格式	**int** ***rand***(**void**);
功能	计算 0~**RAND_MAX** 的伪随机整数序列。此外，其他库函数在运行时会无视本函数的调用
返回值	返回生成的伪随机数整数

这个函数生成的随机数是 **int** 型的整数。在所有编程环境中其最小值都为 0，但最大值则取决于编程环境，所以我们用 <stdlib.h> 头文件将其定义成一个名为 **RAND_MAX** 的**对象宏**（object-like macro），其定义的示例如下所示。

RAND_MAX

```
#define RAND_MAX  32767        /* 定义的示例：值根据编程环境而有所差别*/
```

RAND_MAX 的值根据规定不得低于 32767，因此 ***rand*** 函数的运行过程如 Fig.1-6 所示。

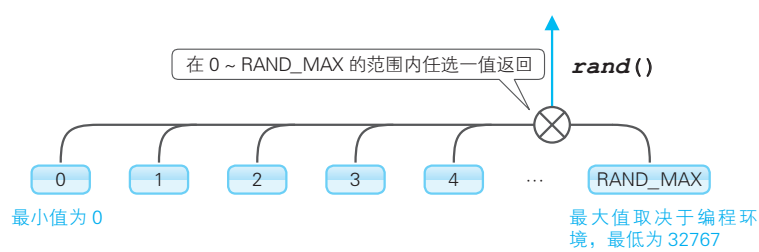


Fig.1-6 通过 rand 函数生成随机数

下面我们来尝试实际生成并显示随机数。请运行 List 1-4 所示的程序。

List 1-4 chap01/random1.c

```
/* 生成随机数（其一）*/

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int retry;              /* 再运行一次？ */

    printf("在这个编程环境中能够生成0～%d的随机数。\n", RAND_MAX);

    do {
        printf("\n生成了随机数%d。\n", rand());

        printf("再运行一次？…（0）否（1）是：");
        scanf("%d", &retry);
    } while (retry == 1);

    return 0;
}
```

rand 函数生成的最大随机数

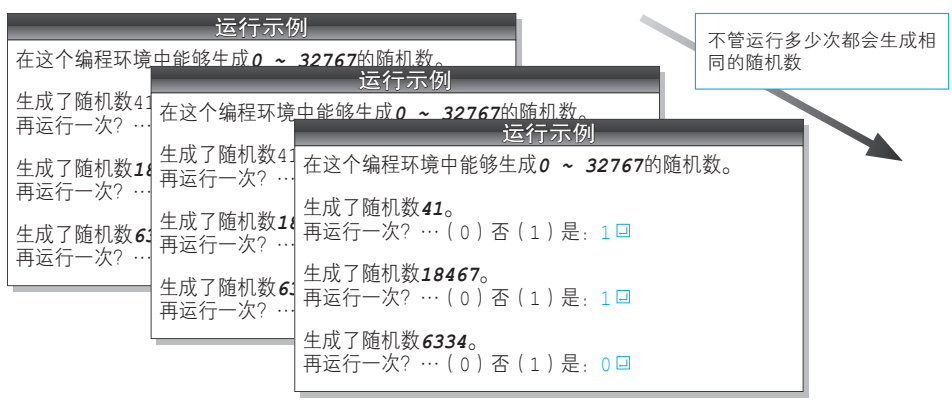
生成 0~RAND_MAX 的随机数并返回

首先显示的是能够生成的随机数的“范围”。最小值是 0，最大值是 **RAND_MAX** 的值（值取决于编程环境）。

然后显示的是 ***rand()*** 返回的随机数值，当然这个值在 0 ～ **RAND_MAX** 的范围内。

对于“再运行一次?”的问题，如果选择了“是”，那么就能重复生成并显示随机数。

请多运行几次程序，结果如 Fig.1-7 所示，**总会生成一个相同的随机数序列**。这很令人费解，***rand*** 函数生成的值**真的是随机的吗?**



※ 这只是一个例子，实际生成什么值将取决于编程环境。

Fig.1-7 List 1-4 的运行示例

srand 函数：设置用于生成随机数的种子

rand 函数是对一个叫作“种子”的基准值加以运算来生成随机数的。之所以先前每次运行

程序都会生成同一个随机数序列，是因为 ***rand*** **函数的默认种子是常量 1**。要生成不同的随机数序列，就必须改变种子的值。

负责执行这项任务的就是 ***srand*** 函数，如下所示。

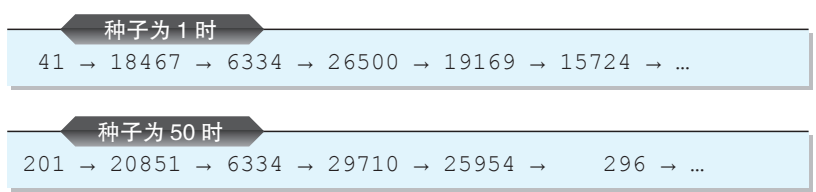
	srand
头文件	**#include** <stdlib.h>
格式	**void** ***srand***(**unsigned** *seed*);
功能	给后续调用的 ***rand*** 函数设置一个种子（*seed*），用于生成新的伪随机数序列。如果用同一个种子的值调用本函数，就会生成相同的伪随机数序列。如果在调用本函数之前调用了 ***rand*** 函数，就相当于程序在一开始调用了本函数，把 *seed* 设定成了 1，最后会生成一个种子值为 1 的序列。此外，其他库函数在运行时会无视本函数的调用
返回值	无

比如，假设我们调用了 ***srand*****(50)**。这样一来，之后调用的 ***rand*** 函数就会利用设定的新种子值 50来生成随机数。

Fig.1-8 所示为在某个编程环境中生成的随机数序列的示例。

当种子值为 1 时，在最初调用 ***rand*** 函数时生成的是 41，再调用时生成 18467，接下来是 6334……

如果种子值是 50，则会依次生成 201、20851、6334……



※ 这只是一个例子，实际生成什么值将取决于编程环境。

Fig.1-8 种子和 rand 函数生成的随机数序列的示例

如上图所示，一旦决定了种子的值，之后生成的随机数序列也就确定了。因此如果想要每次运行程序时都能生成不同的随机数序列，**就必须把种子值本身从常量变成随机数**。

然而，为了生成随机数而需要随机数，这本身很矛盾。

▶ ***rand*** 函数生成的是叫作伪随机数的随机数。伪随机数看起来像随机数，却是基于某种规律生成的。因为能预测接下来会生成什么数值，所以才叫作伪随机数。真正的随机数是无法预测接下来会生成什么数值的。

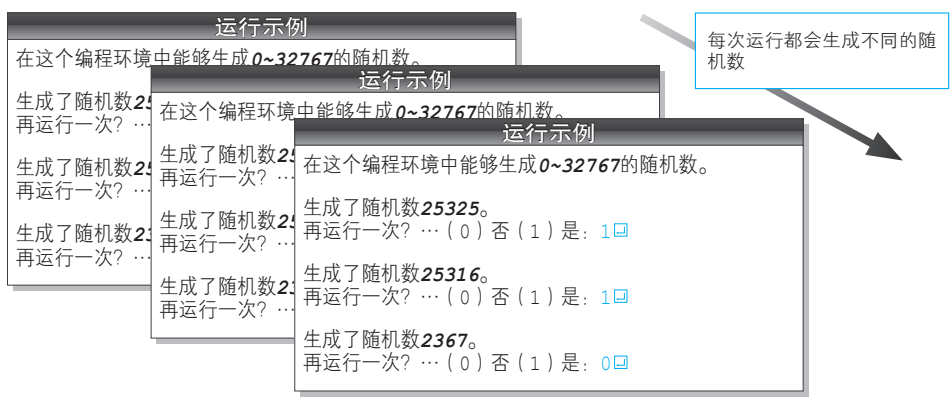
我们一般使用的方法是**把运行程序时的时间当作种子**。List 1-5 的程序中就使用了这个方法。

List 1-5 chap01/random2.c

```
/* 生成随机数（其二：根据当前时间设定随机数的种子）*/

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

int main(void)
{
    int retry;                  /* 再运行一次？ */

    srand(time(NULL));          /* 根据当前时间设定随机数的种子 */

    printf("在这个编程环境中能够生成0～%d的随机数。\n", RAND_MAX);

    do {
        printf("\n生成了随机数%d。\n", rand());

        printf("再运行一次？…（0）否（1）是：");
        scanf("%d", &retry);
    } while (retry == 1);

    return 0;
}
```

请运行一下程序。如 Fig.1-9 所示，每次启动都会生成不同的随机数序列。

►关于获取当前时间所使用的 *time* 函数，我们会在第 6 章详细学习，在此之前，只需把程序中的阴影部分当成是固定的一部分即可（**#include** <time.h> 也是必不可少的）。



※这只是一个例子，实际生成什么值将取决于编程环境。

Fig.1-9 List 1-5 的运行示例

随机设定目标数字

rand 函数生成的值范围是 0 ～ **RAND_MAX**，话虽如此，但我们需要的随机数不会每次都恰好在这个范围内。

一般情况下，我们需要的是**某个特定范围内的随机数**。如果我们需要“大于等于 0 且小于等于 10”的随机数，可以像下面这样求出。

```
rand() % 11          /* 生成大于等于0且小于等于10的随机数 */
```

这里使用的方法是把非负整数值除以 11，就得到余项（余数）为 0，1，…，10。

▶大家注意不要把非负整数值错除以 10。用 10 除得到的余数是 0，1，…，9，无法生成 10。

*

现在大家已经掌握了如何生成随机数，那么我们就来把猜数游戏中的“目标数字”设定为 0 ~ 999 的随机数吧。对应的程序如 List 1-6 所示。

▶笔者没有以 **while** 语句版本的 List 1-3 为基准，而只在 **do** 语句版本的 List 1-2 的基础上做了一些细微的修改，增加了部分内容。

List 1-6 chap01/kazuate3.c

```
/* 猜数游戏（其三：目标数字是0 ~ 999的随机数）*/

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

int main(void)
{
    int no;      /* 读取的值 */
    int ans;     /* 目标数字 */

    srand(time(NULL));       /* 设定随机数的种子 */
    ans = rand() % 1000;     /* 生成0~999的随机数 */

    printf("请猜一个0~999的整数。\n\n");

    do {
        printf("是多少呢：");
        scanf("%d", &no);

        if (no > ans)
            printf("\a再小一点。\n");
        else if (no < ans)
            printf("\a再大一点。\n");
    } while (no != ans);                    /* 重复到猜对为止 */

    printf("回答正确。\n");

    return 0;
}
```

运行示例

```
请猜一个0~999的整数。

是多少呢：499↵
再大一点。
是多少呢：749↵
再小一点。
是多少呢：624↵
回答正确。
```

阴影部分把生成的随机数除以 1000 后得到的余数赋给了变量 *ans*。

仅仅是把目标数字变成了随机数，就大大地提升了猜数游戏的趣味性。大家可以多运行几次感受一下。

话说回来，大家知道怎么才能**最快猜中**吗？一开始输入 499，然后根据程序的判定结果（是

大还是小）再输入 749 或者 249，**每次都把范围缩小到一半**。

*

目标数字的范围很容易变更。下面举两个具体的例子。

把目标数字定为 1 ~ 999

将程序的阴影部分改写成下面这样。

```
ans = 1 + rand() % 999;          /* 生成1～999的随机数 */
```

把目标数字定为 3 位数的整数（100 ~ 999）

将程序的阴影部分改写成下面这样。

```
ans = 100 + rand() % 900;        /* 生成100～999的随机数 */
```

*

最后只要把测试版本的 kazuate1.c、kazuate2.c、kazuate3.c 再重复追加和修正 2 到 3 行，猜数游戏就完成了。

小结

生成随机数的准备工作（设定种子）

生成随机数之前需要基于当前时间设定“种子”的值。

```
#include <time.h>
#include <stdlib.h>
/* … */
srand(time(NULL));               /* 设定随机数的种子 */
```

在最初调用 **rand** 函数之前先调用 **srand** 函数（至少调用一次，调用次数不限）。
如果没有做上述准备工作，那么种子的值就会默认为 1，会生成相同的随机数序列。

生成随机数

一旦调用 **rand** 函数，就会得到一个大于等于 0 且小于等于 **RAND_MAX** 的随机数。**RAND_MAX** 的值取决于编程环境，即大于等于 32767。

此外，如果想把随机数定在某个特定范围内，可以像下面这样操作。

```
    rand() % (a + 1)             /* 大于等于0且小于等于a的随机数 */
b + rand() % (a + 1)             /* 大于等于b且小于等于b + a的随机数 */
```

限制输入次数

只要不断输入数值，终会猜对。为了**给玩家以紧张感**，我们把玩家最多可输入的次数限制在 10 次之内。变更后的程序如 List 1-7 所示。

List 1-7 chap01/kazuate4.c

```
/* 猜数游戏（其四：限制输入次数）*/

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

int main(void)
{
    int no;                         /* 读取的值 */
    int ans;                        /* 目标数字 */
    const int max_stage = 10;       /* 最多可以输入的次数 */
    int remain = max_stage;         /* 还可以输入几次？ */

    srand(time(NULL));              /* 设定随机数的种子 */
    ans = rand() % 1000;            /* 生成0～999的随机数 */

    printf("请猜一个0～999的整数。\n\n");

    do {
        printf("还剩%d次机会。是多少呢：", remain);
        scanf("%d", &no);
        remain--;                   /* 把所剩次数进行减量 */

        if (no > ans)
            printf("\a再小一点。\n");
        else if (no < ans)
            printf("\a再大一点。\n");
    } while (no != ans && remain > 0);

    if (no != ans)
        printf("\a很遗憾，正确答案是%d。\n", ans);
    else {
        printf("回答正确。\n");
        printf("您用了%d次猜中了。\n", max_stage - remain);
    }

    return 0;
}
```

变量 *max_stage* **表示玩家最多可输入的次数**，在这里是 10 次。

另一个新的变量 *remain* 表示**还能够输入多少次**。当然，其初始值是 *max_stage*，也就是 10。如 Fig.1-10 所示，玩家每次输入数值时，都会对 *remain* 的值进行减量操作（如 10, 9, 8, …），即在原基础上减去 1。

当这个值为 0 时，游戏就结束了，因此 **do** 语句的判断不仅包含表达式 *no* **!=** *ans*，还要加上阴影部分的 *remain* **>** 0。

连接两个表达式的逻辑与运算符 **&&** 只会在两边的操作数都不为 0 时生成 **int** 型的 1，否则便生成 0。

因此，不仅当玩家猜中时（图 a）循环会结束，当玩家输入 10 次都没猜中，*remain* 变成 0（图 b）时，循环也会结束。

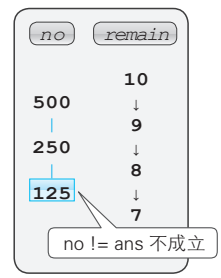
▶关于循环结束的条件和 **&&** 运算符，我们会在**专栏 1-2** 中学到。

此外，用 *max_stage* 减去 *remain* 就可以知道玩家是在第几次猜中了目标数字。如图 a 所示，游戏结束时的 *remain* 值是 7，所以用 *max_stage* 减去 *remain*，也就是用 10减去 7，答案为 3。

▶因为 *max_stage* 的声明中已经指定了 **const**，所以 *max_stage* 的值无法变更。这样一来，如果把应该写成 *remain*-- 的部分写成了 *max_stage*--，就会发生编译错误（防止遗漏）。

a 猜125（猜到第3次就猜对了）

```
运行示例
请猜一个0~999的整数。

还剩10次机会。是多少呢：500
再小一点。
还剩9次机会。是多少呢：250
再小一点。
还剩8次机会。是多少呢：125
回答正确。
您用了3次猜中了。
```



b 猜139（猜了10次也没猜对）

```
运行示例
请猜一个0~999的整数。

还剩10次机会。是多少呢：500
再小一点。
还剩9次机会。是多少呢：250
再小一点。
还剩8次机会。是多少呢：125
再大一点。
还剩7次机会。是多少呢：187
再小一点。
还剩6次机会。是多少呢：156
再小一点。
还剩5次机会。是多少呢：140
再小一点。
还剩4次机会。是多少呢：133
再大一点。
还剩3次机会。是多少呢：136
再大一点。
还剩2次机会。是多少呢：137
再大一点。
还剩1次机会。是多少呢：138
再大一点。
很遗憾，正确答案是139。
```

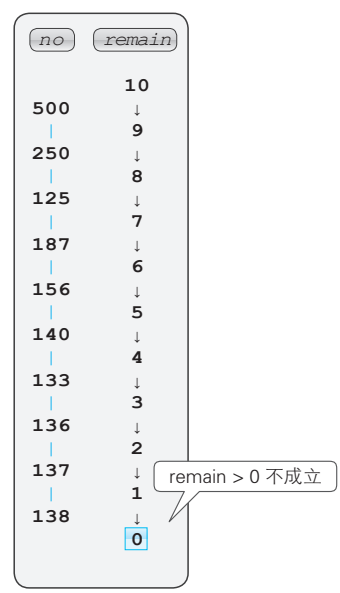


Fig.1-10 List 1-7的运行示例

专栏 1-2 逻辑运算和德・摩根定律

我们编写了一个限制玩家输入次数的“猜数游戏”程序，用于控制输入次数的 **do** 语句如 Fig.1C-5 a 所示。这个 **do** 语句即使像图 b 这样实现也一样能运行。

a List 1–7的do语句

```
do {
    /*… 省略 …*/
} while (no != ans && remain > 0);
```

“没有回答正确”
且
“还有剩余次数”

b 执行相同操作的do语句

```
do {
    /*… 省略 …*/
} while (!(no == ans || remain <= 0));
```

“答对了”
或者
“没有剩余次数了”
的否定

Fig.1C-5 do 语句的控制表达式

图 a 中用到了求逻辑与的逻辑与运算符 **&&**，图 b 中用到了求逻辑或的逻辑或运算符 **||**。Fig.1C-6 中总结了这些运算符的动作。

■逻辑与 如果 x 和 y 都为真，则结果为真

x	*y*	*x* && *y*
非 0	非 0	1
非 0	0	0
0	非 0	0
0	0	0

当 x 为 0 时不求值

■逻辑或 x 或 y 有一方为真，则结果为真

x	*y*	*x* || *y*
非 0	非 0	1
非 0	0	1
0	非 0	1
0	0	0

当 x 为非 0 时不求值

Fig.1C-6 用于求逻辑与的运算符 && 和用于求逻辑或的运算符 ||

C 语言中规定 0 以外的值为真，0 为假，因此对逻辑与运算符 **&&** 而言，如果两边的操作数都为真（0 以外的值）就会生成 1，否则就会生成 0。而逻辑或运算符 **||** 在操作数有一方为真（0 以外的值）时会生成 1，否则就会生成 0。

假设变量 *no* 读取的值是正确答案，此时表达式 *no* **!=** *ans* 的求值结果是表示假的 **int** 型的 0。因此不用特意去判断右操作数 *remain* **>** 0，也能得知控制表达式 *no* **!=** *ans* **&&** remain **>** 0 为假，也就是说，此控制表达式的值为 0。因为左操作数 *x* 和右操作数 *y* 两者只要有一方是 0，就说明整个逻辑表达式 *x* **&&** *y* 是假，即逻辑表达式 *x* **&&** *y* 为 0。

这样一来，如果程序对 **&&** 运算符的左操作数求值后得出的结果为 0，那么程序就**不会再对右操作数进行求值**（不再对表格中灰色部分的表达式 *y* 进行求值）。

|| 运算符也是一样。如果左操作数的求值结果为 1，那么程序就**不会再对右操作数进行求值**（不再对表格中灰色部分的表达式 *y* 进行求值）。因为如果 *x* 和 *y* 中有一方为真（0 以外的值），那么整个表达式都为真，即整个表达式的结果为 1。

这种只用左操作数的求值结果就可以确定整个表达式的求值结果，而不用对右操作数进行求值的情况称为**短路求值**（short circuit evaluation）。

*

现在回到 Fig.1C-5 的程序。图b的控制表达式中使用了逻辑非运算符 `!`。逻辑非运算符的动作如 Fig.1C-7 所示（表达式 `!x` 生成的值和表达式 `x == 0` 生成的值相同）。

■逻辑非　如果 x 为假则 y 为真

x	!*x*
非 0	0
0	1

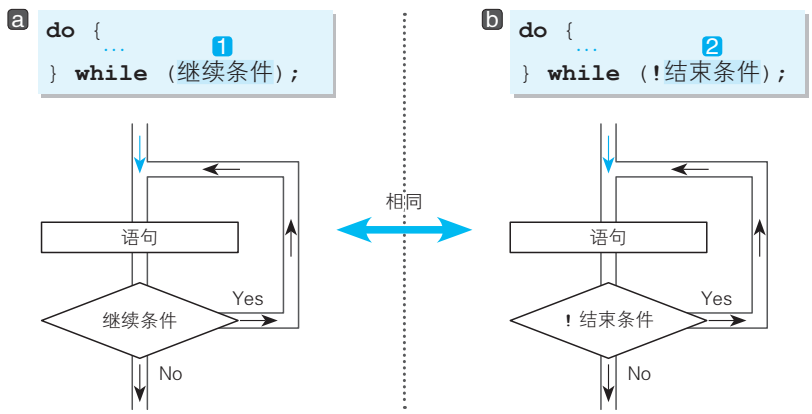
● Fig.1C-7　用于求逻辑非的 ! 运算符

对原命题取非，等于对该命题的各个子命题取非，再互换其中所有的逻辑与和逻辑或，这叫作**德·摩根定律**。该定律一般表示为下面这样。

① *x* `&&` *y* 和 `!(!`*x* `||` `!`*y*`)` 相等。
② *x* `||` *y* 和 `!(!`*x* `&&` `!`*y*`)` 相等。

图a的控制表达式 *no* `!=` *ans* `&&` remain `>` 0 是用于继续循环的“继续条件”，而图b的表达式 `!(`*no* `==` *ans* `||` remain `<=` 0`)` 是对结束循环的“结束条件”取非。

总之，上述概念如 Fig.1C-8 所示。



● Fig.1C-8　循环中的继续条件和结束条件

1-4 保存输入记录

如果程序能保存玩家输入的值，玩家就能在游戏结束时确认自己猜的数字距离目标数字有多近（或者有多远）。

数组

下面我们来把程序改良一下，令其能保存玩家已输入的数值，并在游戏结束时显示这些数值。改良后的程序如 List 1-8 所示。

►程序的运行示例可以在第 26 页看到。

本程序利用**数组**（array）来存储已输入的值。数组是一种将同一类型的变量排成一列的数据结构，数组内的各个变量就是数组元素。

在声明数组时，数组元素的个数必须是常量表达式。也就是说，下面这样的声明会引起编译错误。

```
int max_stage = 10;
int num[max_stage];     /* 错误：max_stage不是常量表达式 */
```

因此本程序中没有采用变量 *max_stage*，而设了一个对象宏 *MAX_STAGE*，将其声明为 1。

►编译初期，要把宏 *MAX_STAGE*（程序的 3 处灰色阴影部分）替换成 10。

接下来把存储所输入数值的数组 *num* 声明为 2。如 Fig.1-11 所示，数组 *num* 的元素类型是 **int** 型，元素个数是 10。

►因为这个声明 **int** *num*[*MAX_STAGE*]; 会被替换成 **int** *num*[10];，所以不会发生错误。

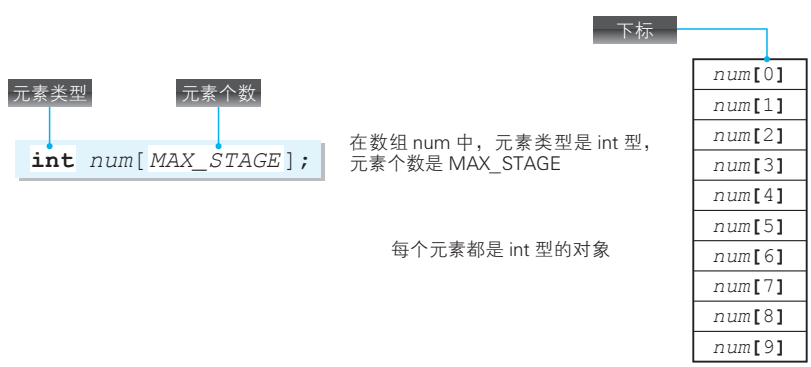


Fig.1-11 数组

List 1-8 chap01/kazuate5.c

```
/* 猜数游戏（其五：显示输入记录）*/

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

#define MAX_STAGE   10          /* 最多可以输入的次数 */

int main(void)
{
    int i;
    int stage;                  /* 已输入的次数 */
    int no;                     /* 读取的值 */
    int ans;                    /* 目标数字 */
    int num[MAX_STAGE];         /* 读取的值的历史记录 */

    srand(time(NULL));          /* 设定随机数的种子 */
    ans = rand() % 1000;        /* 生成0~999的随机数 */

    printf("请猜一个0~999的整数。\n\n");

    stage = 0;
    do {
        printf("还剩%d次机会。是多少呢：", MAX_STAGE - stage);
        scanf("%d", &no);
        num[stage++] = no;              /* 把读取的值存入数组 */

        if (no > ans)
            printf("\a再小一点。\n");
        else if (no < ans)
            printf("\a再大一点。\n");
    } while (no != ans && stage < MAX_STAGE);

    if (no != ans)
        printf("\a很遗憾，正确答案是%d。\n", ans);
    else {
        printf("回答正确。\n");
        printf("您用了%d次猜中了。\n", stage);
    }

    puts("\n--- 输入记录 ---");
    for (i = 0; i < stage; i++)
        printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);

    return 0;
}
```

1 替换成10

2

在数组的声明中，[] 里的值是**元素个数**，而 **[]** 里的值是**下标**（subscript），用于访问（读取）各个元素。

首个元素的下标是 0，之后的下标逐一递增，因此可以用表达式 *num*[0]，*num*[1]，…，*num*[9] 依次访问数组 *num* 的元素。由于末尾元素的下标值等于元素个数减 1，因此不存在 *num*[10] 这个元素。

数组 *num* 的各个元素和一般的（非数组的单独的）**int** 型对象具有相同的性质，能够赋值和获取值。

▶声明 **int** a[10]; 中的 [] 是用于声明的符号（标点），用于访问元素的 a**[3]** 中的 **[]** 是**下标运算符**（subscript operator）。

本书中将前者用 [] 表示，后者用**粗体**的 **[]** 表示。

把输入的值存入数组

让我们结合 Fig.1-12 来理解如何把玩家输入的值存入数组的元素中。

本程序中新引入的变量是 *stage*。这个变量用于代替 List 1-7 中表示剩余输入次数的变量 *remain*。游戏开始时其初始值为 0，之后玩家每输入一个数值，*stage* 的值都会逐次递增，当值等于 *MAX_STAGE*，也就是等于 10时，游戏结束。

负责把读取的值存入数组的正是图中的阴影部分。这里共有三个运算符，即 **[]**、**++**、**=**。

*

增量运算符 **++**（也称为递增运算符）包括前置形式的 **++***a* 和后置形式的 *a***++** 两种形式。我们先来了解一下它们都有哪些不同之处。

前置增量运算符 ++a

前置形式的 **++***a* 会在对整个表达式进行求值**之前**，先对操作数的值进行增量。因此当 *a* 的值为 3 时，运行以下代码的话，*a* 首先被增量成 4，然后程序会把表达式 **++***a* 的求值结果 4 赋给 *b*，最终 *a* 和 *b* 都等于 4。

```
b = ++a;                /* 先对a进行增量再赋给b */
```

后置增量运算符 a++

后置形式的 *a***++** 会在对整个表达式进行求值**之后**，再对操作数的值进行增量。因此当 *a* 的值为 3 时，运行以下代码的话，表达式 *a***++** 的求值结果 3 首先被赋给 *b*，然后程序会对 *a* 进行增量，增量结果为 4。最后的结果 *a* 等于 4，*b* 等于 3。

```
b = a++;                /* 先赋给b再对a进行增量 */
```

▶这里所说的前置和后置的求值时间也同样适用于进行减量操作的减量运算符 **--**。

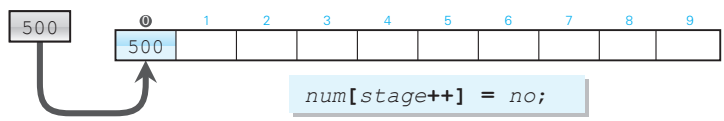
本程序的阴影部分中使用了**后置形式**的增量运算符。下面我们来了解一下玩家输入的值是如何一个一个地保存到数组元素中的。

▶ 1–4 节的 Fig.1-11 把数组的各个元素纵向排列，方框中写有访问各个元素的“表达式”。这次 Fig.1-12 则把各个元素横向排列，方框中写着各个元素的“值”，各个元素的下标是方框上面的小数字。

另外，实心圆符号●中所写的下标值和变量 *stage* 的值是一致的。

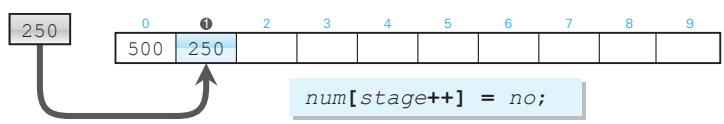
```
stage = 0;
do {
    printf("还剩%d次机会。是多少呢: ", MAX_STAGE - stage);
    scanf("%d", &no);
    num[stage++] = no;          /* 把读取的值存入数组 */
    /*… 省略 …*/
} while (no != ans && stage < MAX_STAGE);
```

ⓐ 存入玩家第1次输入的值（stage为0）



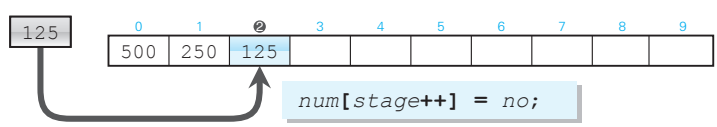
把 500 赋给 num[0] 后马上将 stage 增量为 1

ⓑ 存入玩家第2次输入的值（stage为1）



把 250 赋给 num[1] 后马上将 stage 增量为 2

ⓒ 存入玩家第3次输入的值（stage为2）



把 125 赋给 num[2] 后马上将 stage 增量为 3

… 以下省略 …

Fig.1-12 把输入记录存入数组中

ⓐ玩家输入 500，因为变量 *stage* 的值是 0，所以程序会把 500赋给 *num*[0]，再把 *stage* 的值增量为 1。

ⓑ玩家输入 250，因为变量 *stage* 的值是 1，所以程序会把 250赋给 *num*[1]，再把 *stage* 的值增量为 2。

通过反复进行上述处理，即可把玩家输入的值按顺序依次存入数组。

通过 for 语句来显示输入记录

一旦游戏结束，程序就会显示出玩家的输入记录。负责进行这项操作的就是下面这个 **for** 语句。

```
for (i = 0; i < stage; i++)
    printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);
```

可以像下面这样解释这个 **for** 语句所进行的循环。

> 首先把 *i* 的值设为 0，当 *i* 的值小于 *stage* 时，就不断往 *i* 的值上加 1，以此来让循环体运行 *stage* 次。

猜数游戏的主体 **do** 语句结束时，变量 *stage* 的值等于玩家输入数值的次数。如果玩家输入到第 7 次就猜对了，那么 *stage* 的值就是 7。此时通过 **for** 语句循环的次数是 7 次。

如 Fig.1-13 所示，在各个循环中，数组 *num* 内元素 *num*[*i*] 的下标是 *i*。实心圆符号●内的下标和变量 *i* 的值是一致的。

我们在循环体内通过 ***printf*** 函数来显示 3 个值。

① 第几次输入	*i* + 1
② 玩家输入的值	*num*[*i*]
③ 玩家输入的值与正确答案之差	*num*[*i*] - *ans*

运行示例

```
请猜一个0~999的整数。

还剩10次机会。是多少呢：500
再小一点。
还剩9次机会。是多少呢：250
再小一点。

… 省略 …

还剩4次机会。是多少呢：116
回答正确。
您用了7次猜中了。

--- 输入记录 ---
  1 :  500 +384
  2 :  250 +134
  3 :  125   +9
  4 :   62  -54
  5 :   93  -23
  6 :  108   -8
  7 :  116   +0
```

① 表示的是变量 *i* 加上 1 之后的值。下标是从 0开始的，而我们数数是从 1 开始的，加上 1 是为了弥补变量的值和显示的值之间的差距。

▶如图C，变量 *i* 的值是 2，加上 1 后显示结果就是 3。

② 则会直接显示出玩家输入的值 *num*[*i*]。

▶如图C，*num*[*i*] 也就等于 *num*[2]，显示结果是 125。

③ 表示的是玩家输入的值和正确答案之差，如果玩家输入的值大于正确答案，就在显示结果中加上符号“+”来表示，如果玩家输入的值小于正确答案，就在显示结果中加上符号“-”来表示。

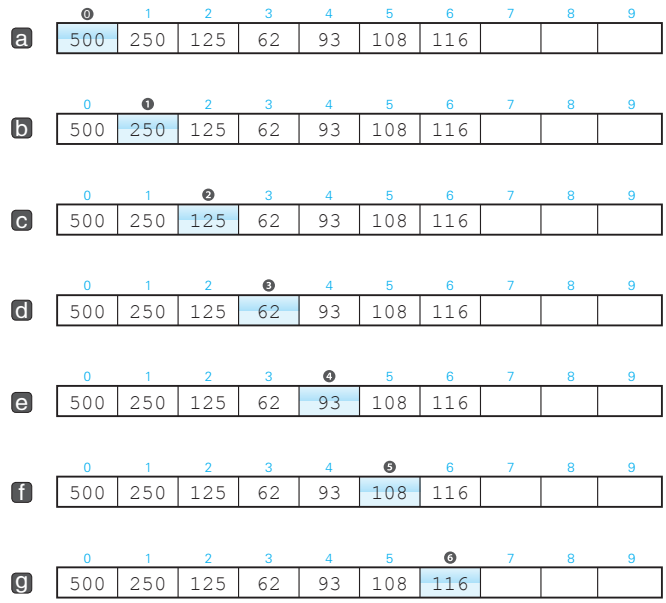
▶如图C，因为 *num*[*i*] 的值为 125，减去正确答案 116，得出差值为 9，显示结果就是“+9”。

大家都知道（通过平日的积累也应该有所了解），在使用格式字符串 "%d" 表示 **int** 型的数值时，只有当数值为负值时才会在数值前加上符号“-”。

一旦将格式字符串设为 "%d"，那么**数值即使是正值和 0 也会带有符号**。

▶我们将会在第 2 章详细学习 ***printf*** 函数和格式字符串。

```
for (i = 0; i < stage; i++)
    printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);
```



● Fig.1-13　通过遍历数组 num 来显示输入记录

按顺序逐个访问数组内的各个元素就叫作**遍历**（traverse）。这是一个基础术语，还请大家务必牢记。

*

接下来，**for** 语句会在变量 *i* 的值小于 *stage* 的期间一直循环。因此，**for** 语句结束时变量 *i* 的值就等于 *stage*，而不是 *stage* - 1。

▶把本程序的 **for** 语句改写成 **while** 语句时的代码如下所示。

```
i = 0;
while (i < stage) {
    printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);
    i++;
}
```

循环体会在变量 *i* 的值为 0，1，…，*stage* - 1 时运行，共运行 *stage* 次。最后调用 ***printf*** 函数时，变量 *i* 的值为 *stage* - 1。当这个值增量后等于 *stage* 时，控制表达式 *i* < *stage* 不成立，循环结束。

数组元素的初始化

我们再来详细学习一下数组。首先是用于初始化的声明。

要将元素初始化，需要对应各个元素把初始值按顺序依次排列并用逗号“,”一一隔开，再用“{}”把它们括起来。

例如像下面这样，一旦进行了声明，元素 a[0]、a[1]、a[2]、a[3]、a[4] 就会依次被初始化为 1、2、3、4、5。

```
int a[5] = {1, 2, 3, 4, 5};
```

下面是把所有元素初始化为 0的声明。

```
int a[5] = {0, 0, 0, 0, 0};      /* 把所有元素都初始化为0 */
```

但是，在给出了“{}”形式的初始值的数组声明中，**没有被赋予初始值的元素会被初始化为 0**。因此如果我们像下面这样声明的话，a[1] 之后没有被赋予初始值的所有元素都会被初始化为 0。这样看上去会更简洁一些。

```
int a[5] = {0};                  /* 把所有元素都初始化为0 */
```

▶对有静态存储期（5-3 节）的数组（包括在函数外定义的数组和在函数内加上 **static** 定义的数组）而言，即使不赋予该数组初始值，所有的元素也都会被初始化为 0。

在赋予了初始值的数组的声明中，可以**省略元素个数**。

```
int a[] = {1, 2, 5};             /* 省略元素个数 */
```

此时根据初始值的个数，数组 a 的元素个数被视为 3 个。也就是说，上面的声明和下面的声明是一样的。

```
int a[3] = {1, 2, 5};
```

另外，如果初始值的个数超过了数组的元素个数，程序就会报错。

```
int a[3] = {1, 2, 3, 5};         /* 错误：初始值太多了 */
```

此外，初始值 {1, 2, 3} 不能作为右侧表达式用于赋值，因此以下赋值会导致程序报错。

```
int a[3];
a = {1, 2, 3};                    /* 错误：不能这样赋值*/
```

▶关于初始值我们还会在后面的章节继续学习。

获取数组的元素个数

List 1-8 在声明数组以前把该数组的元素个数定义成了宏。

在一些不太适合用宏定义元素个数的情况下，首先要声明数组，再求元素个数。

求数组元素个数最常用的方法是使用 **sizeof** 运算符，List 1-9 中的程序就采用了这个方法。

List 1-9 chap01/array.c

```
/* 显示数组的元素个数和各个元素的值 */
#include <stdio.h>

int main(void)
{
    int i;
    int a[] = {1, 2, 3, 4, 5};
    int na = sizeof(a) / sizeof(a[0]);    /* 元素个数 */

    printf("数组a的元素个数是%d。\n", na);

    for (i = 0; i < na; i++)
        printf("a[%d] = %d\n", i, a[i]);

    return 0;
}
```

运行结果

```
数组a的元素个数是5。
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

如 Fig.1-14 所示，通过 **sizeof**(*a*) 可求出数组的大小，通过 **sizeof**(*a*[0]) 则可求出各个元素的大小。

int 型的大小根据编程环境的不同而有所不同，但通过 **sizeof**(*a*) / **sizeof**(*a*[0]) 求出的值是数组的元素个数，跟 **int** 型的大小无关。例如，如果 **int** 型是 2 字节，那么 **sizeof**(*a*) 就是 10，**sizeof**(*a*[0]) 就是 2，因此可求出元素个数等于 10 / 2，也就是 5。此外，如果 **int** 型是 4 字节，那么通过计算 20 / 4，可得到元素个数仍为 5。

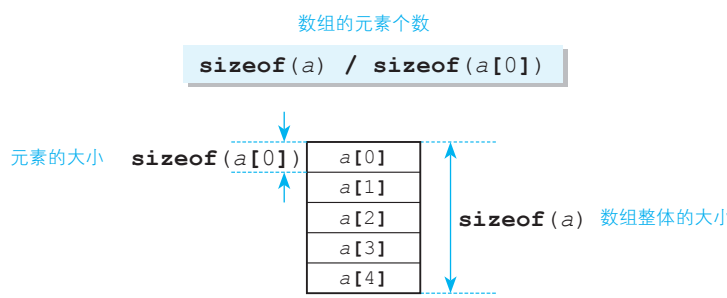


Fig.1-14 求数组元素个数的表达式

由前文可知，变量 *na* 会被初始化为数组 *a* 的元素个数 5。如果把数组 *a* 的声明进行如下变更，那么变量 *na* 就会被初始化为 6。实际的运行示例也是如此（如右图所示）。

```
数组a的元素个数是6。
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
```

```
int a[] = {1, 3, 5, 7, 9, 11};
```

不需要随着初始值的增减去**修改程序的其他地方**。

▶有些教材介绍的是采用 **sizeof**(*a*) **/ sizeof**(**int**) 而非 **sizeof**(*a*) **/ sizeof**(*a*[0]) 来求数组元素个数的方法，但这种方法并不可取。

各位想象一下，如果因为某种原因要变更元素类型的话，那我们该怎么办？假设“因为要存入数组元素中的数值超出了 **int** 型的范围，所以需要将元素类型变更成 **long** 型”，在这种情况下，就必须把表达式 **sizeof**(*a*) **/ sizeof**(**int**) 改成 **sizeof**(*a*) **/ sizeof**(**long**)。

采用表达式 **sizeof**(*a*) **/ sizeof**(*a*[0]) 就不必考虑元素类型。

小结

✻增量运算符和减量运算符

对操作数的值进行增量操作的增量运算符“**++**”，以及对操作数的值进行减量操作的减量运算符“**--**”都包括前置形式和后置形式。前置形式会在对表达式进行求值之前对操作数进行增量或减量操作，后置形式则会在对表达式进行求值之后再对操作数进行增量或减量操作。

✻数组

数组是一种将同一类型的元素排成一列的数据结构。声明数组时需要赋予数组元素类型和元素个数。此时元素个数必须是一个常量表达式。我们在访问各个元素时使用下标运算符“**[]**”，第一个元素的下标是 0。

数组的初始值的形式是对应各个元素把初始值按照顺序依次排列并用逗号“,”一一隔开，再用“{}”把它们括起来。

```
int a[] = {1, 2, 3};
```

✻数组的元素个数

一般情况下，即使我们不进行声明，也需要知道数组的元素个数，大家可以像下面这样声明。

① 事先用对象宏定义元素个数。

```
#define NA 7                          /* 先定义数组a的元素个数 */
int a[NA];
```

② 声明数组后获取元素个数。

```
int a[7];
int na = sizeof(a) / sizeof(a[0]);      /* 后获取数组a的元素个数 */
```

✍ 自由演练

建议大家不要满足于读懂本书中的程序，还要试着解答下述问题，自己来设计和开发程序，锻炼自己的编程能力。

* 因为是自由演练，所以没有答案。

练习 1–1

编写一个“抽签”的程序，生成 0~6 的随机数，根据值来显示“大吉”“中吉”“小吉”“吉”“末吉”“凶”“大凶”。

练习 1-2

把上一练习中的程序加以改良，使求出某些运势的概率与求出其他运势的概率不相等（例如可以把求出“末吉”“凶”“大凶”的概率减小）。

练习 1-3

编写一个“猜数游戏”，让目标数字是一个在 –999 和 999 之间的整数。

同时还需思考应该把玩家最多可输入的次数定在多少合适。

练习 1-4

编写一个“猜数游戏”，让目标数字是一个在 3 和 999 之间的 3 的倍数（例如 3, 6, 9, …, 999）。编写以下两种功能：一种是当输入的值不是 3 的倍数时，游戏立即结束；另一种是当输入的值不是 3 的倍数时，不显示目标数字和输入的数值的比较结果，直接让玩家再次输入新的数值（不作为输入次数计数）。

同时还需思考应该把玩家最多可输入的次数定在多少合适。

练习 1-5

编写一个“猜数游戏”，不事先决定目标数字的范围，而是在运行程序时才用随机数决定目标数字。打个比方，如果生成的两个随机数是 23 和 8124，那么玩家就需要猜一个在 23 和 8124 之间的数字。

另外，根据目标数字的范围自动（根据程序内部的计算）选定一个合适的值，作为玩家最多可输入的次数。

练习 1-6

编写一个“猜数游戏”，让玩家能在游戏开始时选择难度等级，比如像下面这样。

请选择难度等级 (1)1 ~ 9 (2)1 ~ 99 (3)1 ~ 999 (4)1 ~ 9999:

练习 1-7

使用 List 1-8 的程序时，即使玩家所猜数字和正确答案的差值是 0，输入记录的显示结果也会带有符号，这样不太好看。请大家改进一下程序，让差值 0不带符号。

练习 1-8

把 List 1-8 里的 **do** 语句改写成 **for** 语句。