第 12 章 标准输入输出库

程序如果不能接受你的指令并返回处理的结果就没有多少用处。因此几乎所有的程序都会做些输入输出。C语言的输入输出是通过库函数实现的——这些函数在标准输入输出（或称“stdio”）库[①]中——这些库函数因此也是C语言库中最常用的一部分。

基于C语言的最小化原则，标准库采纳了一种简单、直接的输入输出模型，可以打开、读取、写入文件。文件被当作有序字节流进行处理，当然也可以定位。在有意义的情况下，也可以区分文本和二进制文件。使用字符串来代表文件的任意名称或路径名，然后由底层的操作系统来对它进行解释。除了在路径名称之外，没有目录的概念，也没有什么标准的方法来创建目录或获取目录内容（参见第19章）。程序隐含打开3个标准输入输出流。可以从stdin中读取，通常这是一个交互键盘，可以向stdout或stderr写入，这两个通常都是用户的显示屏。但是，几乎没有什么预定义的功能可以处理键盘和屏幕的具体细节（参见第19章）。

本章的很多问题都跟printf（问题12.7到12.12）和scanf（问题12.13到12.22）有关。问题12.23到12.34涉及了stdio库中的其他函数。当需要访问一个具体文件时，可以用fopen（问题12.29到12.34）打开它，也可以把一个标准流重定向到它（问题12.35到12.38）。如果对文本输入输出不满意，还可以求助于“二进制”流（问题12.40到12.45）。当然，在深入这些具体的细节之前还有一些简单的、介绍性的输入输出问题。

基本输入输出

12.1

问：这样的代码有什么问题？

```
char c;
while((c = getchar()) != EOF) ...
```

答：首先，保存getchar的返回值的变量必须是int型。EOF是getchar返回的“超出范围”的

① 当我们谈到“stdio”库的时候，我们的实际意思是“标准C运行库中的stdio函数”或者说“<stdio.h>描述的函数”。

特殊值，它跟getchar可能返回的其他任何字符值都不一样。（在时新的系统上，文件中已经不再保存真正的文件结束符了，EOF只不过是一个没有更多字符的信号而已。）getchar返回的值必须保存在一个比char型大的变量中，这样才能保存所有的char值和EOF。

像前面的代码片段那样将getchar的返回值赋给char可能产生两种失败情况。

(1) 如果char型有符号而EOF（像通常那样）定义为-1，则十进制值为255的字符（'\377'或'\xff'）会被符号扩展，跟EOF比较的时候会相等，从而过早地结束输入。[①]

(2) 如果char型无符号，则EOF会被截断（扔掉最高位，可能变成255或0xff）而不再被识别为EOF，从而导致无休止的输入[②]。

然而，如果char型有符号而输入的又都是7位的字符，则这个错误可能持续很长时间而不被发现。（普通char型是否有符号由实现定义。）

参考资料：[18, Sec. 1.5 p. 14]
[19, Sec. 1.5.1 p. 16]
[35, Sec. 3.1.2.5, Sec. 4.9.1, Sec. 4.9.7.5]
[8, Sec. 6.1.2.5, Sec. 7.9.1, Sec. 7.9.7.5]
[11, Sec. 5.1.3 p. 116, Sec. 15.1, Sec. 15.6]
[22, Sec. 5.1 p. 70]
[12, Sec. 11 p. 157]

12.2

问：我有个读取直到EOF的简单程序，但是我如何才能在键盘上输入那个“\EOF”呢？我看<stdio.h>中定义的EOF是-1，是不是说我该输入-1？

答：考虑一下就知道，你输入的绝不能是-1，因为-1是两个字符，而getchar每次读入一个字符。事实上，在你的C程序中看到的EOF值和你在键盘上发出文件结束符的按键组合之间并没有什么关系。EOF不过是向程序发出的一个信号，指明输入不再有任何字符了，不论什么原因（磁盘文件结束、用户结束输入、网络流关闭和I/O错误等。）根据你的操作系统，你可能使用不同的按键组合来表示文件结束，通常是Ctrl-D或Ctrl-Z。操作系统和标准输入输出库安排你的程序接收EOF值。（然而请注意，这一路有好几个转换。通常情况下，你不能自己检查Ctrl-D或Ctrl-Z值，你在stdio.h文件中也不会发现EOF宏定义成了这样的值。）

12.3

问：为什么这些代码把最后一行复制了两遍？

① 值255假设char为8位。某些系统上char型可能更大，但类似的失败情况不可避免。
② 跟前一段一样，值255假设char为8位。某些系统上char型更大，但类似的失败情况不可避免。

```
while(!feof(infp)){
    fgets(buf, MAXLINE, infp);
    fputs(buf, outfp);
}
```

答：在C语言中，只有输入例程试图读取并失败以后才能得到EOF。（换言之，C的I/O和Pascal的不一样。）通常只需要检查输入例程的返回值：

```
while(fgets(buf, MAXLINE, infp) != NULL)
    fputs(buf, outfp);
```

一般说来，完全没有必要使用feof。（偶尔可以用feof或ferror在stdio调用返回EOF或NULL之后判断是文件结束条件还是读取错误。）

参考资料：[19, Sec. 7.6 p. 164]
[35, Sec. 4.9.3, Sec. 4.9.7.1, Sec. 4.9.10.2]
[8, Sec. 7.9.3, Sec. 7.9.7.1, Sec. 7.9.10.2]
[11, Sec. 15.14 p. 382]

12.4

问：我用fgets将文件的每行内容读入指针数组。为什么结果所有的行都是最后一行的内容呢？

答：参见问题7.6。

12.5

问：我的程序的屏幕提示和中间输出有时没有在屏幕上显示，尤其是当我用管道通过另一个程序输出的时候。为什么？

答：在输出需要显示的时候最好使用显式的fflush(stdout)调用。[①]有几种机制会努力帮助你在“适当的时机”执行fflush，但这仅限于stdout为交互终端的时候。参见问题12.26。

参考资料：[35, Sec. 4.9.5.2]
[8, Sec. 7.9.5.2]

12.6

① 另一种方法是用setbuf和setvbuf关掉输出流的缓冲。但缓冲是个好东西，完全关掉它可能引发极度的低效率。

问：我怎样才能不等待回车键而一次输入一个字符？

答：参见问题19.1。

printf格式

12.7

问：如何在printf的格式串中输出一个'%'字符？我试过\%，但是不行。

答：只需要重复百分号：%%。

用printf输出%之所以困难是因为%正是printf的转义字符。任何时候printf遇到%，它都会等待下一个字符，然后决定如何处理。而双字符序列%%就被定义成了单独的%字符。

要理解为什么\%不行，得知道反斜杠\是编译器的转义字符，它控制编译器在编译时对源代码中字符的解释。而这里我们的问题是printf如何在运行时控制它的格式串。在编译器看来，\%可能没有定义或者代表一个%字符。就算printf会对\特殊处理，\和%在printf中都有效的可能性也不大。

参见问题8.8和19.20。

参考资料：[18, Sec. 7.3 p. 147]
[19, Sec. 7.2 p. 154]
[35, Sec. 4.9.6.1]
[8, Sec. 7.9.6.1]

12.8

问：为什么这么写不对？

```
long int n = 123456;
printf("%d\n", n);
```

答：任何时候用printf输出long int型都必须在printf的格式串中使用l（小写的“ell”）修饰符（例如%ld）。因为printf不知道传入的数据类型，必须通过使用正确的格式说明符让它知道。

12.9

问：有人告诉我不能在printf中使用%lf。为什么printf()用%f输出double型，而scanf却用%lf呢？

答：printf的%f说明符的确既可以输出float型又可以输出double型。[①]根据“默认参数提升”规则（在printf这样的函数的可变参数列表中[②]，不论作用域内有没有原型，都适用这一规则）float型会被提升为double型。因此printf()只会看到双精度数。参见问题15.2。

对于scanf，情况就完全不同了，它接受指针，这里没有类似的类型提升。（通过指针）向float存储和向double存储大不一样，因此，scanf区别%f和%lf。

下表列出了printf和scanf对于各种格式说明符可以接受的参数类型。

格　式	printf	scanf
%c	int	char *
%d, %i	int	int *
%o, %u, %x	unsigned int	unsigned int *

（续）

格　式	printf	scanf
%ld, %li	long int	long int *
%lo, %lu, %lx	unsinged long int	unsigned long int *
%hd, %hi	int	short int *
%ho, %hu, %hx	unsigned int	unsigned short int *
%e, %f, %g	double	float *
%le, %lf, %lg	n/a	double *
%s	char *	char *
%[...]	n/a	char *
%p	void	void **
%n	int *	int *
%%	none	none

（严格地讲，%lf在printf下是未定义的，但是很多系统可能会接受它。要确保可移植性，就要坚持使用%f。）

参见问题12.15和15.2。

参考资料：[18, Sec. 7.3 pp. 145-47, Sec. 7.4 pp. 147-150]
[19, Sec. 7.2 pp. 153-44, Sec. 7.4 pp. 157-159]

① 此处的描述同样适用于%e和%g以及对应的scanf格式%le和%lg。
② 实际上，默认参数提升仅适用于可变参数列表的可变部分。参见第15章。

[35, Sec. 4.9.6.1, Sec. 4.9.6.2]
[8, Sec. 7.9.6.1, Sec. 7.9.6.2]
[11, Sec. 15.8 pp. 357-364, Sec. 15.11 pp. 366-378]
[22, Sec. A.1 pp. 121-133]

*12.10

问：对于size_t那样的类型定义，当我不知道它到底是long还是其他类型的时候，我应该使用什么样的printf格式呢？

答：把那个值转换为一个已知的长度够大的类型，然后使用与之对应的printf格式。例如，输出某种类型的长度，可以使用

```
printf("%lu", (unsigned long)sizeof(thetype));
```

12.11

问：如何用printf实现可变的域宽度？就是说，我想在运行时确定宽度而不是使用%8d？

答：使用printf("%*d", width, x)。格式说明符中的星号表示，参数列表中的一个int值用来表示域的宽度。（注意，在参数列表中，宽度在输出的值之前。）

参见问题12.17。

参考资料：[18, Sec. 7.3]
[19, Sec. 7.2]
[35, Sec. 4.9.6.1]
[8, Sec. 7.9.6.1]
[11, Sec. 15.11.6]
[22, Sec. A.1]

12.12

问：如何输出在千位上用逗号隔开的数字？货币格式的数字呢？

答：<locale.h>提供了一些函数可以完成这些操作，但是没有完成这些任务的标准方法。（printf唯一一处对应自定义区域设置的地方就是改变它的小数点字符。）

这个小函数可以格式化逗号分隔的数字，如果区域设置有千位分隔符，它也会利用：

```
#include <locale.h>

char *commaprint(unsigned long n)
{
    static int comma = '\0';
    static char retbuf[30];
    char *p = &retbuf[sizeof(retbuf)-1];
    int i = 0;

    if(comma == '\0') {
        struct lconv *lcp = localeconv();
        if(lcp != NULL) {
            if(lcp->thousands_sep != NULL &&
                *lcp->thousands_sep != '\0')
                comma = *lcp->thousands_sep;
            else
                comma = ',';

        }
    }

    *p = '\0';
    do {
        if(i%3 == 0 && i != 0)
            *--p = comma;
        *--p = '0' + n % 10;
        n /= 10;
        i++;
    } while(n != 0);

    return p;
}
```

更好的实现应该使用lconv的grouping域而不该直接假设按3位分组。对于retbuf更安全的大小可能是4*(sizeof(long)*CHAR_BIT+2)/3/3+1。参见问题12.23。

参考资料：[35, Sec. 4.4]
[8, Sec. 7.4]
[11, Sec. 11.6 pp. 301-304]

12.13

问：为什么scanf("%d", i)调用不行?

答：传给scanf的参数必须是指针：对于每个转换的值，scanf都会写入你传入的指针指向的位置（参见问题20.1。）。改为scanf("%d", &i)即可修正上面的问题。

12.14

问：为什么

```
char s[30];
scanf("%s", s);
```

不用&也可以？我原以为传给scanf的每个变量都要带&。

答：总是需要指针，但并不表示一定需要&操作符。当向scanf传入一个数组的时候，不需要使用&，因为不论是否带&操作符，数组总是以指针形式传入函数的。参见问题6.3和6.4（如果你使用了显式的&，你会得到错误的指针类型。参见问题6.11。）

12.15

问：为什么这些代码不行？

```
double d;
scanf("%f", &d);
```

答：跟printf不同，scanf用%lf代表double型，用%f代表float型。[1] %f格式告诉scanf准备接收float型指针，而不是你提供的double型指针。要么使用%lf，要么将接收变量声明为float。参见问题12.9。

12.16

问：为什么这段代码不行？

```
short int s;
scanf("%d", &s);
```

答：在转换%d的时候，scanf需要int型指针。要转换成short int，则应该使用%hd。（参见问题12.9中的表格。）

*12.17

问：怎样在scanf格式串中指定可变的宽度？

答：不能。scanf格式串中的星号表示禁止赋值。可以使用ANSI的字符串化和字符串拼接操作

① 关于%e、%g及对应的格式%le和%lg也有这样的区别。

符，基于一个包含特定宽度的预处理宏构造一个常量格式说明符：

```
#define WIDTH 3

#define Str(x) #x
#define Xstr(x) Str(x)      /* see question 11.19 */

scanf("%" Xstr(WIDTH) "d", &n);
```

但是，如果宽度是运行时变量，就只能在运行时创建格式说明符了：

```
char fmt[10];
sprintf(fmt, "%%%dd", width);
scanf(fmt, &n);
```

（对于标准输入这样的scanf格式不太可能，但对于fscanf和sscanf恐怕还有些用处。）

参见问题11.19和12.11。

12.18

问：怎样从特定格式的数据文件中读取数据？怎样读入10个float而不用使用包含10次%f的奇怪格式？如何将一行的任意多个域读入一个数组中？

答：一般来说，主要有3种分析数据行的方法：

- 使用带有正确格式串的fscanf和sscanf。虽然有本章提及的各种局限性（参见问题12.22），但scanf族的函数功能还是很强大的。尽管空白分隔的域总是最容易处理的，scanf格式串也可以用来处理更紧凑的、基于列的、FORTRAN风格的数据。例如：

  ```
  1234ABC5.678
  ```

  就可以用"%d%3s%f"读出。（参见问题12.21的最后一个例子。）
- 用strtok或等价的其他工具（参见问题13.6）将数据行分解为用空白（或其他分隔符）隔开的域，然后，用atoi或atol等函数单独处理每个域。（一旦数据行被分解为域以后，处理这些域的代码就跟传统的main()函数处理argv数组的形式类似了。参见问题20.3。）这种方法尤其适用于将任意多个（即事先不知道数量）域的一行读入一个数组中。

  这里有个简单例子，可以将最多10个浮点数的（用空白分隔的）一行读入一个数组：

  ```
  #include <stdlib.h>
  #define MAXARGS 10

  char *av[MAXARGS];
  int ac, i;
  double array[MAXARGS];
  char line[] = "1 2.3 4.5e6 789e10";
  ```

```
ac = makeargv(line, av, MAXARGS);
for(i = 0; i < ac; i++)
    array[i] = atof(av[i]);
```

（makeargv的定义参见问题13.6。）

- 使用任何就手的指针操作和库函数进行特别处理。（ANSI的strol和strtod函数对这类解析特别有用，因为它们能返回一个表明它们停止读取的位置的指针。）这是最一般的方法，但同时也是最困难和容易出错的方法，因为很多C程序中最痛苦的部分就是那些使用大量的指针技巧分解字符串的代码。

设计数据文件和输入格式的时候，尽量避免那些神秘的操作，最好采用比较简单的方法（如1和2）进行解析。这样，处理文件的时候就会轻松很多了。

scanf问题

尽管scanf看起来好像不过是和printf互补的函数，但它却有许多基本的限制，有的程序员建议干脆完全避免使用它。

12.19

问：我像这样用"%d\n"调用scanf从键盘读取数字：

```
int n;
scanf("%d\n", &n);
printf("you typed %d\n", n);
```

好像要多输入一行才返回。为什么？

答：可能令人吃惊，\n在scanf格式串中不表示等待换行符，而是读取并放弃连续的空白字符。

（事实上，scanf格式串中的任何空白字符都表示读取并放弃空白字符。而且，诸如%d这样的格式也会扔掉前边的空白，因此你通常根本不需要在scanf格式串中加入显式的空白。）

因此，"%d\n"中的\n会让scanf读到非空白字符为止，而它可能需要读到下一行才能找到这个非空白字符。这种情况下，去掉\n仅仅使用"%d"即可（但你的程序可能需要跳过那个没有读入的换行符。参见问题12.20。）

scanf函数是设计来读取自由格式的输入的，而在读取键盘输入的时候，你所得到的往往并不是你所想要的。“自由格式”意味着scanf在处理换行符的时候跟其他的空白一样。格式"%d%d%d"既可读入

```
1  2  3
```

又可以读入

```
1
2
```

```
3
```

（比较一下就可得知，C、Pascal和LISP的源码是自由格式的，而BASIC和FORTRAN的则不是。）

如果你真的要坚持，scanf的确可以用“scanset”指令读取换行符：

```
scanf("%d%*[\n]", &n);
```

scanset尽管功能强大，但还是不能解决所有的scanf问题。参见问题12.22。

参考资料：[19, Sec. B1.3 pp. 245-246]
[35, Sec. 4.9.6.2]
[8, Sec. 7.9.6.2]
[11, Sec. 15.8 pp. 357-364]

12.20

问：我用scanf和%d读取一个数字，然后再用gets()读取字符串：

```
int n;
char str[80];

printf("enter a number: ");
scanf("%d", &n);
printf("enter a string: ");
gets(str);
printf("you typed %d and \"%s\"\n", n, str);
```

但是编译器好像跳过了gets()调用！

答：如果你向问题中的程序输入两行：

```
42
a string
```

scanf会读取42，但却不会读到紧接其后的换行符。换行符会保留在输入流中，然后被gets()读取，后者会读入一个空行。而第二行的“a string”则根本不会被读取。

如果你在同一行输入数字和字符串：

```
42 a string
```

则代码会多少如你所愿地运行。

作为一个一般规则，不能混用scanf和gets或任何其他的输入例程的调用，scanf对换行符的特殊处理几乎一定会带来问题。要么就用scanf处理所有的输入，要么干脆不用。

参见问题12.22和12.25。

参考资料：[35, Sec. 4.9.6.2]
[8, Sec. 7.9.6.2]

[11, Sec. 15.8 pp. 357-364]

12.21

问：我发现如果坚持检查返回值以确保用户输入的是我期待的数值，则scanf的使用会安全很多。

```
int n;

while(1) {
    printf("enter a number: ");
    if(scanf("%d", &n) == 1)
        break;
    printf("try again: ");
}

printf("you typed %d\n", n);
```

但有的时候好像会陷入无限循环。[①]为什么？

答：在scanf转换数字的时候，它遇到的任何非数字字符都会终止转换并被保留在输入流中。因此，除非采用了其他的步骤，那么未预料到的非数字输入会不断“阻塞”scanf，因为scanf永远都不能越过错误的非数字字符而处理后边的合法数字字符。如果用户在应对前文的代码时输入类似'x'的字符，那么代码会永远循环提示“try again”，但却不会给用户重试的机会。

你可能很奇怪为什么scanf会把未匹配的字符留在输入流中。假如你有一个紧凑的数据文件，包含了由数字和字母代码组成的不带空白的行：

```
123CODE
```

你可能希望用"%d%s"格式的scanf来解析这行文本。但是，如果%d不把未匹配的字符保留在输入流中，则%s会错误地读入"ODE"而不是"CODE"。这是词法分析中的一个标准问题：在扫描任意长度的数字常量或字母数字标识符的时候，只有读到“超越”位置，你才能知道它已经结束。（这也正是ungetc存在的原因。）

参见问题12.22。

参考资料：[35, Sec. 4.9.6.2]
[8, Sec. 7.9.6.2]
[11, Sec. 15.8 pp. 357-364]

12.22

问：为什么大家都说不要使用scanf？那我该用什么来代替呢？

① 如果不能用Control-C退出或者准备重启，千万不要尝试运行问题中的代码。

答：scanf有很多问题，可参见问题12.19、12.20和12.21。而且，它的%s格式有着和gets()一样的问题（参见问题12.25），即很难保证接收的缓冲区不溢出。[①]

更一般地讲，scanf的设计适用于相对结构化的、格式整齐的输入。设计上，它的名称就是来自“scan formatted”。如果你注意，它会告诉你成功或失败，但它只能提供失败的大致位置，至于失败的原因，就无从得知了。对scanf做错误恢复几乎是不可能的。

而交互的用户输入又是最缺乏格式化的输入。设计良好的用户界面应该允许用户输入各种东西——不仅仅是在等待数字的时候输入了字母或标点，还包括输入过短、过长、根本没有字符输入（例如，直接按了回车键）、提前的EOF或其他任何东西。使用scanf来优雅地处理所有这些潜在问题几乎不可能。可以先用fgets这样的函数读入整行，然后再用scanf或其他技术进行解释。（strtol、strtok和atoi等函数通常有用。参见问题12.18和13.6。）如果真的要用scanf的任何变体，一定要检查返回值，以确定是否找到了期待的值。而使用%s格式的时候,一定要小心缓冲区溢出。

另外要注意，对scanf的种种诟病并不一定也适用于fscanf和sscanf。scanf读入的标准输入通常都是交互的键盘，因此所受约束最少也导致问题最多。而如果数据文件的格式已知，则使用fscanf就可能很合适了。（只要检查了返回值）用sscanf来处理字符串也很适宜，因为如果不能匹配可以很容易地恢复控制、重启扫描或放弃输入。

参考资料：[19, Sec. 7.4 p. 159]

其他stdio函数

12.23

问：我怎样才知道对于任意的sprintf调用需要多大的目标缓冲区？怎样才能避免sprintf目标缓冲区溢出？

答：对这两个极好的问题（暂时还）没有什么好答案。而这也可能正是传统stdio库最大的弱点。

当用于sprintf的格式串已知且相对简单时，有时可以预测出缓冲区的大小。如果格式串中包含一个或两个%s，你可以数出固定字符的个数（或用sizeof计算）再加上对插入的字符串的strlen调用的返回值。对于整型，%d输出的字符数不会超过

```
((sizeof(int) * CHAR_BIT + 2) / 3 + 1)    /* +1 for '-' */
```

CHAR_BIT在<limits.h>中定义，但是这个计算可能有些过于保守了。它计算的是数字以八进制存储需要的字节数，十进制的存储可以保证使用同样或更少的字节数。

当格式串更复杂或者在运行前未知的时候，预测缓冲区大小会变得跟重新实现sprintf

① 指明域宽度，如%20s，可能会有帮助。参见问题12.17。

一样困难，而且会很容易出错。有一种最后防线的技术，就是用fprintf向一块内存区或临时文件输出同样的内容，然后检查fprintf的返回值或临时文件的大小，但请参见问题19.13，并提防写文件错误。

如果不能确保缓冲区足够大，就不能调用sprintf，以防缓冲区溢出后改写其他的内存区。如果格式串已知，可以用%.Ns控制%s扩展的长度，或者使用%.*s。参见问题12.11。

要避免溢出问题，可以使用限制长度的sprintf版本，即snprintf。这样使用：

```
snprintf(buf, bufsize, "You typed \"%s\"", answer);
```

snprintf在几个stdio库中已经提供好几年了，包括GNU和4.4bsd。在C99中已经被标准化了。

还有一个好处是，C99的snprintf提供了预测任意sprintf调用所需的缓冲区大小的方法。C99的snprintf返回它可能放到缓冲区的字符数，而它又可以用空指针和缓冲区大小0进行调用。因此，

```
nch = snprintf(NULL, 0, fmtstring, /*other arguments*/);
```

这样的调用就可以预测出格式串扩展后所需要的字符数。

另一个（非标准的）选择是asprintf函数，在BSD和GNU的C库中都有提供，它调用malloc为格式串分配空间，并返回分配内存区的指针。这样使用:

```
char *buf;
asprintf(&buf, "%d=%s", 42, "forty-two");
    /*now buf points to malloc'ed space containing formatted string*/
```

参考资料：[9, Sec. 7.13.6.6]

12.24

问： sprintf的返回值是什么？是int还是char *？

答： 标准声称它返回int值（写入的字符数，跟printf和fprintf一样）。曾经有段时间，在某些C语言的库中，sprintf用char *返回它的第一个参数，指向完成的结果（即跟strcpy的返回值类似）。

12.25

问： 为什么大家都说不要使用gets？

答： 跟fgets不同，gets不能被告知输入缓冲区的大小，因此一旦输入行太长，则无法避免缓冲

区的溢出——墨菲定律告诉我们，迟早都会出现超长的输入行[①]。作为一个一般规则，永远使用fgets。（你可能会认为，由于这样那样的原因，你的程序中不会出现超过最大限制的输入行，但是也可能出错[②]，况且，不管怎么说，用fgets和gets一样简单。）

fgets和gets的另一个区别是fgets保留'\n'，但可以很容易地将它扔掉。问题7.1中有一段代码，演示了如何用fgets代替gets。参见问题7.1 中用fgets代替gets的代码片段。

参考资料：[14, Sec. 4.9.7.2]
[11, Sec. 15.7 p. 356]

12.26

问：我觉得我应该在一长串的printf调用之后检查errno，以确定是否有失败的调用：

```
errno = 0;
printf("This\n");
printf("is\n");
printf("a\n");
printf("test.\n");
if(errno != 0)
    fprintf(stderr, "printf failed: %s\n", strerror(errno));
```

为什么当我将输出重定向到文件的时候会输出奇怪的“printf failed: Not a typewriter”信息？

答：如果stdout是终端，stdio库的很多实现都会对其行为进行细微的调整。为了做出判断，这些实现会执行某些当stdout不为终端时会失败的操作。尽管输出操作成功完成，errno还是会被置入错误代码。这种行为确实有点令人困惑，但严格地讲，它并不错误，因为只有当函数报告错误之后检查errno的内容才有意义。（更严格地讲，只有当库函数在出错时设置errno并返回错误代码的时候，errno才有意义。）

一般来说，最好通过检查函数的返回值来检测错误。要检查一连串的stdio调用之后的累积错误，可以使用ferror。参见问题12.3和20.4。

参考资料：[8, Sec. 7.1.4, Sec. 7.9.10.3]
[22, Sec. 5.4 p. 73]
[12, Sec. 14 p. 254]

12.27

① 在讨论gets的缺点的时候，人们会习惯性地指出1988年的“因特网蠕虫”是利用了UNIX finger程序的gets调用作为攻击手段之一的。它用仔细设计的二进制数据使gets溢出，从而改写了栈上的一个返回地址，导致控制流转向了二进制数据。

② 你可能会认为你的操作系统会限制最大的键盘输入行长度，可如果输入是从文件重定向的呢？

问： fgetops/fsetops和ftell/fseek之间有什么区别？fgetops和fsetops到底有什么用处？

答： ftell和fseek用long int型表示文件内的偏移量（位置），因此，偏移量被限制在2G($2^{31}-1$)以内。而新的fgetpos和fsetpos函数使用了一个特殊的类型定义fpos_t来表示偏移量。适当选择这个类型后，fgetpos和fsetpos可以表示任意大小的文件偏移量。fgetpos和gsetpos也可以用来记录多字节流式文件的状态。参见问题1.4。

参考资料：[19, Sec. B1.6 p. 248]
[35, Sec. 4.9.1, Secs. 4.9.9.1, 4.9.9.3]
[8, Sec. 7.9.1, Secs. 7.9.9.1, 7.9.9.3]
[11, Sec. 15.5 p. 252]

12.28

问： 如何清除用户的多余输入，以防止在下一个提示符下读入？用fflush(stdin)可以吗？

答： 在标准C中，fflush()仅对输出流有效。因为它对“flush”的定义是用于完成缓冲字符的写入（而不是扔掉他们），放弃未读取的输入并不是fflush在输入流上的类比意义。

没有什么标准的方法用来放弃输入流中未读取的数据。有些厂商的确实现fflush，让fflush(stdin)放弃未读取的字符，但可移植程序不能依靠这样的实现。（有些版本的stdio库实现了fpurge和fabor调用，可以完成同样的工作，但这些也不是标准的。）同时应该注意，仅仅清空标准输入的缓冲区并不一定足够，未读取的字符还可能在其他操作系统级的缓冲区上累积。

如果你需要清空输入，得使用某种系统特有的技术，如fflush(stdin)（如果它碰巧能完整这项工作）或其他操作系统相关的例程（如问题19.1和19.2）。不过要记住，如果你扔掉了用户输入太快的字符，他/她可能会感觉十分受挫。

参考资料：[35, Sec. 7.9.5.2]
[8, Sec. 7.9.5.2]
[11, Sec. 15.2]

打开和操作文件

12.29

问：我写了一个函数用来打开文件：

```
myfopen(char *filename, FILE *fp)
{
    fp = fopen(filename, "r");
}
```

可我这样调用的时候：

```
FILE *infp;
myfopen("filename.dat", infp);
```

infp指针并没有正确设置。为什么？

答：C语言的函数总是接收参数的副本，因此函数永远不能通过向参数赋值“返回”任何东西。参见问题4.8。

对于这个例子，一种解决方法是让myfopen返回FILE *：

```
FILE *myfopen(char *filename)
{
    FILE *fp = fopen(filename, "r");
    return fp;
}
```

然后这样调用：

```
FILE *infp;
infp = myfopen("filename.dat");
```

另外，也可以让myfopen接受FILE *的指针（FILE的指针的指针）：

```
myfopen(char *filename, FILE **fpp)
{
    FILE *fp = fopen(filename, "r");
    *fpp = fp;
}
```

然后这样调用：

```
FILE *infp;
myfopen("filename.dat", &infp);
```

12.30

问：连一个最简单的fopen调用都不成功！这个调用有什么问题？

```
FILE *fp = fopen(filename, 'r');
```

答：问题在于fopen的mode参数必须是字符串，如"r"，而不是字符（'r'）。参见问题8.1。

12.31

问：为什么我不能用完整路径名打开一个文件？这个调用总是失败：

```
fopen("c:\newdir\file.dat", "r");
```

答：你可能需要重复那些反斜杠。参见问题19.20。

12.32

问：我想用fopen模式"r+"打开一个文件，读出一个字符串，修改之后再写入，从而就地更新一个文件。可是这样不行。为什么？

答：确保在写操作之前先调用fseek，回到你准备覆盖的字符串的开始，况且在读写“+”模式下的读和写操作之间总是需要fseek或fflush。同时,记住改写同样数量的字符，而且在文本模式下改写可能会在改写处把文件长度截断，因而你可能需要保存行长度。参见问题19.16。

参考资料：[35, Sec. 4.9.5.3]
[8, Sec. 7.9.5.3]

12.33

问：如何在文件中间插入或删除一行（一条记录）？

答：参见问题19.16。

12.34

问：怎样从打开的流中恢复文件名？

答：参见问题19.17。

重定向stdin和stdout

12.35

问：怎样在程序里把stdin或stdout重定向到文件？

答：使用freopen。如果你希望通常写入stdout的函数f()将输出发送到一个文件，而又不能改写f的代码，可以使用这样的调用序列：

```
freopen(file, "w", stdout);
f();
```

但请参见问题12.36。

参考资料：[35, Sec. 4.9.5.4]
[8, Sec. 7.9.5.4]
[11, Sec. 15.2 pp. 347-348]

12.36

问：一旦使用freopen之后，怎样才能恢复原来的stdout（或stdin）？

答：没有什么好办法。如果你需要恢复回去，那么最好一开始就不要使用freopen。可以使用你自己的可以随意赋值的输出（输入）流变量，而不要去动原来的stdout（或stdin）。例如，声明一个全局变量

```
FILE *ofp;
```

然后用fprintf(ofp,...)代替printf(...)调用。（很明显，你还需要检查putchar和puts调用。）然后就可以将ofp置为stdout或其他任何东西了。

你也许在想是否可以这样完全跳过freopen：

```
FILE *savestdout = stdout;
stdout = fopen(file, "w");          /*WRONG*/
```

然后再用

```
stdout = savestdout;                /*WRONG*/
```

来恢复stdout。

这样的代码恐怕不行，因为stdout（及stdin和stderr）通常都是常量，不能被赋值（乍看起来，这也正是freopen存在的原因）。

有一种不可移植的办法，可以在调用freopen()之前保存流的信息，以便其后恢复原来的流。一种办法是使用系统相关的调用如dup()、dup2()等。另一种办法是复制或查看FILE结构的内容，但是这种方法完全没有可移植性，而且很不可靠。

某些系统下，你可以显式地打开控制终端（参见问题12.38），但这也不一定就是你所需要的，因为原来的输入或输出（即在调用freopen之前的stdin和stdout）可能已经在命

令行被重定向了。

如果你想获取一个子程序的执行结果，freopen可能无论如何都不行。可以参见问题19.35。

12.37

问：如何判断标准输入或输出是否经过了重定向，即是否在命令行上使用了“<”或“>”？

答：不能直接判断，但是通常可以查看其他东西以帮助你做出判断。如果你希望你的程序在没有输入文件的时候从stdin获取输入，那么只要argv没有提供输入文件或者提供了占位符（如"-"）而不是文件名，就可以从stdin获取输入了。如果你希望在输入不是来自交互终端的时候禁止输出，那么在某些系统（如UNIX和MS-DOS）下，可以使用isatty(0)或isatty(fileno(stdin))来做出判断。

12.38

问：我想写个像"more"那样的程序。怎样才能在stdin被重定向之后再回到交互键盘？

答：没有可移植的办法来完成这个任务。在UNIX下，可以打开特殊文件/dev/tty。在MS-DOS下，可以尝试打开“文件”CON或使用BIOS调用，如getch。无论是否进行了输入重定向，它都会获取键盘输入。

*12.39

问：怎样同时向两个地方输出，如同时输出到屏幕和文件？

答：直接做不到这点。但是你可以写出你自己的printf变体，把所有的内容都输出两次。下边有个简单的例子：

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

void f2printf(FILE *fp1, FILE *fp2, char *fmt,...)
{

    va_list argp;
    va_start(argp, fmt); vfprintf(fp1, fmt, argp); va_end(argp);
    va_start(argp, fmt); vfprintf(fp2, fmt, argp); va_end(argp);
```

```
}
```

这里的f2printf就跟fprintf一样，只是它接受两个文件指针（如stdout和logfp）并同时输出到两个文件。

参见问题15.5。

“二进制”输入输出

普通的流包含可打印的文本，可能会被适当地转换以适应底层的操作系统习惯。如果想准确无误地读写任意数据，拒绝任何转换，需要“二进制”输入输出。

12.40

问：我希望按字节在内存和文件之间直接读写数字，而不像fprintf和fscanf进行格式化。我该怎么办？

答：你想完成的一般称为“二进制”输入输出。首先，确保你使用了"b"（"rb"、"wb"等。参见问题12.41）修饰符调用fopen。然后用&和sizeof操作符获取准备传输的字节序列的句柄。通常，需要使用fread和fwrite函数。参见问题2.12的例子。

但是注意，fread和fwrite并不一定代表二进制输入输出。如果你已经用二进制方式打开了文件，就可以在其上使用任何输入输出调用了（参见问题12.45 中的例子）。如果用文本方式打开了文件，也可以在方便的时候使用fread和fwrite。

最后，注意二进制文件不太可移植。参见问题20.5。

参见问题12.43。

12.41

问：怎样正确地读取二进制文件？有时看到0x0a和0x0d容易混淆，而且如果数据中包含0x1a的话，我好像会提前遇到EOF。

答：读取二进制数据文件的时候应该用"rb"调用fopen，以确保不会发生文本文件的解释。类似地，写二进制文件时使用"wb"。（在类似UNIX那样的不区分文本文件和二进制文件的系统中，"b"可以省略，但加上也没有什么坏处。）

注意文本/二进制区别只是发生在文件打开时，一旦文件打开之后，在其上调用何种I/O函数无关紧要。

参见问题12.43、12.45和20.5。

参考资料：[35, Sec. 4.9.5.3]

[8, Sec. 7.9.5.3]
[11, Sec. 15.2.1 p. 348]

12.42

问：我在写一个二进制文件的“过滤器”，但是stdin和stdout却被作为文本流打开了。怎样才能把它们的模式改为二进制？

答：没有标准的方法来完成这个任务。类UNIX系统没有文本/二进制文件的区别，因此也就没有修改模式的必要。有些MS-DOS编译器提供setmode调用。其他情况下，你就只能靠自己了。

12.43

问：文本和二进制输入输出有什么区别？

答：文本模式下，文件应该包含可打印的字符行（可能包括tab字符）。stdio库的例程（getc、putc和其他函数）完成C程序中的\n和底层操作系统的行结束符之间的转换。因此读写文本文件的C程序无需考虑底层系统换行符习惯。当C程序写入'\n'的时候，底层库会写入正确的换行字符，而stdio库检测到行结束的时候，它也会向调用程序返回入'\n'。[①]

而二进制方式下，数据在程序和文件之间读写的时候没有经过任何解释。（在MS-DOS系统下，二进制方式也会关掉对Control-Z作为文件结束符的检测。）

文本方式的转换也会在读入的时候影响到文件表面上的大小。因为文本方式下读出和写入的字符不一定和文件中的字符完全相同，磁盘文件的大小也不一定和可以读出的字符数相等。而且，基于类似的原因，fseek和ftell处理的也不一定是从文件开始的纯的字节数。（严格地讲，在文本方式下，fseek和ftell使用的偏移值根本就不能解释。ftell的返回值只能再用作fseekd的参数，而fseekd的参数也只能使用ftell的返回值。）

二进制方式下，fseek和ftell的确使用纯的字节偏移。但是，某些系统可能会在二进制文件的尾部添加一些空字节，用以补全一条记录。

参见问题12.40和19.13。

参考资料：[35, Sec. 4.9.2]
[8, Sec. 7.9.2]
[14, Sec. 4.9.2]

① 有些系统可能会以空格填充的记录的方式保存文本文件。在这些系统上，尾部空格在以文本方式读取的时候会被丢弃，因此显式写入的任何尾部空格也会在再读入的时候丢失。

[11, Sec. 15 p. 344, Sec.15.2.1 p. 348]

12.44

问：如何在数据文件中读写结构？

答：参见问题2.12。

12.45

问：怎样编写符合旧的二进制数据格式的代码？

答：由于机器字大小和字节顺序差别、浮点数格式以及结构填充等问题，要这么做很困难。要控制这些细节，你可能得一次一字节地读写，边看边调整。（这并不一定像听上去那么坏，至少它让你写出可移植的代码，同时又能全盘掌控。）

例如，假设你要从流fp读入一个数据结构（包含一个字符、一个32位整数和一个16位整数）到C结构，

```
struct mystruct {
    char c;
    long int i32;
    int i16;
};
```

可以使用这样的代码：

```
s.c = getc(fp);

s.i32 = (long)getc(fp) << 24;
s.i32 |= (long)getc(fp) << 16;
s.i32 |= (unsigned)(getc(fp) << 8);
s.i32 |= getc(fp);

s.i16 = getc(fp) << 8;
s.i16 |= getc(fp);
```

这段代码假设getc读取8位字符而且数据以高字节在前（“大端”）方式存储。转换成(long)可以确保16位和24位移位作用在long型值上(参见问题3.16)，转换成(unsigned)可以防止符号扩展。(一般而言，写这类代码的时候都使用unsigned类型会更安全。但请参见3.21。

编写这个结构的对应代码如下：

```
putc(s.c, fp);
```

```
putc((unsigned)((s.i32 >> 24) & 0xff), fp);
putc((unsigned)((s.i32 >> 16) & 0xff), fp);
putc((unsigned)((s.i32 >> 8) & 0xff), fp);
putc((unsigned)(s.i32 & 0xff), fp);


putc((s.i16 >> 8) & 0xff, fp);
putc(s.i16 & 0xff, fp);
```

参见问题2.13、12.41和20.5。