第八章　文件和输入输出

程序处理的数据从何而来，得到的结果送到哪里去？输出输入解决这方面的问题。程序总需要输出输入，我们前面的程序都通过标准输入输出与外界交互。这种方式可以完成许多程序，但有时我们也希望程序能使用系统中的命名文件。本章将讨论文件的一般性概念及 C 程序里的文件使用，并介绍输入输出格式控制的细节。这里还要介绍一些相关问题和概念，并通过程序设计实例，讨论输入输出中的一些问题和有用技术。

8.1 文件的概念

存于变量中的数据至多保存到程序结束，因为变量是程序内部的东西，其生命周期不能跨越程序的两次不同执行，不能将一次执行得到的信息带到程序的另一次执行。当某程序被启动时，操作系统将为它分配一块存储区。程序结束后该存储区完全可能分给其他程序使用。在每次执行开始时，所有外部变量都将初始化，其值与程序的前面执行毫无关系。此外，由于目前计算机内存器件的特性，存于其中的数据在关机后将立刻消失。由于这些原因，为了*持续性*地保存数据，就必须借助外存设备，如磁盘、磁带等。这样，写程序时也就需要了解如何访问和使用外存，程序语言也必须提供这方面的功能。

在目前的计算机系统里，外存信息都通过目录和文件方式组织起来，构成操作系统管理下的外存信息结构。目录可看作是子目录和文件的集合，文件是封装起的一组数据。每个目录或文件有名字，可以通过名字被操作和使用。程序执行中与外存打交道，主要就是访问使用作为外存信息实体的文件。

程序向外传送信息的操作是*输出*，从外部取得信息的操作是*输入*。输出输入操作可以是文件，也可以是一些标准设备，如键盘、显示器、打印机或者其他设备。许多操作系统都采用统一的观点，把所有与输入输出有关的操作都统一到*文件*的概念中，程序与外部的联系都通过文件概念实现。常常把键盘、显示器等设备也看作文件，甚至给定了“文件名”，对它们的操作都通过相应文件名进行。

C 语言本身没有专用于输入输出的语言结构。为了提供一种统一标准，ANSI C 把文件和输入输出功能作为标准库的一部分，以提高程序的可移植性。标准库将所有与输入输出有关的机制都统一到文件的概念中，定义了一些与输入输出有关的数据结构，提供了一组与输入输出有关的操作。

8.1.1 流和文件指针

标准库对文件输入输出采用的概念称为*流*。一个文件或者是信息的来源，或者是接受信息的目标，总之是输入输出操作的对象。为能与这种对象交换信息，就需要建立与它们联系，*流*就是这种联系。为了从一个已有的文件输入信息，程序就需要创建一个与该文件关联的输入流，建立一条信息输入通道。同理，要想向一个文件输出，就要建立一个与之关联的输出流。有时还可能建立既能输入又能输出的流。这种建立联系（创建流）的动作被形象地称为*打开文件*，文件被打开后就可以进行操作了。图 9.1 是这一情况的形象描述。当一个文件不再需要时，程序可以切断与它的联系，撤消有关的流，这称为*关闭文件*。打开和关闭文件都是文件处理的基本操作。

标准库的流分为两类：正文流（或称为字符流）和二进制流。正文流把文件看作行的序

列，每行包含 0 个或多个字符，一行的最后有换行符号'\n'。正文流适合一般输出和输入，包括与人有关的输入输出。二进制流用于把内存数据按内部形式直接存储入文件。二进制流操作保证，在写入文件后再以同样方式读回，信息的形式和内容都不改变。二进制流主要用于程序内部数据的保存和重新装入使用，其操作过程中不做信息转换，在保存或装入大批数据时有速度优势，但这种保存形式不适合人阅读。

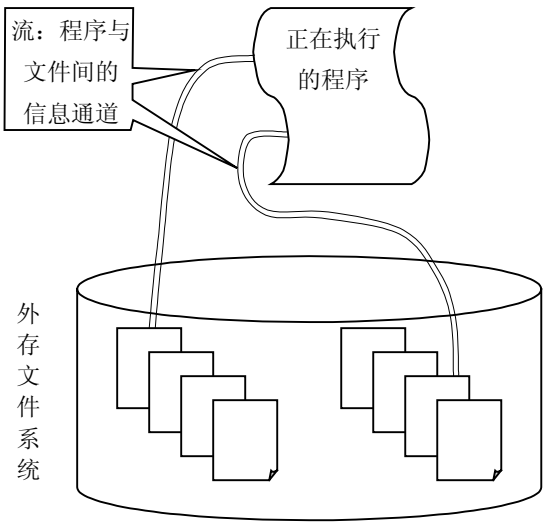

图 8.1　程序与文件的连接

标准库提供了一套流操作函数，包括流的创建（打开文件）、撤消（关闭文件），对流的读写（实际上是通过流对文件的读和写），以及一些辅助函数。

流通过一种特殊数据结构实现，标准库为此定义了类型 FILE，其中存储与流操作有关的（与打开的文件有关的）所有信息*。打开文件操作将返回一个指向 FILE 的指针（称为*文件指针*），它代表所创建的流，对这个流的所有操作都将通过这个指针进行。因此也可以认为文件指针就是流的体现，人们也把文件指针作为流的代名词。

C 程序启动时自动创建三个流（建立三个文件指针并指定值）：标准输入流（指针名为 stdin）、标准输出流（stdout）和标准错误流（stderr）。stdin 通常与操作系统的标准输入连接，stdout 与操作系统的标准输出连接，stderr 通常直接与显示器连接，这说明 stderr 不能重新定向。前面程序所用的标准输入输出操作都是对这些流进行的。

8.1.2 缓冲式输入输出

在介绍各种文件操作之前，先简单介绍一下标准库文件功能的实现方式。标准库定义的输入输出称为*缓冲式*输入输出，这是一种常用的输入输出方式。由于外存（磁盘、磁带等）速度较慢，一般采用成块传递方式，一次传递一批数据。而程序里对数据的使用则往往不是这样。为了缓和两者间在数据提供和使用方面的差异，人们提出开辟一块存储区（称为*数据缓冲区*，简称*缓冲区*），作为文件与使用数据的程序之间的传递媒介。

以输入流操作为例，其工作方式如图 9.2 表示，在程序与文件间的传输通道上设置了一个缓冲区。文件中的数据将以成块方式复制到缓冲区；程序需要读入数据时就由缓冲区读取，不必每次访问外存，这样可以大大提高程序的工作效率。如果程序要求读取数据时缓冲区的数据已用完，系统就会自动执行一个内部操作，从文件里取得一批数据，将缓冲区重新填满。此后程序又可以按照正常方式读取数据

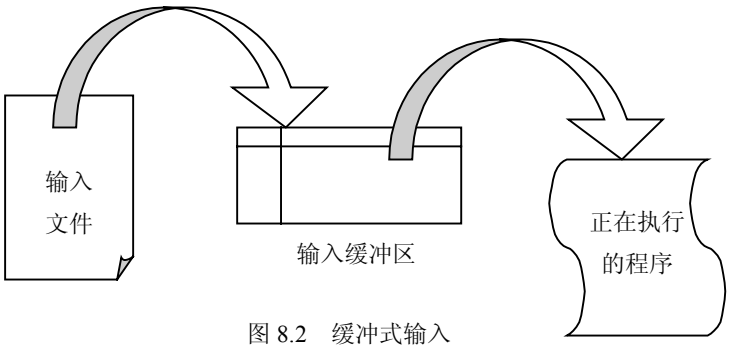

图 8.2　缓冲式输入

* 我们不必关心 FILE 类型的具体定义，应把它看成一种抽象东西。程序里不需要直接操作这种类型，只需（也只应该）通过标准库函数使用它。在程序工作（和许多与计算机有关的工作）中采取这种思维方式非常重要，这也是一种超脱。如果我们希望关心一切细节，那么就不可能处理复杂的东西了。

了。缓冲方式可以较好地弥合程序与外存在数据操作方式和速度方面的差距。输出操作的处理方式与此类似，只是方向相反：每当缓冲区装满后自动执行一次对文件的成块写操作。

标准库文件操作函数实现缓冲式的输入输出功能。在打开文件时，系统自动为所创建的流建立一个缓冲区（一般通过动态存储分配），文件与程序间的数据传递都通过这个缓冲区进行。文件关闭时释放缓冲区。应该看到，虽然在程序与外存间有这样一个缓冲区，但从操作效果看，这个中间过程却像不存在似的（透明的），程序就像直接在与外存打交道。这种透明性的思想在计算机领域里非常重要，是许多方面各种设计的基础，在许多领域都能看到。

下面两节将首先介绍一般文件的使用问题和有关的标准库函数。随后以标准流的输入输出函数为例，进一步介绍输入输出的格式转换控制。

8.2 文件的使用

要把外存文件作为输入输出对象，一个可能方式是通过标准输入输出的重新定向，把标准流转接到指定文件。这样做能解决一些问题。但这种做法有很大局限性，因为这样形成的定向在程序执行期间不能改变。为能在程序中方便地根据需要使用各种文件，就必须利用标准库的文件操作函数，通过为有关文件建立特定输入输出流的方式使用它们。

8.2.1 文件的打开和关闭

打开文件的操作通过标准库函数 fopen 完成，函数 fopen 返回一个 FILE 类型的指针值。为了使用打开的文件，程序里需要用文件指针变量接受 fopen 的返回值。在文件打开操作完成后，通过这种指针就可以进行文件操作了。函数 fopen 的原型是：

```
FILE *fopen(const char *filename, const char *mode);
```

其中 filename 的实参是字符串，表示希望打开的文件名；mode 是另一字符串，用于指定文件打开方式。这一字符串中可用的字符包括 r，w，a 和 +，分别表示读、写、附加和更新。另可加字符 b 表示以二进制方式打开文件。字符串里可以写它们的合理组合。常用的文件打开方式有：

"r"	以读方式打开文件，如找不到文件，则打开失败
"w"	以写方式打开，如文件已有则丢弃原有内容
"a"	添加方式打开或创建文件，从文件已有部分后面接着写
"r+"	读更新方式，可以对文件读或者写
"w+"	写更新方式，可以写或者读，如文件存在则先丢弃原有内容
"a+"	添加并可读方式，从文件尾接着写

这里是若干说明：1）当文件打开操作不能正常完成时，函数 fopen 返回空指针值。由于文件打开操作是与程序外部打交道，操作能否完成依赖于程序运行的环境。所以，在文件打开操作之后必须检查函数的返回值，以确保后续操作的有效性。显然，对空的文件指针操作不会有任何意义。2）上面例子中的写法都是以正文文件方式打开文件。若需要以二进制方式打开文件，就需要在模式串中加字符 b 说明。例如，"rb"、"wb+"、"a+b"分别表示以二进制读方式、二进制写更新方式、二进制添加并可读方式打开文件。3）对于以读写方式打开的文件，在读操作和写操作之间切换时，必须做文件重新定位，并需要调用函数 fflush 刷新流的缓冲区。这方面情况在本章有关辅助函数的一节里有进一步介绍。

关闭文件通过函数 fclose 完成，它的原型是：

```
int fclose(FILE * stream);
```

该函数完成关闭流的所有工作。对于输出流，fclose 将在实际关闭文件前做缓冲区刷新，即把当时缓冲区里所有数据实际输出到文件（无论缓冲区满不满）；对输入流，文件关闭将

丢掉缓冲区当时的内容。在关闭操作中还要释放动态分配的缓冲区。fclose 正常完成时返回 0，出问题时返回值为 EOF。

请注意，一个程序可以同时打开的文件数通常是有限的。所以，文件使用完毕后应及时将它关闭。程序结束时，所有尚未关闭的文件都将被自动关闭。

下面程序段反映了文件使用的一般过程：

```
FILE *fp;
fp = fopen("myfile.abc", "r");
if (fp == NULL) {
    /* 当文件打不开时的处理 */
}
... ... /* 对文件的各种操作 */
fclose(fp);
```

有时也可能需要由用户那里得到文件名，下面是程序示例：

```
int main() {
    char fname[128];/* 有越界危险 */
    FILE *fp;
    ... ...
    printf("File name: ");
    scanf("%s", fname);
    if((fp=fopen(fname,"r"))!=NULL){ ... ... }
    ... ...
}
```

如果必须得到正确的文件名，也可以用循环：

```
while (1) {
    printf("File name: ");
    scanf("%s", fname);
    if ((fp = fopen(fname,"r"))!=NULL) break;
    printf("File name error!\n");
}
```

8.2.2 输入输出函数

下面介绍针对正文流的各种输入输出函数。

字符输入和输出函数

基本字符输入和输出函数 fgetc 和 fputc，其原型分别是：

```
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
```

这两个函数分别从指定的流读一个字符，或向指定的流写一个字符，它们的返回值就是所读或写的那个字符。遇到文件结束时 fgetc 返回值 EOF，操作出错时两个函数都返回 EOF。请注意，这两个函数的返回值都是 int 类型，其中的原委在前面讨论 getchar 时已经有详细说明。

getc 和 putc 的功能与上述函数类似，只不过它们是通过宏定义实现的。前面章节里使用的标准流字符读写函数通常借助于这两个宏定义，标准库文件里通常有下面定义：

```
#define getchar()  getc(stdin)
#defien putchar(c) putc(c, stdout)
```

这些字符输入输出函数的使用都与我们已经熟悉的 getchar 和 putchar 类似。

此外，还有一个与字符输入操作有关的函数 ungetc，其原型是：

```
int ungetc(int c, FILE* stream);
```

利用这个函数可以将一个字符退回到流 stream 里，它保证下一次对 stream 读时第一个遇到的就是这个退回的字符。标准库只保证能退回一个字符，正常情况下 ungetc 的返回值为字符 c，出错时返回 EOF 值。提供这一函数的原因是有时需要看到后面的字符才知道一段输入结束了，而最后这个字符并不是本段输入的一部分，因此应该退回去。

格式化输入输出函数

文件的基本格式化输入输出函数 fscanf 和 fprintf 的原型分别是：

```
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

它们在功能和使用上都与 scanf、printf 相对应，只是增加了指定输入或输出对象的流参数。两个函数的第二个参数都是格式描述串，其形式和意义分别与函数 scanf 和 printf 相同。读者对 scanf、printf 已经比较熟悉，因此也就不难理解这两个函数了。前面介绍了格式转换描述的一些常用情况，下一节有对格式控制的更详细讨论。

行式输入和输出函数

标准库还提供了两个以行为单位进行输入和输出的函数。调用行式输入输出函数时，需要用一个字符数组保存输入或输出的信息。

行式输入函数的原型是：

```
char *fgets(char *buffer, int n, FILE *stream)
```

其中 buffer 应是一个字符数组的地址。fgets 由流 stream 读入至多 n-1 个字符，将它们存入 buffer 指定的字符数组里。读入过程遇到换行就结束，换行符号也存入数组。无论操作如何完成，函数都在数组中已存入的字符之后存放一个'\0'（做成字符串形式）。正常完成时函数返回参数 buffer，遇文件结束或操作出错时返回空指针。显然这里要求数组 buffer 至少能容纳 n 个字符。为防止数组越界，参数 n 的值必须符合有关数组的情况。

行式输出函数的原型是：

```
int fputs(const char *buffer, FILE *stream)
```

这个函数将 buffer 里的字符串送到流 stream，最后不向流中添加换行符号（输出字符串里可包含换行符，也可以没有）。函数正常完成时返回非负值，出错时返回 EOF 值。

由于这两个函数都与字符串有关，也可以将它们看作一对字符串输入输出函数。当然，fgets 的输入单位是行。

8.2.3 程序实例

例 1，写程序 cat。没有命令行参数时它完成由标准输入向标准输出的复制；如果有参数，cat 把所有参数作为需要复制的文件的名字，把这些文件顺序复制到标准输出。

利用上面介绍的文件操作函数及命令行机制，这个程序不难写出：

```
#include <stdio.h>

void filecopy(FILE *ifp, FILE *ofp) {
    int c;
    while ((c = fgetc(ifp)) != EOF) fputc(c, ofp);
}

int main(int argc, char *argv[]) {
    FILE *ifp;
    char *name = argv[0];
    if (argc == 1) { /* 没有参数，从标准输入复制到标准输出 */
        filecopy(stdin, stdout);
        return 0;
    }
    while (*++argv != NULL)
        if ((ifp = fopen(*argv, "r")) == NULL)
            printf("%s, can't open input file: %s\n", name, *argv);
        else {
            filecopy(ifp, stdout);
            fclose(ifp);
        }
    return 0;
}
```

例 2，假定文件里保存了一批货物单价和数量数据。写一个程序，通过命令行参数为它提供文件名，它最终输出货物的总货值。

假设文件里的数据以一个单价和一个数量的方式成对出现，我们应该用 fscanf 读入。这里考虑把读入一对数据的工作写成函数 nextentry，在发现所有数据都处理完毕后，让它返回一个 0 值，通知调用的地方。这里没有特别处理数据可能出错的情况，有关修改可以参考前面的讨论，留给读者作为练习。

主函数也很简单：首先检查命令行参数个数是否正确，而后打开文件。在遇到错误无法正常执行时输出错误信息并终止程序，也用返回值报告出现了问题。

```
#include <stdio.h>

double nextentry(FILE *fp) {
    double pr, num;
    int n = fscanf(fp, "%lf%lf", &pr, &num);
    return n == EOF ? 0 : pr*num;
}

int main(int argc, char** argv)
{
    double total = 0, x;
    FILE * ifp;
    if (argc == 1) { /* 没有参数，给用户提供错误信息 */
        printf("Missing data file name. Stop!\n");
        return 1;
    }

    if ((ifp = fopen(argv[1], "r")) == NULL) {
        printf("Can't open file: %s. Stop!\n", argv[1]);
        return 2;
    }

    while ((x = nextentry(ifp)) != 0.0)
        total += x;

    printf("Total price: %f\n", total);
    fclose(ifp);
    return 0;
}
```

上面两个程序展示了处理文件的程序的基本框架。当然，这里都用命令行参数提供文件名。提供文件名的另外两种常见方式是：直接将文件名写在程序里，或者通过与用户的交互取得文件名。修改上面的例子，采用这两种方式都很简单，请读者自己练习。

8.2.4 直接输入输出函数

前面反复讲过，各种文件格式化输入输出函数（如 fscanf、fprintf）在工作过程中，都要做数据的内部形式与外部形式之间的转换。例如要输出一个整型变量的值，这个值的内部形式是固定位数的二进制位序列，而实际送到外部的是一串数字字符。这种形式转换就需要由格式化输出函数完成。

完成数据形式转换要花时间。如果产生的输出是给人看，或者输入数据来自人易读的形式，那么这种转换是必需的。但如果输出到文件的目的只是为了以后重新取回程序里使用，例如为了在程序运行的后面阶段中使用，或者为了在程序下次执行时重新装入使用，那么做这种转换就没有必要了。此外，有时数据转换还会丢失信息，尤其是对实数类型的数据，转换输出可能产生误差，再输入又可能产生误差，这样得到的数据已不是原来的数据了。为能更好地解决这类问题，标准库提供了二进制流和直接输入输出函数。

两个直接输入输出函数的原型分别是：

```
size_t fread(void *pointer, size_t size,
             size_t num, FILE *stream);
size_t fwrite(const void *pointer, size_t size,
              size_t num, FILE *stream);
```

前面已经反复讲过，size_t 是语言中的某个无符号整型，具体情况由 C 系统确定。函数 fwrite 向流 stream 输出一批数据（应该是某个数组的一批元素），数据的起始位置由指针 pointer 给定，元素大小是 size，共 num 个。这些数据元素将顺序存入与流 stream 相关联的文件里。

函数 fread 的功能正好与 fwrite 对应，它要求读入 num 个数据元素，每个元素的大小为 size，指针参数 pointer 应指向接受数据的起始存储位置。显然，此时 pointer 应指定一个数组，数组元素的类型应该与以前向文件直接输出时所用元素类型一致，数组的大小至少应是 num。标准库的设计保证，以某一确定方式（某种元素大小和个数）直接存入文件的信息，再用同样的方式读回来，所得到的数据内容不变。

函数 fwrite 返回实际写出的数据元素个数，如果这个数小于 num，那就说明函数执行中出现了错误。fread 返回实际读入的元素个数。对于 fread 操作，应当用另一个标准函数 feof 检查是否读到了文件结束（该函数在文件结束时返回非 0 值）。

下面是一个说明直接输入输出函数使用方式的例子。假设 datatable 是一种 100 个双精度数的数组类型：

```
enum { TBLEN = 100 };
typedef double datatable[TBLEN];
datatable m, n;
```

假设在程序里通过计算已经给数组 m 的元素设置了值，而且现在需要把这个数组整体存入一个文件。假设文件已经打开，建立的二进制输出流名为 msave。保存整个数组 m 里全部信息的工作可以通过下面的一次函数调用完成：

```
fwrite(m, sizeof(double), TBLEN, msave);
```

如果在程序里另一处需要把原来存储的同样形式的数据重新装入，存入数组 n。假设已经为这一输入工作，针对相应文件建立了流 msaved。下面函数调用将完成这一输入工作，整型变量 num 用于接受函数的返回值：

```
num = fread(n, sizeof(double), TBLEN, msaved);
```

读入正常完成后，数组 n 的内容将与原来存文件操作时 m 的内容完全一样。

请注意，由于函数 fwrite 用直接方式进行输出，用它建立的文件一般无法用普通的编辑器查看和修改。直接输入输出操作通常在比较复杂的应用程序里使用。对于初学者所做的小程序而言，基本上没有使用这种函数的必要性。

8.3 标准流输入输出与格式控制

本节首先介绍另外几个标准流的输入输出函数，而后以 printf 和 scanf 为例，详细介绍格式化输入输出函数的格式控制问题。

8.3.1 行式输入和输出

对于标准输入和标准输出流也有一对行式输入和输出函数。它们的原型分别是：

```
char *gets(char *s)
int puts(const char *s)
```

函数 gets 的参数 s 应是一个字符数组的开始地址。gets 从标准输入读一个完整的行（从标准输入读，一直读到遇到了换行字符），把读到的内容存放入由 s 指定的字符数组里，并用空字符'\0'取代行尾的'\n'，最后返回指针 s（存储信息的开始位置）。如果执行中出现错误或者遇到文件结束，gets 就返回空指针值 NULL。

函数 puts 的参数应当是一个存着字符串的字符数组。puts 将该字符串的内容以及一个换行符送到标准输出。puts 正常完成时返回一个非负值，出错时返回 EOF。

这两个函数一般使用的形式如下：

```
char s[256];
...
if (gets(s) != NULL) ...
...
if (puts(s) != EOF) ...
```

写测试是为了能处理输入输出操作没有正常完成的情况。

函数 gets 的一个问题无法防止输入数组的可能越界问题。与前面一般性的文件行式输入函数相比，gets 缺少长度控制参数，因此使用中就无法控制对字符数组的写入长度。由于输入来自程序外部，我们根本无法预料外部提供的一行到底有多长，因此上述代码有安全问题。在实际中，不少程序和软件存在漏洞的原因就在这个方面，甚至某些关键系统的安全漏洞也源于此。由于这种情况，许多有经验的程序工作者提出不应该在程序里使用 gets。如果需要从标准输入流按行输入，更安全的是用：

```
fgets(s, 256, stdin);
```

假定 s 是上面定义长度为 256 的 char 数组，这一写法就可以保证不会出现越界问题。

用 fgets 代替 gets 还要注意它们功能上的一点差异。gets(s) 在读入一行时将最后的换行符号用 '\0' 取代，而 fgets(s, 256, stdin)将换行符也存入字符数组 s，后面再加字符 '\0'。函数 puts 和 fputs 之间也有类似的不同，puts(s)输出字符串 s 后还会输出一个换行符，而 fputs(s, stdout)只输出字符串 s。

8.3.2 输入格式控制

scanf 是前面程序中反复使用的标准库函数。这里将以它为例，详细介绍标准库格式化输入函数的格式控制问题，其他各种格式化输入函数（如文件输入函数 fscanf 等）在格式描述方面的规定都与 scanf 完全一样。

前面章节里一直没有给出函数 scanf 的原型，其原型如下：

```
int scanf(const char *format, ...)
```

可以看到参数表最后的三个圆点，这是 C 语言函数参数的一种特殊描述方式，它表示这个函数除了所列出的参数 format 外，还可以有任意多个其他参数。在 C 语言里用这种方式定义具有可变数目参数的函数。如果我们写程序时需要定义这种函数，就必须用同样方式写函数头部，而后借助标准库头文件<stdargs.h>提供的功能取得实际参数的值，有关情况在后面讨论标准库功能的章节里有详细介绍。scanf 是一个实参数目可变的函数，它应该有一个字符串形式的格式描述串，而后可以根据需要有任意多个其他参数。

对于以上述方式定义的函数，有一点特别值得注意：对于函数原型里由...代表的那些实参，编译程序不能做任何类型检查，也不会根据情况确定所需的类型转换。正因为这样，本书前面一直强调 scanf、printf 的实参必须写正确，应是什么类型就必须写什么类型。如果实参的类型不合适，编译系统不会发现，不会转换，有不会产生错误信息。而在程序执行中，即使输出时不出现动态运行错误，由实参取得的值也不会是正确的。

在输入处理过程中，scanf 把输入流中的信息看成由空白字符（空格、制表符、换行符等）分隔的一个个字段，其读入过程就是顺序地处理这些字段。格式描述串参数 format 描述了程序所要求的转换方式，它控制着 scanf 的读入过程。scanf 把转换成功时得到的值赋给相应变量，这些变量的地址由写在格式串后面的参数指定。scanf 一直执行到处理完整个格式描述串，或是遇到转换失败，一般情况下它返回成功完成转换的项数。

在 scanf 的格式串 format 里可以有各种字符，其意义与作用如下表所述：

字符等	作　用
空白字符	包括空格、制表符、换行符。它们将被忽略，但也会导致 scanf 抛弃掉读入中遇到的所有空白字符，直至遇到非空白字符。
普通字符	遇到除字符 % 外的非空白字符，scanf 将它与输入流中的下一非空白字符匹配，字符相同则匹配成功。这里有可能出现匹配失败的情况。
转换描述	转换描述由字符 % 开始的若干个字符组成。% 字符之后可以有：一个星号*，表示只进行匹配和转换，不向参数赋值；一个字段长度描述，表示这个转换应处理的输入字符个数；一个对赋值目标的长度指示字符（字母 h、l 或 L）；最后是转换字符本身。各种转换字符的意义见下面表格。

一个转换描述说明了输入中下一字段的转换方式。如果转换能顺利完成，scanf 就把转换的结果赋给对应参数所指定的变量（在转换描述中无星号时）。如果有关转换描述中包含了字段长度，scanf 就会把输入流中指定数目的字符作为当前字段。如果在转换字符前面有星号（如转换描述“%*s”、“%*d”等），那么 scanf 就把将这一转换描述所匹配的字段直接丢掉，不做赋值。

下表列出了各个转换字符的意义，也包括了它们所要求的实际输入、以及对应的参数所应该具有的类型：

转换字符	要求的输入数据形式	要求的参数
d	十进制形式的整数	int*
i	整数，可以是十进制表示（起始数字非 0），八进制表示（由数字 0 开始），或者十六进制表示（由 0x 或 0X 开始）	int*
o	八进制表示的整数，可以有或者没有先导的数字字符 0	int*
u	无符号十进制整数	unsigned*
x	十六进制表示的整数，可以有或者没有先导的 0x 或 0X	int*
c	字符。若指定输入宽度，这个转换可以将多个字符输入到字符数组里。读字符过程中不跳过空白字符，读入多个字符时不加'\0'	char*
s	读入一个非空白字符序列，可以有长度限制。读入后在字符数组的最后加空字符'\0'（做成字符串）。作为参数的字符数组应当足够存放读入的所有字符和结尾的'\0'	char*
e,f,g	符合 C 语言规定形式的浮点数	float*
p	指针值，其形式与用 printf("%p", ...)输出形式的一样。这使人可以把通过 printf 输出的指针值重新读回程序里。	void*
n	向对应参数中写入本次函数调用执行到此已经读的字符个数。处理这一“转换描述”时不读入字符，也不计入转换的项数	int*
[...]	与输入流里由方括号中列出的字符形成的最长字符序列匹配，将这些字符写入由参数确定的字符数组里，并附加一个'\0'。 可用[]...]的形式表示被匹配字符串里也包含“]”	char*
[^...]	与输入流里不包括方括号中列出的任何字符的最长字符序列匹配，将这些字符写入由参数确定的字符数组里并附加一个'\0'。 可用[^]...]的形式表示被匹配字符串里不能包括“]”	char*
%	与输入流中的字符%匹配，没有赋值操作	--

在转换描述字符 d、i、o、u、x 之前可以加一个字符说明赋值目标长度，加 h 表示被赋值的是 short 变量；加 l 表示被赋值的是 long 变量。转换描述字符 e、f、g 前也可以加字符说明，加 l 表示被赋值的是 double 变量，加 L 表示被赋值的是 long double 变

量。这些字符要求函数 scanf 按照指定的类型去构造值并完成赋值。

例，假设有函数调用 scanf("...%ld...", ..., &ii, ...)，当时函数处理到转换描述串%ld，scanf 的处理动作是（上面的...表示省略了某些东西）：

1. 由于%ld 要求的输入是十进制数字序列，scanf 将跳过在输入流中遇到的所有空白字符（可以有多个，也可以没有），从遇到的第一个非空白字符开始做实际匹配和转换。
2. 如果遇到的第一个非空白字符不能看作数的开始（不是字符 0～9，也不是正负号），匹配失败，scanf 返回至此已成功完成的转换项数，输入流指示器停在这个未能成功匹配的字符处，该字符也留给随后的输入使用。
3. 如果遇到的第一个非空白字符可看作数的开始。scanf 就逐个读入字符，直至遇到第一个不能是数的部分的字符为止。读入的这些字符（可能包含正负号及一个数字字符序列）根据%ld 的要求做成一个长整数的内部形式，然后赋给指定变量 ii。

在 scanf 执行中，如果格式串 format 用完，或者遇到实际读入数据与 format 描述不匹配而无法进行转换，或者在执行中出现错误，本次 scanf 的执行都结束。

如果在没完成任何转换之前出错，或没完成任何转换前遇到文件结束，函数返回 EOF 值；在其他情况下都返回正确完成转换及赋值的数据项数（一个非负整数值）。返回值 0 表示在第一个转换时匹配失败。应当特别注意，当 scanf 转换匹配失败时，导致失败的输入字符仍留在输入流里，下一次调用输入函数时将首先读到这个字符。这种情况有时会引起一些我们不希望的后果。例如有下面程序片段：

```
while (scanf("%d", &n) != EOF) {
    ... /* 对输入数据的处理 */
}
```

写这段程序是想读入一系列整数，遇到文件结束时退出循环。但是这个写法不安全。假如人在输入数据时不慎键入一个字母，例如 p（它不能看成整数的部分），就会导致这个程序陷入无穷循环。为什么呢？因为 while 条件里的 scanf 读到字符 p 时导致转换失败，但函数返回值不是 EOF（返回 0），这一循环不会结束，而字符 p 却仍然留在输入流里。下一次再执行函数 scanf，转换又失败，返回值仍然不是 EOF。这就使程序进入了无穷循环。

这个例子说明，处理输入必须注意 scanf 可能匹配失败。不检查 scanf 的返回值是一种盲目做法，匹配失败时并没有真正的输入，继续处理就完全没有价值了。检查返回值时也需要注意，不当的检查方式也可能引起问题。把上面程序段改造为：

```
while (scanf("%d", &n) > 0) {
    ... ...
}
```

或者前面一直使用的：

```
while (scanf("%d", &n) == 1) {
    ... ...
}
```

就不会出现无限循环了。请注意理解这种写法的效果，此时实际输入中任何无法转换的字符都导致循环立刻结束。有时这种行为方式不符合我们的需要，此时可能需要写：

```
while ((k = scanf("%d", &n)) != EOF) {
    if (k == 0) {
        /* 对出现不合理字符的处理 */
    }
    ... ...
}
```

总而言之，通过检查 scanf 的返回值可能发现输入中出了问题，例如出现不合要求的字符。在确定情况后可以根据需要处理。例如读入并丢掉一些字符，直到遇到空格或换行等：

```
while ((k = scanf("%d", &n)) != EOF) {
    if (k == 0)
        while (!isspace(getchar()))
            ;
```

```
        ... ...
    }
```

这一具体做法是丢掉了输入中的一段，是否合适要看程序的具体情况。

从上面这些讨论可以看出，写好输入不是一件简单的事情。这里的原因也很明显，输入操作描述程序与外部打交道的动作，以便程序根据外部提供的信息决定内部的工作方式。而在程序执行时，外部的情况完全不受写程序的人控制。所以，要写好处理输入的程序片段，实际上需要考虑各种可能外部情况并适当地处理（这里很难说“正确”，只能说“适当”）。在外部提供的实际输入不满足程序要求时，应该设法给外部提供一些信息。

8.3.3 输出格式控制

我们也把 printf 看作格式化输出函数的代表，本节将以它为例，讨论标准库格式化输出函数的格式描述问题。其他格式化输出函数（如 fprintf 等）在格式描述方面的情况与 printf 完全相同。printf 的原型是：

```
int printf(const char *format, ...)
```

它也是一个参数个数可变化的函数。前面说了，对格式描述串的后面用...表示的参数，编译程序不做任何检查，如果所提供的实参类型与格式描述串中所要求的类型转换不符，就无法保证经过输出转换得到是的所需要的结果。初学 C 程序设计的人常遇到这种情况：程序的输出结果不对，但怎么检查也找不出来程序中的错误。最后才发现是输出语句的转换描述和输出表达式的类型不匹配。

printf 根据格式描述串 format 完成输出转换，把生成的输出字符序列送到标准输出流。操作中出错时函数返回负值；没出错时返回本调用执行中输出的总字符数。

format 应是一个字符串。按函数 printf 的看法，该串的内容分为两类：一类是普通字符，printf 遇到普通字符时将它们直接送到输出流。另一类是由%开头的转换描述，这种描述由连续的若干个字符组成，它们并不输出，而是作为处理函数实参的指示。printf 根据格式串中的转换描述顺序处理函数的其他实参。处理一个参数得到的字符序列称为一个输出字段，各输出字段插入对应转换描述在格式串里的位置，形成整个输出序列。

格式串里的转换描述总以%开始，到一个转换字符为止（所有转换字符见下面的表），两者之间顺序地可以有下面几种成分（几个字符，也可以没有）：

1. 标志字符。下面几个字符可以按任意顺序出现，可以有一个或者多个：

-	将转换结果在字段范围内由最左端开始输出（居左输出）
+	在数值的前面总输出一个正号或负号
空格	如果转换后产生的第一个字符不是正负号，就首先输出一个空格
0	用于数值输出。如果输出不能填满整个字段，那么在有效输出之前填满 0
#	指定另一种规定形式。对转换字符 o，数值之前总加 0；对转换字符 x 和 X，非 0 结果之前总加 0x 或 0X；对于转换字符 e、E、f、g、G，输出中总包含小数点；对于 g 和 G，不去掉最后的那些 0

2. 一个十进制整数。表示本输出字段的最小宽度，要求转换结果至少占这么多个字符的宽度，如果需要可以更宽。如果得到的输出序列不够宽，在其左边（或者右边，如果要求左对齐的话）填满空格。对于数值输出，当有 0 标志时在数字序列的左边填满 0。
3. 一个圆点及另一个表示精度的十进制整数。对于字符串参数，这个数表示应输出的最大字符个数；对 e、E、f 表示小数点之后的数字位数；对 g、G 转换表示有效数字位数；对于整数表示要求输出的最小数字个数，如果数字个数不够就在左边添 0。
4. 目标长度修饰字符 h、l 或者 L。字符 h 和 l 用于整型参数，h 说明相应的参数是 short 或者 unsigned short 类型；l 说明对应参数是 long 或者 unsigned long 类型。

字符 L 用于说明对应参数为长双精度类型。对于上述这些情况，转换描述中都必须用长度修饰字符指明对应参数的表示长度（类型特征）。

字段宽度和精度都可以只写一个星号，表示实际所用的值由 printf 的参数取得，提供值的那一个或两个参数都必须是 int 类型的。

下表给出了对各转换字符的详细说明，包括它们所要求的参数类型和实际输出形式。如果将其他字符写在%的后面，程序的行为没有定义。

转换字符	实际输出形式	要求的参数
d,i	带符号的十进制形式整数	int
o	无符号八进制表示的整数，没有先导的 0	int
x,X	无符号十六进制整数，没有先导的 0x 或 0X。在用转换字符 x 时，十以上数字用 abcdef 表示；用 X 时这些数字用 ABCDEF 表示	int
u	无符号十进制整数	int
c	输出一个字符，将参数转换为 unsigned char 输出	int
s	输出一个字符序列，从参数所指位置开始直到遇到字符'\0'，或者达到字段的指定宽度为止	char*
f	一般实数形式，形式为[-]mmm.ddd，其中小数点后面数字位数由精度描述确定，默认值是 6。精度为 0 时不输出小数点	double
e,E	科学计数形式，形式为[-]m.ddde±xx 或[-]m.dddE±xx，其中小数点后面的位数由精度描述确定，默认为 6 位。精度为 0 时不输出小数点	double
g,G	灵活形式。当指数小于-4 或大于等于精度描述时用%e 或%E 的形式输出，否则用%f 的形式输出。末尾的 0 或小数点不输出	double
p	输出指针的值，采用某种由具体实现确定的形式	void*
n	把这次函数执行到这里已经输出的字符个数写到参数中。处理这个“转换描述”时不产生输出	int*
%	输出字符%，不做任何转换	--

前面没有介绍的主要问题是输出的字段宽度控制和数值的输出精度控制。此外还有左右对齐问题等等。只所以没有早讨论这些问题，是因为这些东西很琐碎，而且（与其他重要问题相比）它们只是不太重要的细节，在初学程序设计时不必过分在意。当然，就实际应用系统而言，输出格式也是程序质量的一部分，但对程序练习而言就是不重要的细节了。

通过在输出语句的格式串里使用各种转换描述形式，可以形成所需要的各种输出形式。下面是几个转换描述的实例，请读者根据上表弄清楚它们的意义：

```
%16.8lf    %-10.6f    %20.12e    %010ld    %.7s
%16.8g    %#10o      %+012d    %+#f      %
```

此外，在转换描述里，可以在表示字段的宽度和精度的位置写星号，这时实际的字段宽度和精度将由对应的参数得到。这种机制使人可以比较容易地在程序里控制输出格式。如果输出函数的格式描述串里用到了包含星号的转换描述，函数 printf 执行遇到时这个转换描述就要用掉两个或三个实际参数。例如下面调用：

```
printf("%s %*.*f\n", "Result:", len, prec, val);
printf("%s %*d\n", "Number:", width, num);
```

这里的第一个输出语句首先输出字符串“Result:”，然后按最小字段宽度 len 和精度 prec 输出变量 val 的值（假定其中 val 是一个双精度变量，len 和 prec 是两个整型变量，它们都已经有了合适的值）。第二个输出语句输出 num 的值（假定 num 是个有定义的整型变量），这个整数输出的字段宽度由（整型变量）width 的值确定。

8.3.4 以字符串作为格式化输入输出对象

文件可以看作是字节的序列，通过文本流打开的文件可以看着是一个字符序列。按照这种观点，对文本流做格式化输入，也就是从这种字符序列的前部取出一段字符，按某种方式转换后，把结果存入指定变量。输出过程正好与此相反，是根据需要将程序内部的数据转换为字符序列，并将它馈入相应的字符流（或者字节流）中。

还有一类对象也是字符的序列，这就是字符串（无论是字符串常量，还是存储在字符数组里的字符串）。根据上面的讨论，似乎也可以针对字符串做与文件输入输出类似的工作。把字符串（字符数组）作为输入或输出对象也很自然。将文件流看成字符序列，其结束就是相应文件的结束。与文件流不同的是字符串存储在内存，其结束用空字符表示。

C 语言标准库确实提供了两个以字符串为对象的格式化输入输出函数。其中输入函数 sscanf 的功能是把一个字符串作为读入对象，从中读入、分解、完成指定转换，并将转换结果赋给指定变量。字符串输出函数 sprintf 实现另一方向的转换，将生成的输出字符序列存入指定的字符数组，并在有效字符序列最后放入表示字符串结束的'\0'，做成字符串的形式。这两个函数的原型与 scanf 和 printf 类似，只是多了一个字符指针参数，用于表示特定的字符串：

```
int sscanf(char *s, const char *format, ...);
int sprintf(char *s, const char *format, ...);
```

对于 sscanf，字符指针参数 s 表示作为输入对象的字符串；对于函数 sprintf，参数 s 应该指定一个足够大的字符数组。这两个函数的功能与文件格式化输入输出函数完全相同，只是所作用的对象不是文件流而是字符串。

首先将文件内容（包括来自标准流的输入）读入字符数组，而后用 sscanf 在字符数组里完成“分析”和格式化，使我们能比较方便地处理一些复杂输入问题。在本章后面有关程序实例的节中将给出这方面的实例。

8.3.5 标准错误流

考虑前面的程序实例 cat，该程序执行中完成一系列文件复制。如果程序执行中遇到有文件打不开，它就会将产生出一行错误信息。在这里让程序输出错误信息，是希望这种信息能显示在计算机屏幕上，使程序的用户可以读到它并采取相应处理措施。看起来这种做法没有什么问题，但在使用中却可能出现麻烦。如果用户在使用这个程序时把输出重定向到某个文件，希望把几个文件的内容都复制到一个文件里，问题就出现了：程序产生的出错信息也会按所给的定向送入指定文件。这种结果显然不是我们希望的。

利用标准库提供的标准错误流（stderr）可以解决这个问题。送到标准错误流的信息将不受输出流重新定向的影响，因此不会混入定向的文件之中。即使标准输出流被重新定向，送到 stderr 的信息仍会显示在计算机屏幕上。

要改造程序 cat，只需修改关于错误信息输出的一个语句，把它改为：

```
fprintf(stderr, "%s, can't open in file: %s\n", name, *argv);
```

8.4 程序实例

本节给出几个稍微大一些的程序实例，介绍有关输入输出程序设计的一些情况，这些也是对前面几节所讨论的标准库功能的应用。

8.4.1 求文件数据的平均值

假设我们有许多文件，每个文件里存着一组实数值。现在要求写一个程序，它不断向用

户要求文件名，并对每个文件里的数值求出平均值输出。

首先考虑程序的功能分解。从一个文件里读取数值，求出它们的平均值并输出是一项独立工作，可以考虑定义一个函数，其函数原型可以设计为：

```
void pAverage(FILE *fp, char *fname)
```

所要求的参数是已经打开的输入文件指针和相应文件名，并不考虑文件的由来及打开文件的工作。根据上面提出的问题和已知的方法，pAverage 的基本定义可写为：

```
void average(FILE *fp, char *fname) {
    double x, sum = 0.0;
    int l = 0, n = 0, m, c;
    printf("\nFile %s:\n", fname);
    while ((m = fscanf(fp, "%lf", &x)) != EOF) {
        ++l;
        if (m == 1) {
            sum += x;
            ++n;
            continue; /* 继续循环读入 */
        }
        fprintf(stderr, "Date error. Line %d: ", l);
        while (!isspace(c = getc(fp)))
            putchar(c);
        putchar('\n');
    }
    printf("Average: %16.8f\n", sum / n);
}
```

如果在读入中遇到数据错误，这个函数将输出出错行的行号，并输出直至下一空白字符的那几个字符，以便用户检查。处理完毕后输出的是文件中正确数据的平均值。

主函数基本功能的定义并不困难，就是一个不断向用户要求文件名并调用 pAverage 的基本循环。但这里需要确定一种终止循环的方法。由于循环中要从用户得到一系列文件名（字符串），如果以某个字符作为结束信号，文件名字里就不能包含这个字符。可能的方法是用一个不能作为文件名的字符，或者利用文件结束控制循环。下面定义里用文件结束信息终止循环，当程序要求被处理文件名时，给它一个文件结束信号，程序就结束了：

```
int main(void) {
    char name[256];
    FILE *fp;
    while (1) {
        printf("File name (end-of-file to end): ");
        if (fscanf("%255s", name) == EOF) break;
        if ((fp = fopen(name, "r")) == NULL)
            fprintf(stderr, "Can't open file: %s\n", name);
        else {
            average(fp, name);
            fclose(fp);
        }
    }
    printf("Bye!\n");
    return 0;
}
```

把这两个部分写成一个程序，就可以完成题目所要求的工作了。读者从这里可以看到文件处理过程中的一些一般性的情况。

本程序可以有另一个版本，它要求用户通过命令行参数提供文件名。下面是修改后的主函数。启动程序时提供表示文件名的命令行参数，程序就处理这些文件；否则什么也不做：

```
int main(int argc, char** argv) {
    FILE *fp;
    while (*++argv != NULL) {
```

```
            if ((fp = fopen(*argv, "r")) == NULL)
                printf("Can't open file: %s\n", *argv);
            else {
                average(fp, *argv);
                fclose(fp);
            }
        }
        printf("Bye!\n");
        return 0;
    }
```

因为命令名是首个命令行参数，第一次循环时应过它，并试着去打开第一个数据文件。随后的每次循环打开并处理一个文件，直至用完所有命令行参数。注意，在表示命令行参数的字符指针数组最后有一个空指针，这利用该指针控制循环的结束。

8.4.2 一个背单词程序

现在考虑做一个背英语单词的程序，其基本使用循环是：显示一个中文词，要求用户输入对应的英文单词并给以评判。显然，在这个程序运行时，程序里应该存储着一对一对的中文和英文词。用户不会希望自己每次使用时输入这些词，最好是将单词存放在文件里，每次启动程序后装入其中。这样，本程序就需要从文件输入。

为了使程序的输入工作比较规范，我们要首先设计好单词文件内的数据格式。一种简单设计假定每行里放一对英文词和中文词，例如英文在前而中文在后，用空格分隔。这是一种简单的文件格式设计，下面假定程序输入的是这种文件。

我们还可以假定单词文件不止一个，用户在与程序交互中提供单词文件名，要求程序装入文件内容，而后在交互中做练习。假定将所有中文和英文词用字符串形式保存在一个两维字符数组里。有了这些设计之后，我们就可以写出相关的数据定义和主函数了：

```
enum {
    WDNUM = 1000, /* 中英文词数 */
    WDLEN = 32,   /* 单词存储数组长度 */
    ROUND = 20    /* 一轮练习的次数 */
};

char wds[WDNUM*2][WDLEN];

#define ENGLISH(i) wds[2*(i)]
#define CHINESE(i) wds[2*(i)+1]

int main() {
    char fn[256];
    FILE* fp;
    int terms;

    do {
        getnstr("Word file name: ", 256, fn);
        if ((fp = fopen(fn, "r")) == NULL)
            printf("Wrong file name. ");
        else {
            terms = readfile(fp, WDNUM, wds);
            fclose(fp);
            if (terms == 0) continue;
            wordgame(terms, wds, ROUND);
        }
    } while (next("word file"));

    return 0;
}
```

这里用一个全局数组 wds 存放程序运行中使用的单词对。计划采用的存放方式是顺序安放

英文单词和对应中文词，这里定义了两个宏，以使单词访问描述更清晰。main 的局部变量 fn 存放用户提供的文件名，fp 是文件指针，terms 记录读入的单词对的数目。

main 里只有一个循环：首先通过通用函数 getnstr 向用户要求文件名。如果文件能正常打开，就用 readfile 从文件里读入单词。只要文件中有单词就进入下面 wordgame 函数，开始交互式练习。当用户不希望继续用本文件练习时 wordgame 结束，next 询问是读入其他文件继续做练习，还是立即结束。

getnstr 的定义很简单，它读入一段字符，遇到空白字符就结束。

```
void getnstr (char prompt[], int lim, char bf[]) {
    int c, i = 0;
    printf("%s", prompt);
    while (i < lim-1 && (c = getchar()) != EOF && !isspace(c))
        bf[i++] = c;
    if (c != '\n')
        while (getchar() != '\n') ; /* 吃掉本行剩余字符 */
    bf[i] = '\0';
}
```

这个定义的特殊之处在于认定一个输入行里的有用信息就是一个字符段（由空白界定），如果后面有其他信息就都应该抛弃。读入字符序列存入参数数组 bf，最后放好空字符，做成字符串形式。注意，读入数组的动作都必须检查越界问题。

这里的 next 是前面章节里简单函数的推广，它提供了一个提示字符串参数，通过读入的 y 或者 n 确定返回真假值。本函数只在遇到 y 时返回 1，其余情况都返回 0：

```
int next(char s[]) {
    int c;
    printf("Next %s? (y/n): ", s);
    while (isspace(c = getchar())) /* 读到下一个非空白字符 */
        ;
    while (getchar() != '\n') ; /* 吃掉本行剩余字符 */
    if (c == 'y' || c == 'Y') return 1;
    else return 0;
}
```

这里还特别注意了 y 的大小写问题。这个函数可能在各种程序里使用。

下面先考虑函数 wordgame，因为它也比较简单。这一函数里应该有一个循环，其中反复选择并显示中文单词，要求用户输入对应的英文单词。最简单选词策略的是顺序显示单词。但是这种做法太单调，也缺乏变化。一种稍有些变化的方法是采用随机选词的策略。当然还可以有更复杂的选词策略，在这里不准备考虑。下面定义采用随机选择的策略：

```
void wordgame(int terms, char wds[][WDLEN], int rd) {
    int n, i;
    char wd[WDLEN];
    do {
        for (i = 0; i < rd; ++i) {
            n = rand()%terms;
            printf("%s ", CHINESE(n));
            getnstr("> ", wd, WDLEN);
            if (strcmp(wd, ENGLISH(n)) == 0)
                printf("Ok!\n");
            else
                printf("Wrong! It is: %s\n", ENGLISH(n));
        }
    } while (next("Round?"));
}
```

函数的内层循环体以执行 rd 次为一个周期，而后询问用户是否继续。在循环体的每次执行

中选择一个下标，显示单词，读取用户输入并完成评判。

剩下的问题就是文件读入函数 readfile 了，其原型是：

```
int readfile(FILE *fp, int lim, char wds[][WDLEN]);
```

根据已确定的文件格式，我们可以让这个函数采用按行输入的方式。这样做的优点有几方面：1）将一行读入内存而后分析，较容易适应各种复杂输入的要求。2）遇到格式不符合要求的行时更容易提供信息。在这种考虑之下可以得到如下的实现：

```
int readfile(FILE *fp, int lim, char wds[][WDLEN]) {
    char line[256], *p;
    int ln = 0, n = 0;

    while (n < lim && fgets(line, 256, fp) != NULL) {
        ++ln;
        p = line;
        p = charscopy(WDLEN-1, ENGLISH(n), p, ' ');
        p = charscopy(WDLEN-1, CHINESE(n), p, ' ');
        if (*ENGLISH(n) == '\0' || *CHINESE(n) == '\0')
            printf("Wrong line #%d: %s", ln, line);
        else ++n;
    }

    return n;
}
```

这个函数两次调用另一个辅助函数 charscopy，把由 p 所指位置开始的不超过 WDLEN-1 个字符复制到单词数组的一个元素里。这里的 ' ' 表示以空格作为单词的分隔符号。如果需要也可以采用其他分隔符号，这种设计使函数的功能更广泛一点。这个定义实际上假定了每个输入行不超过 255 个实际字符。

最后是 charscopy 的定义，也很简单：

```
char* charscopy(int lim, char t[], char s[], char delim) {
    int i;
    while (isspace(*s)) ++s;
    for (i = 0; i<lim && *s != '\0' && *s != '\n' && *s != delim;
         ++i, ++s, ++t)
        *t = *s;
    *t = '\0';

    return s + 1;
}
```

循环完成简单的字符序列复制。

我们也可以用下面程序段完成程序的输入，这里用了前面介绍的从字符串做格式输入的标准库函数。使用转换描述 %s，就表示用空白字符作为单词的分隔字符：

```
char line[256];
int n = 0;

while (n < lim && fgets(line, 256, fp) != NULL) {
    sscanf(line, "%s %s", ENGLISH(n), CHINASE(n));
    if (*(ENGLISH(n)) == '\0' || *(ENGLISH(n)) == '\0') {
        printf("Wrong line #%d: %s", n+1, line);
        continue;
    }
    ++n;
}
```

这一程序段里首先将文件中的一行读入字符数组 line，而后将这个字符串作为格式化输入的对象，将串中两个由空白分隔的字符段存入指定位置。

至此我们完成了一个大约 100 行的程序，它真的能做一点事情。将所有函数定义按适当顺序写在一个源文件里，加上适当的头文件包含命令，就形成了一个完整的程序。这个程序

由 6 个不长的函数定义构成，每个函数完成一方面工作，结构很清晰。由于已经将复杂的程序功能分解为逻辑性很强的函数，整个程序也不难理解，不难修改它去满足新的需要。

当然，作为实用程序，上面程序还有许多可以改进的地方。本章后面习题提出了一些问题，后面章节里还会考虑这些问题。

8.4.3 资金账目系统

假设现在要开发一个管理资金来往账目的系统，其中需要记录每笔收支的日期、项目简记和相应金额。其中用负数表示支出，正数表示收入。假设程序的输入按行进行，每行输入一笔账目。这里不想考虑整个系统，只准备研究程序的数据输入问题。

为了在程序里记录来往账目，可以考虑用一个 n×3 的整数数组记录各笔收支发生的日期，用一个二维字符数组表示项目简记，另用一个浮点数组记录帐目金额。通过数组下标建立几个数组中项目之间的联系，例如，日期数组第 k 个子数组表示的是简记数组第 k 项和金额数组第 k 项收支的日期。可以看出，这种设计有缺点，因为相互有关的数据（例如，第 k 项记录的所有信息）分别存放在三个不同数组里。这是因为我们目前的数据组织手段只有数组，而作为数组基本元素的只能是各种基本类型的数据。学习了下一章的内容之后，我们就能有更好的方式组织程序里的这类数据了。

现在定义如下变量存储程序里使用的各种数据：

```
enum { DATANUM = 400,
       DESCLEN 32 };
int dates[DATANUM][3];
int descrbs[DATANUM][DESCLEN];
double currs[DATANUM];
```

假定数据输入通过输入流（文件指针）fp 进行，下面语句将完成一项收支的输入：

```
fscanf(fp, "%d%d%d %32s%lf", &dates[i][0], &dates[i][1], &dates[i][2],
                             descrbs[i], &currs[i]);
```

因为 descrbs[i] 就是 descrbs 的第 i 个子数组，输入的字符串字段应该存入其中。这里用 %32s 表示输入不超过 31 个字符的一个输入字段，并将其以字符串形式存入 descrbs 数组的指定子数组里。这实际上也要求相应字段确实不超长。

如果文件指针 fp 关联于一个文件，文件里有如下形式的行：

```
1998 3 24 project-funding 30008.44
1998 3 26 buy-hard-disk -2437.50
... ...
```

通过反复执行上述输入语句，就可以将这些数据读入到数组里。

如果文件中某行的内容是：

```
1998 4 30 buy main board -1430.20
```

程序读到这一行时就会出问题。函数开始对日期的输入转换能正确完成，但在读入收支项目描述串时，将会把字符串 "buy" 赋进字符数组 descrbs[i]，将后面的部分留下来。随后的转换描述要求读表示浮点数的字段，遇到的却是：

```
main board -1430.20
```

因此转换失败。不但 currs[i] 没赋值（以后使用它可能出现错误），留下的半行字符还会导致随后的输入失败。如果读入文件内容的循环没有精心考虑，写成了：

```
while (i < DATANUM &&
       fscanf(fp, "%d%d%d %32s%lf", &dates[i][0], &dates[i][1],
              &dates[i][2], descrbs[i], &currs[i]) != EOF))
    ++i;
```

遇到上面的行，虽然由于有 i 的控制不会导致程序陷入无穷循环，但随后的输入都将失败，导致后面数组元素都没有正确赋值，给后续程序执行埋藏下隐患。做了些改造后的循环：

```
while (i < DATANUM &&
       fscanf(fp, "%d%d%d %32s%lf", &dates[i][0], &dates[i][1],
              &dates[i][2], descrbs[i], &currs[i]) == 5))
    ++i;
```

也有问题：一旦遇到上面这类不符合要求的行，输入循环不提供任何信息就结束了，而用户根本不知道输入中遇到了错误。在实际应用程序里，这种情况也很危险，因为它给用户造成一切正常的假相，而实际计算结果却可能完全没价值。

一般而言，这种情况下已出现不完整的输入数据。想恢复出问题的数据将非常困难，通常完全不太可能了，因为写程序时无法确定所有的可能情况。遇到这种问题时，一般说我们只能退而求其次，考虑丢掉不完整的剩余数据，设法使输入过程回到某种可以把握的状态，从那里继续进行下去。下面是一种可行处理方案：

```
while (i < DATANUM &&
       (k = fscanf(fp, "%d%d%d %32s%lf", &dates[i][0],&dates[i][1],
                   &dates[i][2], descrbs[i], &currs[i])) != EOF)
{
    if (k != 5) {
        fprintf(stderr, ... ...); /* 输出错误信息，最好能为纠错提供合理线索 */
        while (getchar() != '\n') ; /* 常用策略，丢掉直至本行剩余字符 */
    }
    else
        ++i;
}
```

这一写法实际假定了一项收支记录结束之后总有换行符号（包括最后一行。如果不能保证，就应在丢字符小循环的条件中加上对文件结束的检查）。上述写法一定能继续到读完整个文件，因为循环体每执行一次，一定能消耗掉输入流里的一些字符。

正如前面反复强调的，我们无法保证外界用户总能满足程序需求，能正确处理与程序的交互问题，从不出错。从写程序的角度说，也没有万能的方案去合理处理外界所有可能情况，常常必须做出一些假定。在设计实现程序的输入部分时，一是要设法保护程序，使之能够抵御错误输入的侵害；二是在必要时应设法将数据错误的情况通知用户。

现在考虑一个新问题。假定我们的输入文件有多种不同来源，而不同来源的文件里日期的写法有几种不同形式。包括：

```
1998 10 17      1998,10,17      17/10/1998
```

怎样写程序的输入部分，才能保证这三种不同形式的输入都能得到正确处理呢？

直接通过 scanf 读入将很难处理。因为如果程序首先假定了某种形式，并按照该形式读入，在处理了部分信息之后就可能失败。此时再考虑弥补措施已经很困难了。对这种问题的更好处理方法是：先用行式输入函数把一个完整输入行读进一个数组，然后用字符串格式化输入函数进行分析。这样，如果按一种方式的分析失败，还可以方便地试验另一种方式。当然，这里我们实际上也假定，一项收支的完整信息在文件里存为一行。

要实现这种输入方式，程序里需要使用一个字符数组。可以假定一行数据的表示不超过 256 个字符。这样就可以定义一个字符数组 line 存放输入行的字符：

```
char line[256];
```

实现上面所要求的处理过程的程序片段是：

```
while (i < DATANUM && fgets(line, 256, fp) != NULL) {
   if ((sscanf(line, "%d %d %d %32s %lf", &dates[i][0], &dates[i][1],
               &dates[i][2], descrbs[i], &currs[i])) == 5) ||
       (sscanf(line, "%d , %d , %d %32s %lf", &dates[i][0],&dates[i][1],
               &dates[i][2], descrbs[i], &currs[i])) == 5) ||
       (sscanf(line, "%d / %d / %d %32s %lf", &dates[i][2],&dates[i][1],
               &dates[i][0], descrbs[i], &currs[i])) == 5)) {
        ++i;
   }
   else
       fprintf(stderr, ... ...); /* 输出错误信息，最好能为纠错提供合理线索 */
}
```

这段程序能将符合各种格式要求的日期信息正确读入数组中。请注意程序段里 if 的书写方式：这里用了三个函数调用的或作为控制条件。如果其中某个条件成立（成功转换了 5 项），

不但这个函数调用完成了所有赋值工作，它也将导致其后面的函数调用不会再执行，并因条件为真而使 i 加一，循环继续。当然，如果读者不习惯这种情况，也可以将函数调用分开写成几个语句，采用一连串 if 语句描述。

采用这种首先用行式输入，而后在内存里分析的方法，我们甚至还可以进一步扩充程序的功能，写出能够处理出现分段的项目简记的输入行，例如：

```
1998 4 30 buy main board -1430.20
```

相应工作留给读者自己完成。当然，做这类扩充时都需要做出一些假定，不可能写出万能的输入程序段。另一方面，扩充允许的输入形式也会使程序的输入处理变得更加复杂。因此，我们应该仔细分析实际需求，做出恰当的设计选择。

标准头文件<stdio.h>还提供了另外一些功能，主要对输入和输出的各种控制。那些方面有不少细节，服务于一些特别的需要，其中有些问题的讨论还依赖于另一些知识。因此我们将有关功能留在最后介绍标准库功能的一章里。

练习

1. 写一个程序打印乘法九九表。利用格式控制保证表的的项能很好对齐。
2. 写一个程序，其命令行要求有三个参数。该程序把这些参数看成文件名，完成的工作是把前两个文件的内容连接起来，存放到第三个文件里（文件连接）。
3. 将前面一些程序改写为采用文件输入输出的实现的，通过命令行或者输入语句为程序提供运行的数据或输入输出的文件名。
4. 修改前一章有关猜单词游戏的练习，让程序从文件中读入一批单词，文件的名字从命令行得到。如果调用函数时未提供文件名，则向用户要求文件名。
5. 修改前面的学生成绩统计和直方图生成程序。假定成绩文件中的每一行是一个学生的姓名（例如，连续字符序列表示的汉语拼音姓名）和相应成绩。程序首先输入文件，而后先输出成绩不及格的学生姓名和成绩，再输出成绩及格的学生姓名和成绩。
6. 修改学生成绩统计和直方图生成程序。假定文件里的每行记录了一个学生的姓名和几门课程的成绩，程序读入这种文件，计算出每个同学的平均成绩，而后：首先输出平均成绩不及格的同学名单和对应成绩；而后输出及格的同学名单和对应成绩。
7. 假设电价分段计费，不同时段每度电的单价不同。每个用户的用电记录在文件里存为一段，其中的第一行是用户名，随后的每个记录项包括两个浮点数据，第一个数据是一段时间的用电度数，第二个数据是每度的单价，都由空格或者换行分隔。请写出一个完整的程序，它不断要求人提供输入文件的名字，对每个文件算出各个用户应付的电费并输出结果。程序结束时输出所有用户应付的总电费额。这个程序在启动后应能处理任意多个文件，并使用户能在处理完希望处理的文件之后控制程序结束。
8. 写一个程序，其命令行有两个文件名参数，在第一个文件里是一个单词表，第二个文件是被处理的文件。程序生成一个新文件，其中包含单词表中的每个单词，并附以各单词在被处理文件里出现位置的行号（可能有多个行号，但应当避免重复）。
9. 文中的背单词程序用两维数组记录单词。为了防止单词超出范围，那里定义了一个很大的常量 WDLEN，而实际上许多单词和对应正文词所需的存储都远远小于这个量。请修改这一程序，用一个指针 char* 的数组代替那里的两维数组。在读入英文单词和中文词时用较大的 char 数组作为临时存储，而后通过动态分配的方法为每个英文或正文单词字符串建立适当大小的数组，并将字符串复制进去。为了这种改动，程序的其他部分还可以保持原样吗？