第15章

输入/输出函数

ANSI C 和早期 C 相比的最大优点之一就是它在规范里所包含的函数库。每个 ANSI 编译器必须支持一组规定的函数，并具备规范所要求的接口，而且按照规定的行为工作。这种情况较之早期的 C 是一个巨大的改进。以前，不同的编译器可以通过修改或扩展普通函数库的功能来“改善”它们。这些改变可能在那个特定的作出修改的系统上很有用，但它们却限制了可移植性，因为依赖这些修改的代码在其他缺乏这些修改（或者具有不同修改）的编译器上将会失败。

ANSI 编译器并未被禁止在它们的函数库的基础上增加其他函数。但是，标准函数必须根据标准所定义的方式执行。如果你关心可移植性，只要避免使用任何非标准函数就可以了。

本章讨论 ANSI C 的输入和输出（I/O）函数。但是，我们首先学习两个非常有用的函数，它们用于报告错误以及对错误作出反应。

15.1 错误报告

perror 函数以一种简单、统一的方式报告错误。ANSI C 函数库的许多函数调用操作系统来完成某些任务，I/O 函数尤其如此。任何时候，当操作系统根据要求执行一些任务的时候，都存在失败的可能。例如，如果一个程序试图从一个并不存在的磁盘文件读取数据，操作系统除了提示发生了错误之外就没什么好做的了。标准库函数在一个外部整型变量 errno（在 errno.h 中定义）中保存错误代码之后把这个信息传递给用户程序，提示操作失败的准确原因。

perror 函数简化向用户报告这些特定错误的过程。它的原型定义于 stdio.h，如下所示：

```
void  perror( char const *message );
```

如果 message 不是 NULL 并且指向一个非空的字符串，perror 函数就打印出这个字符串，后面跟一个分号和一个空格，然后打印出一条用于解释 errno 当前错误代码的信息。

提示：

perrno 最大的优点就是它容易使用。良好的编程实践要求任何可能产生错误的操作都应该在执行之后进行检查，确定它是否成功执行。即使是那些十拿九稳不会失败的操作也应该进行检查，因为它们迟早可能失败。这种检查需要稍许额外的工作，但与你可能付出的大量调试时间相比，它们

还是非常值得的。perror 将在本章许多地方以例子的方式进行说明。

注意，只有当一个库函数失败时，errno 才会被设置。当函数成功运行时，errno 的值不会被修改。这意味着我们不能通过测试 errno 的值来判断是否有错误发生。反之，只有当被调用的函数提示有错误发生时检查 errno 的值才有意义。

15.2 终止执行

另一个有用的函数是 exit，它用于终止一个程序的执行。它的原型定义于 stdlib.h，如下所示：

```
void  exit( int status );
```

status 参数返回给操作系统，用于提示程序是否正常完成。这个值和 main 函数返回的整型状态值相同。预定义符号 EXIT_SUCCESS 和 EXIT_FAILURE 分别提示程序的终止是成功还是失败。虽然程序也可以使用其他的值，但它们的具体含义将取决于编译器。

当程序发现错误情况使它无法继续执行下去时，这个函数尤其有用。你经常会在调用 perrno 之后再调用 exit 终止程序。尽管终止程序并非处理所有错误的正确方法，但和一个注定失败的程序继续执行以后再失败相比，这种做法更好一些。

注意这个函数没有返回值。当 exit 函数结束时，程序已经消失，所以它无处可返。

15.3 标准 I/O 函数库

K&R C 最早的编译器的函数库在支持输入和输出方面功能甚弱。其结果是，程序员如果需要使用比函数库所提供的 I/O 更为复杂的功能时，他不得不自己实现。

有了标准 I/O 函数库(Standard I/O Library)之后，这种情况得到了极大的改观。标准 I/O 函数库具有一组 I/O 函数，实现了在原先的 I/O 库基础上许多程序员自行添加实现的额外功能。这个函数库对现存的函数进行了扩展，例如为 printf 创建了不同的版本，可以用于各种不同的场合。函数库同时引进了缓冲 I/O 的概念，提高了绝大多数程序的效率。

这个函数库存在两个主要的缺陷。首先，它是在某台特定类型的机器上实现的，并没有对其他具有不同特性的机器作过多考虑。这就可能出现一种情况，就是在某台机器上运行良好的代码在另一台机器上无法运行，原因仅仅是两台机器之间的架构不同。第 2 个缺陷与第 1 个缺陷有直接有关系。当设计者发现上述问题后，他们试图通过修改库函数进行修正。但是，只要他们这样做了，这个函数库就不再“标准”，程序的可移植性就会降低。

ANSI C 函数库中的 I/O 函数是旧式标准 I/O 库函数的直接后代，只是这些 ANSI 版函数作了一些改进。在设计 ANSI 函数库时，可移植性和完整性是两个关键的考虑内容。但是，与现有程序的向后兼容性也不得不予以考虑。ANSI 版函数和它们的祖先之间的绝大多数区别就是那些在可移植性和功能性方面的改进。

对可移植性最后再说一句：这些函数是对原来的函数进行诸多完善之后的结果，但是它们仍可能进一步改进，使它们变得更完美。ANSI C 的一个主要优点就是这些修改将通过增加**不同**函数的方式实现，而不是通过对现存函数进行修改来实现。因此，程序的可移植性不会受到影响。

15.4　ANSI I/O 概念

头文件 stdio.h 包含了与 ANSI 函数库的 I/O 部分有关的声明。它的名字来源于旧式的标准 I/O 函数库。尽管不包含这个头文件也可以使用某些 I/O 函数，但绝大多数 I/O 函数在使用前都需要包含这个头文件。

15.4.1　流

当前的计算机具有大量不同的设备，很多都与 I/O 操作有关。CD-ROM 驱动器、软盘和硬盘驱动器、网络连接、通信端口和视频适配器就是这类很常见的设备。每种设备具有不同的特性和操作协议。操作系统负责这些不同设备的通信细节，并向程序员提供一个更为简单和统一的 I/O 接口。

ANSI C 进一步对 I/O 的概念进行了抽象。就 C 程序而言，所有的 I/O 操作只是简单地从程序移进或移出字节的事情。因此，毫不惊奇的是，这种字节流便被称为流(stream)。程序只需要关心创建正确的输出字节数据，以及正确地解释从输入读取的字节数据。特定 I/O 设备的细节对程序员是隐藏。

绝大多数流是完全缓冲的(fully buffered)，这意味着“读取”和“写入”实际上是从一块被称为缓冲区(buffer)的内存区域来回复制数据。从内存中来回复制数据是非常快速的。用于输出流的缓冲区只有当它写满时才会被刷新（flush，物理写入）到设备或文件中。一次性把写满的缓冲区写入和逐片把程序产生的输出分别写入相比效率更高。类似，输入缓冲区当它为空时通过从设备或文件读取下一块较大的输入，重新填充缓冲区。

使用标准输入和输出时，这种缓冲可能会引起混淆。所以，只有当操作系统可以断定它们与交互设备并无联系时才会进行完全缓冲。否则，它们的缓冲状态将因编译器而异。一个常见（但并不普遍）的策略是把标准输出和标准输入联系在一起，就是当请求输入时同时刷新输出缓冲区。这样，在用户必须进行输入之前，提示用户进行输入的信息和以前写入到输出缓冲区中的内容将出现在屏幕上。

警告：

尽管这种缓冲通常是我们所需的，但当你调试程序时仍可能引起混淆。一个常见的调试策略是把一些 printf 函数的调用散布于程序中，确定错误出现的具体位置。但是，这些函数调用的输出结果被写入到缓冲区中，并不立即显示于屏幕上。事实上，如果程序失败，缓冲输出可能不会被实际写入，这就可能使程序员得到关于错误出现位置的不正确结论。这个问题的解决方法就是在每个用于调试的 printf 函数之后立即调用 fflush，如下所示：

```
printf("something or other" );
fflush( stdout );
```

fflush（本章后面将有更多描述）迫使缓冲区的数据立即写入，不管它是否已满。

一、文本流

流分为两种类型，文本(text)流和二进制(binary)流。文本流的有些特性在不同的系统中可能不同。其中之一就是文本行的最大长度。标准规定至少允许 254 个字符。另一个可能不同的特性是文本行的结束方式。例如，在 MS-DOS 系统中，文本文件约定以一个回车符和一个换行符（或称为行反馈

符）结尾。但是，UNIX 系统只使用一个换行符结尾。

提示：

标准把文本行定义为零个或多个字符，后面跟一个表示结束的换行符。对于那些文本行的外在表现形式与这个定义不同的系统上，库函数负责外部形式和内部形式之间的翻译。例如，在 MS-DOS 系统中，在输出时，文本中的换行符被写成一对回车/换行符。在输入时，文本中的回车符被丢弃。这种不必考虑文本的外部形式而操纵文本的能力简化了可移植程序的创建。

二、二进制流

另一方面，二进制流中的字节将完全根据程序编写它们的形式写入到文件或设备中，而且完全根据它们从文件或设备读取的形式读入到程序中。它们并未作任何改变。这种类型的流适用于非文本数据，但是如果你不希望 I/O 函数修改文本文件的行末字符，也可以把它用于文本文件。

15.4.2 文件

stdio.h 所包含的声明之一就是 FILE 结构。请不要把它和存储于磁盘上的数据文件相混淆。FILE 是一个数据结构，用于访问一个流。如果你同时激活了几个流，每个流都有一个相应的 FILE 与它关联。为了在流上执行一些操作，你调用一些合适的函数，并向它们传递一个与这个流关联的 FILE 参数。

对于每个 ANSI C 程序，运行时系统必须提供至少三个流——标准输入(standard input)、标准输出(standard output)和标准错误(standard error)。这些流的名字分别为 stdin、stdout 和 stderr，它们都是一个指向 FILE 结构的指针。标准输入是缺省情况下输入的来源，标准输出是缺省的输出设置。具体的缺省值因编译器而异，通常标准输入为键盘设备，标准输出为终端或屏幕。

许多操作系统允许用户在程序执行时修改缺省的标准输入和输出设备。例如，MS-DOS 和 UNIX 系统都支持用下面这种方法进行输入/输出重定向：

```
$ program < data > answer
```

当这个程序执行时，它将从文件 data 而不是键盘作为标准输入进行读取，它将把标准输出写入到文件 answer 而不是屏幕上。请查阅你的系统文档中有关 I/O 重定向的细节。

标准错误就是错误信息写入的地方。perror 函数把它的输出也写到这个地方。在许多系统中，标准输出和标准错误在缺省情况下是相同的。但是，为错误信息准备一个不同的流意味着，即使标准输出重定向到其他地方，错误信息仍将出现在屏幕或其他缺省的输出设备上。

15.4.3 标准 I/O 常量

在 stdio.h 中定义了数量众多的与输入和输出有关的常量。你已经见过的 EOF 是许多函数的返回值，它提示到达了文件尾。EOF 所选择的实际值比一个字符要多几位，这是为了避免二进制值被错误地解释为 EOF。

一个程序同时最多能够打开多少个文件呢？它和编译器有关，但可以保证你能够同时打开至少 FOPEN_MAX 个文件。这个常量包括了三个标准流，它的值至少是 8。

常量 FILENAME_MAX 是一个整型值，用于提示一个字符数组应该多大以便容纳编译器所支持的最长合法文件名。如果对文件名的长度没有一个实际的限制，那个这个常量的值就是文件名的推荐最大长度。其余的一些常量将在本章剩余部分和使用它们的函数一起描述。

15.5　流 I/O 总览

标准库函数使我们在 C 程序中执行与文件相关的 I/O 任务非常方便。下面是关于文件 I/O 的一般概况。

1．程序为必须同时处于活动状态的每个文件声明一个指针变量，其类型为 FILE *。这个指针指向这个 FILE 结构，当它处于活动状态时由流使用。

2．流通过调用 fopen 函数打开。为了打开一个流，你必须指定需要访问的文件或设备以及它们的访问方式（例如，读、写或者既读又写）。fopen 和操作系统验证文件或设备确实存在（在有些操作系统中，还验证你是否允许执行你所指定的访问方式）并初始化 FILE 结构。

3．然后，根据需要对该文件进行读取或写入。

4．最后，调用 fclose 函数关闭流。关闭一个流可以防止与它相关联的文件被再次访问，保证任何存储于缓冲区的数据被正确地写到文件中，并且释放 FILE 结构使它可以用于另外的文件。

标准流的 I/O 更为简单，因为它们并不需要打开或关闭。

I/O 函数以三种基本的形式处理数据：单个字符、文本行和二进制数据。对于每种形式，都有一组特定的函数对它们进行处理。表 15.1 列出了用于每种 I/O 形式的函数或函数家族。函数家族在表中以斜体表示，它指一组函数中的每个都执行相同的基本任务，只是方式稍有不同。这些函数的区别在于获得输入的来源或输出写入的地方不同。这些变种用于执行下面的任务：

表 15.1　　执行字符、文本行和二进制 I/O 的函数

函数名或函数家族名

数据类型	输　入	输　出	描　述
字符	*getchar*	*putchar*	读取（写入）单个字符
文本行	*gets* *scanf*	*puts* *printf*	文本行未格式化的输入（输出） 格式化的输入（输出）
二进制数据	fread	fwrite	读取（写入）二进制数据

1．只用于 stdin 或 stdout。

2．随作为参数的流使用。

3．使用内存中的字符串而不是流。

需要一个流参数的函数将接受 stdin 或 stdout 作为它的参数。有些函数家族并不具备用于字符串的变种函数，因为使用其他语句或函数来实现相同的结果更为容易。表 15.2 列出了每个家族的函数。各个函数将在本章的后面详细描述。

表 15.2　　输入/输出函数家族

家　族　名	目　　的	可用于所有的流	只用于 stdin 和 stdout	内存中的字符串
getchar	字符输入	fgetc, getc	getchar	①
putchar	字符输出	fputc, putc	putchar	①
gets	文本行输入	fgets	gets	②
puts	文本行输出	fputs	puts	②
scanf	格式化输入	fscanf	scanf	sscanf
printf	格式化输出	fprintf	printf	sprintf

① 对指针使用下标引用或间接访问操作从内存获得一个字符（或向内存写入一个字符）。

② 使用 strcpy 函数从内存读取文本行（或向内存写入文本行）。

15.6 打开流

fopen 函数打开一个特定的文件，并把一个流和这个文件相关联。它的原型如下所示：

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

两个参数都是字符串。name 是你希望打开的文件或设备的名字。创建文件名的规则在不同的系统中可能各不相同，所以 fopen 把文件名作为一个字符串而不是作为路径名、驱动器字母、文件扩展名等各准备一个参数。这个参数指定要打开的文件——FILE *变量的名字是程序用来保存 fopen 的返回值的，它并不影响哪个文件被打开。mode（模式）参数提示流是用于只读、只写还是既读又写，以及它是文本流还是二进制流。下面的表格列出了一些常用的模式。

	读　取	写　入	添　加
文本	“r”	“w”	“a”
二进制	“rb”	“wb”	“ab”

mode 以 r、w 或 a 开头，分别表示打开的流用于读取、写入还是添加。如果一个文件打开是用于读取的，那么它必须是原先已经存在的。但是，如果一个文件打开是用于写入的，如果它原先已经存在，那么它原来的内容就会被删除。如果它原先不存在，那么就创建一个新文件。如果一个打开用于添加的文件原先并不存在，那么它将被创建。如果它原先已经存在，它原先的内容并不会被删除。无论在哪一种情况下，数据只能从文件的尾部写入。

在 mode 中添加“a +”表示该文件打开用于更新，并且流既允许读也允许写。但是，如果你已经从该文件读入了一些数据，那么在你开始向它写入数据之前，你必须调用其中一个文件定位函数（fseek、fsetpos、rewind，它们将在本章稍后描述）。在你向文件写入一些数据之后，如果你又想从该文件读取一些数据，你首先必须调用 fflush 函数或者文件定位函数之一。

如果 fopen 函数执行成功，它返回一个指向 FILE 结构的指针，该结构代表这个新创建的流。如果函数执行失败，它就返回一个 NULL 指针，errno 会提示问题的性质。

警告：

你应该始终检查 fopen 函数的返回值！如果函数失败，它会返回一个 NULL 值。如果程序不检查错误，这个 NULL 指针就会传给后续的 I/O 函数。它们将对这个指针执行间接访问，并将失败。下面的例子说明了 fopen 函数的用法。

```
FILE    *input;

input = fopen( "data3", "r" );
if( input == NULL ){
        perror( "data3" );
        exit( EXIT_FAILURE );
}
```

首先，fopen 函数被调用。这个被打开的文件名叫 data3，打开用于读取。这个步骤之后就是非常重要的对返回值的检查，确定文件打开是否成功。如果失败，错误就被报告给用户，程序也将终止。调用 perror 所产生的确切输出结果在不同的操作系统中可能各不相同，但它大致应该像下面这个样子：

```
data3: No such file or directory
```

这种类型的信息清楚地向用户报告有一个地方出了差错，并很好地提示了问题的性质。在那些读取文件名或者从命令行接受文件名的程序中，报告这些错误尤其重要。当用户输入一个文件名时，存在出错的可能性。显然，描述性的错误信息能够帮助用户判断哪里出了错以及如何修正它。

freopen 函数用于打开（或重新打开）一个特定的文件流。它的原型如下：

```
FILE *freopen( char const *filename, char const *mode,      FILE *stream );
```

最后一个参数就是需要打开的流。它可能是一个先前从 fopen 函数返回的流，也可能是标准流 stdin、stdout 或 stderr。

这个函数首先试图关闭这个流，然后用指定的文件和模式重新打开这个流。如果打开失败，函数返回一个 NULL 值。如果打开成功，函数就返回它的第 3 个参数值。

15.7　关闭流

流是用函数 fclose 关闭的，它的原型如下：

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

对于输出流，fclose 函数在文件关闭之前刷新缓冲区。如果它执行成功，fclose 返回零值，否则返回 EOF。

程序 15.1 把它的命令行参数解释为一列文件名。它打开每个文件并逐个对它们进行处理。如果有任何一个文件无法打开，它就打印一条包含该文件名的错误信息。然后程序继续处理列表中的下一个文件名。退出状态(exit_status)取决于是否有错误发生。

我早先说过任何有可能失败的操作都应该进行检查，确定它是否成功执行。这个程序对 fclose 函数的返回值进行了检查，看看是否有什么地方出现了问题。许多程序员懒得执行这个测试，他们争辩说关闭文件没理由失败。更何况，此时对这个文件的操作已经结束，即使 fclose 函数失败也并无大碍。然而，这个分析并不完全正确。

```
/*
** 处理每个文件名出现于命令行的文件
*/
#include <stdlib.h>
#include <stdio.h>

int
main( int ac, char **av )
{
        int  exit_status = EXIT_SUCCESS;
        FILE      *input;

        /*
        ** 当还有更多的文件名时...
        */
        while( *++av != NULL ){
        /*
        ** 试图打开这个文件。
        */
              input = fopen( *av, "r" );
              if( input == NULL ){
```

```
                perror( *av );
                exit_status = EXIT_FAILURE;
                continue;
        }

    /*
    ** 在这里处理这个文件...
    */

    /*
    ** 关闭文件（期望这里不会发生什么错误）。
    */
    if( fclose( input ) != 0 ){
                perror( "fclose" );
                exit( EXIT_FAILURE );
            }
        }

                return exit_status;
}
```

程序 15.1　打开和关闭文件　　open_cls.c

input 变量可能因为 fopen 和 fclose 之间的一个程序 bug 而发生修改。这个 bug 无疑将导致程序失败。在那些并不检查 fopen 函数的返回值的程序中，input 的值甚至有可能是 NULL。在任何一种情况下，fclose 都将会失败，而且程序很可能在 fclose 被调用之前很早便已终止。

那么你是否应该对 fclose（或任何其他操作）进行错误检查呢？在你作出决定之前，首先问自己两个问题。

1．如果操作成功应该执行什么？

2．如果操作失败应该执行什么？

如果这两个问题的答案是不同的，那么你应该进行错误检查。只有当这两个问题的答案是相同时，跳过错误检查才是合理的。

15.8　字符 I/O

当一个流被打开之后，它可以用于输入和输出。它最简单的形式是字符 I/O。字符输入是由 getchar 函数家族执行的，它们的原型如下所示。

```
int fgetc( FILE *stream );
int getc( FILE *stream );
int getchar( void );
```

需要操作的流作为参数传递给 getc 和 fgetc，但 getchar 始终从标准输入读取。每个函数从流中读取下一个字符，并把它作为函数的返回值返回。如果流中不存在更多的字符，函数就返回常量值 EOF。

这些函数都用于读取字符，但它们都返回一个 int 型值而不是 char 型值。尽管表示字符的代码本身是小整型，但返回 int 型值的真正原因是为了允许函数报告文件的末尾（EOF）。如果返回值是 char 型，那么在 256 个字符中必须有一个被指定用于表示 EOF。如果这个字符出现在文件内部，那么这个字符以后的内容将不会被读取，因为它被解释为 EOF 标志。

让函数返回一个 int 型值就能解决这个问题。EOF 被定义为一个整型，它的值在任何可能出现的字符范围之外。这种解决方法允许我们使用这些函数来读取二进制文件。在二进制文件中，所有的字符都有可能出现，文本文件也是如此。

为了把单个字符写入到流中，你可以使用 putchar 函数家族。它们的原型如下：

```
int fputc( int character, FILE *stream );
int putc( int character, FILE *stream );
int putchar( int character );
```

第 1 个参数是要被打印的字符。在打印之前，函数把这个整型参数裁剪为一个无符号字符型值，所以

```
putchar('abc' );
```

只打印一个字符（至于是哪一个，不同的编译器可能不同）。

如果由于任何原因（如写入到一个已被关闭的流）导致函数失败，它们就返回 EOF。

15.8.1　字符 I/O 宏

fgetc 和 fputc 都是真正的函数，但 getc、putc、getchar 和 putchar 都是通过#define 指令定义的宏。宏在执行时间上效率稍高，而函数在程序的长度方面更胜一筹。之所以提供两种类型的方法，是为了允许你根据程序的长度和执行速度哪个更重要选择正确的方法。这个区别实际上不必太看重，通过对实际程序的观察，不论采用何种类型，其结果通常相差甚微。

15.8.2　撤销字符 I/O

在你实际读取之前，你并不知道流的下一个字符是什么。因此，偶尔你所读取的字符是自己想要读取的字符的后面一个字符。例如，假定你必须从一个流中逐个读入一串数字。由于在实际读入之前，你无法知道下一个字符，你必须连续读取，直到读入一个非数字字符。但是如果你不希望丢弃这个字符，那么你该如何处置它呢？

ungetc 函数就是为了解决这种类型的问题。下面是它的原型。

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

ungetc 把一个先前读入的字符返回到流中，这样它可以在以后被重新读入。程序 15.2 说明了 ungetc 的用法。它从标准输入读取字符并把它们转换为一个整数。如果没有 ungetc，这个函数将不得不把这个多余的字符返回给调用程序，后者负责把它发送到读取下一个字符的程序部分。处理这个额外字符所涉及的特殊情况和额外逻辑使程序的复杂性显著提高。

```
/*
** 把一串从标准输入读取的数字转换为整数。
*/

#include <stdio.h>
#include <ctype.h>

int
read_int()
{
        int  value;
```

```
int    ch;

value = 0;

/*
** 转换从标准输入读入的数字，当我们得到一个非数字字符时就停止。
*/
while( ( ch = getchar() ) != EOF && isdigit( ch ) ){
       value *= 10;
       value += ch - '0';
}

/*
** 把非数字字符退回到流中，这样它就不会丢失。
*/
ungetc( ch, stdin );
return value;
}
```

程序 15.2　把字符转换为整数　　char_int.c

每个流都允许至少一个字符被退回。如果一个流允许退回多个字符，那么这些字符再次被读取的顺序就以退回时的反序进行。注意把字符退回到流中和写入到流中并不相同。与一个流相关联的外部存储并不受 ungetc 的影响。

警告：

“退回”字符和流的当前位置有关，所以如果用 fseek、fsetpos 或 rewind 函数改变了流的位置，所有退回的字符都将被丢弃。

15.9　未格式化的行 I/O

行 I/O 可以用两种方式执行——未格式化的或格式化的。这两种形式都用于操纵字符串。区别在于未格式化的 I/O（unformatted line I/O）简单读取或写入字符串，而格式化的 I/O 则执行数字和其他变量的内部和外部表示形式之间的转换。在本节，我们将讨论未格式化的行 I/O。

gets 和 puts 函数家族是用于操作字符串而不是单个字符。这个特征使它们在那些处理一行行文本输入的程序中非常有用。这些函数的原型如下所示。

```
char *fgets( char *buffer, int buffer_size, FILE *stream );
char *gets( char *buffer );

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

fgets 从指定的 stream 读取字符并把它们复制到 buffer 中。当它读取一个换行符并存储到缓冲区之后就不再读取。如果缓冲区内存储的字符数达到 buffer_size-1 个时它也停止读取。在这种情况下，并不会出现数据丢失的情况，因为下一次调用 fgets 将从流的下一个字符开始读取。在任何一种情况下，一个 NUL 字节将被添加到缓冲区所存储数据的末尾，使它成为一个字符串。

如果在任何字符读取前就到达了文件尾，缓冲区就未进行修改，fgets 函数返回一个 NULL 指针。否则，fgets 返回它的第 1 个参数（指向缓冲区的指针）。这个返回值通常只用于检查是否到达了文件尾。

传递给 fputs 的缓冲区必须包含一个字符串，它的字符被写入到流中。这个字符串预期以 NUL 字节结尾，所以这个函数没有一个缓冲区长度参数。这个字符串是逐字写入的：如果它不包含一个换行符，就不会写入换行符。如果它包含了好几个换行符，所有的换行符都会被写入。因此，当 fgets 每次都读取一整行时，fputs 却既可以一次写入一行的一部分，也可以一次写入一整行，甚至可以一次写入好几行。如果写入时出现了错误，fputs 返回常量值 EOF，否则它将返回一个非负值。

程序 15.3 是一个函数，它从一个文件读取输入行并原封不动地把它们写入到另一个文件。常量 MAX_LINE_LENGTH 决定缓冲区的长度，也就是可以被读取的一行文本的最大长度。在这个函数中，这个值并不重要，因为不管长行是被一次性读取还是分段读取，它所产生的结果文件都是相同的。另一方面，如果函数需要计数被复制的行的数目，太小的缓冲区将产生一个不正确的计数，因为一个长行可能会被分成数段进行读取。我们可以通过增加代码，观察每段是否以换行符结尾来修正这个问题。

缓冲区长度的正确值通常是根据需要执行的处理过程的本质而作出的折衷。但是，即使溢出它的缓冲区，fgets 也绝不引起错误。

警告：

注意 fgets 无法把字符串读入到一个长度小于两个字符的缓冲区，因为其中一个字符需要为 NUL 字节保留。

gets 和 puts 函数几乎和 fgets 与 fputs 相同。之所以存在它们是为了允许向后兼容。它们之间的一个主要的功能性区别在于当 gets 读取一行输入时，它并不在缓冲区中存储结尾的换行符。当 puts 写入一个字符串时，它在字符串写入之后向输出再添加一个换行符。

警告：

另一个区别仅存在于 gets，这从函数的原型中就清晰可见：它没有缓冲区长度参数。因此 gets 无法判断缓冲区的长度。如果一个长输入行读到一个短的缓冲区，多出来的字符将被写入到缓冲区后面的内存位置，这将破坏一个或多个不相关变量的值。这个事实导致 gets 函数只适用于玩具程序，因为唯一防止输入缓冲区溢出的方法就是声明一个巨大的缓冲区。但不管它有多大，下一个输入行仍有可能比缓冲区更大，尤其是当标准输入被重定向到一个文件时。

```
/*
** 把标准输入读取的文本行逐行复制到标准输出。
*/
#include <stdio.h>

#define   MAX_LINE_LENGTH          1024 /* 我可以复制的最长行 */

void
copylines( FILE *input, FILE *output )
{
          char  buffer[MAX_LINE_LENGTH];

          while( fgets( buffer, MAX_LINE_LENGTH, input ) != NULL )
                fputs( buffer, output );
}
```

程序 15.3　从一个文件向另一个文件复制文本行　　copyline.c

15.10 格式化的行 I/O

“格式化的行 I/O”这个名字从某种意义上说并不准确，因为 scanf 和 printf 函数家族并不仅限于单行。它们也可以在行的一部分或多行上执行 I/O 操作。

15.10.1 scanf 家族

scanf 函数家族的原型如下所示。每个原型中的省略号表示一个可变长度的指针列表。从输入转换而来的值逐个存储到这些指针参数所指向的内存位置。

```
int fscanf( FILE *stream, char const *format, ... );
int scanf( char const *format, ... );
int sscanf( char const *string, char const *format, ... );
```

这些函数都从输入源读取字符并根据 format 字符串给出的格式代码对它们进行转换。fscanf 的输入源就是作为参数给出的流，scanf 从标准输入读取，而 sscanf 则从第 1 个参数所给出的字符串中读取字符。

当格式化字符串到达末尾或者读取的输入不再匹配格式字符串所指定的类型时，输入就停止。在任何一种情况下，被转换的输入值的数目作为函数的返回值返回。如果在任何输入值被转换之前文件就已到达尾部，函数就返回常量值 EOF。

警告：

为了能让这些函数正常运行，指针参数的类型必须是对应格式代码的正确类型。函数无法验证它们的指针参数是否为正确的类型，所以函数就假定它们是正确的，于是继续执行并使用它们。如果指针参数的类型是不正确的，那么结果值就会是垃圾，而邻近的变量有可能在处理过程中被改写。

警告：

现在，对于 scanf 函数的参数前面为什么要加一个&符号应该是比较清楚的了。由于 C 的传值参数传递机制，把一个内存位置作为参数传递给函数的唯一方法就是传递一个指向该位置的指针。在使用 scanf 函数时，一个非常容易出现的错误就是忘了加上&符号。省略这个符号将导致变量的值作为参数传递给函数，而 scanf 函数（或其他两个）却把它解释为一个指针。当它被解引用时，或者导致程序终止，或者导致一个不可预料的内存位置的数据被改写。

15.10.2 scanf 格式代码

scanf 函数家族中的 format 字符串参数可能包含下列内容：

- 空白字符——它们与输入中的零个或多个空白字符匹配，在处理过程中将被忽略。
- 格式代码——它们指定函数如何解释接下来的输入字符。
- 其他字符——当任何其他字符出现在格式字符串时，下一个输入字符必须与它匹配。如果匹配，该输入字符随后就被丢弃。如果不匹配，函数就不再读取直接返回。

scanf 函数家族的格式代码都以一个百分号开头，后面可以是(1)一个可选的星号，(2)一个可选的宽度，(3)一个可选的限定符，(4)格式代码。星号将使转换后的值被丢弃而不是进行存储。这个技巧可以用于跳过不需要的输入字符。宽度以一个非负的整数给出，它限制将被读取用于转换的输入字符的个数。如果未给出宽度，函数就连续读入字符直到遇见输入中的下一个空白字符。限定符用

于修改有些格式代码的含义，它们在表 15.3 中列出。

表 15.3　　scanf 限定符

使用限定符的结果

格式码	h	l	L
d, i, n	short	long	
o, u, x	unsigned short	unsigned long	
e, f, g		double	long double

警告：

限定符的目的是为了指定参数的长度。如果整型参数比缺省的整型值更短或更长时，在格式代码中省略限定符就是一个常见的错误。对于浮点类型也是如此。如果省略了限定符，可能会导致一个较长变量只有部分被初始化，或者一个较短变量的邻近变量也被修改，这些都取决于这些类型的相对长度。

提示：

在一个缺省的整型长度和 short 相同的机器上，在转换一个 short 值时限定符 h 并非必需。但是，对于那些缺省的整型长度比 short 长的机器上，这个限定符是必需的。因此，如果你在转换所有的 short 和 long 型整数值和 long double 型变量时都使用适当的限定符，你的程序将更具可移植性。

格式代码就是一个单字符，用于指定输入字符如何被解释。表 15.4 描述了这些代码。

让我们来看一些使用 scanf 函数家族的例子。同样，我只显示与这些函数有关的部分代码。我们的第 1 个例子非常简单明了。它从输入流成对地读取数字并对它们进行一些处理。当读取到文件末尾时，循环就终止。

```
int     a, b;

while( fscanf( input, "%d %d", &a, &b ) == 2 ){
        /*
        ** Process the values a and b.
        */
}
```

这段代码并不精致，因为从流中输入的任何非法字符都将导致循环终止。同样，由于 fscanf 跳过空白字符，所以它没有办法验证这两个值是位于同一行还是分属两个不同的输入行。解决这个问题可以使用一种技巧，它将在后面的例子中说明。

下一个例子使用了字段宽度。

```
nfields = fscanf( input, "%4d %4d %4d", &a, &b, &c )
```

这个宽度参数把整数值的宽度限制为 4 个数字或者更少。使用下面的输入，

```
1 2
```

a 的值将是 1，b 的值将是 2，c 的值没有改变，nfields 的值将是 2。但是，如果使用下面的输入，

```
12345 67890
```

a 的值将是 1234，b 的值是 5，c 的值是 6789，而 nfields 的值是 3。输入中的最后一个 0 将保持在未输入状态。

在使用 fscanf 时，在输入中保持行边界的同步是很困难的，因为它把换行符也当作空白字符跳

过。例如，假定有一个程序读取的输入是由 4 个值所组成的一组值。这些值然后通过某种方式进行处理，然后再读取接下来的 4 个值。在这类程序中准备输入的最简单方法是把每组的 4 个值放在一个单独的输入行，这就很容易观察哪些值形成一组。但如果某个行包含了太多或太少的值，程序就会产生混淆。例如，考虑下面这个输入，它的第 2 行包含了一个错误：

```
1 1 1 1
2 2 2 2 2
3 3 3 3
4 4 4 4
5 5 5 5
```

如果我们使用 fscanf 按照一次读取 4 个值的方式读取这些数据时，头两组数据是正确的，但第 3 组读取的数据将是 2, 3, 3, 3，接下来的各组数据也都将不正确。

表 15.4　　scanf 格式码

代　码	参　数	含　义
c	char *	读取和存储单个字符。前导的空白字符并不跳过。如果给出宽度，就读取和存储这个数目的字符。字符后面不会添加一个 NUL 字节。参数必须指向一个足够大的字符数组
i d	int *	一个可选的有符号整数被转换。d 把输入解释为十进制数；i 根据它的第 1 个字符决定值的基数，就像整型字面值常量的表示形式一样
u o x	unsigned *	一个可选的有符号整数被转换，但它按照无符号数存储。如果使用 u，值被解释为十进制数；如果使用 o，值被解释为八进制数；如果使用 x，值被解释为十六进制数。X 和 x 同义
e f g	float *	期待一个浮点值。它的形式必须像一个浮点型字面值常量，但小数点并非必需。E 和 G 分别与 e 和 g 同义
s	char *	读取一串非空白字符。参数必须指向一个足够大的字符数组。当发现空白时输入就停止，字符串后面会自动加上 NUL 终止符
[xxx]	char *	根据给定组合的字符从输入中读取一串字符。参数必须指向一个足够大的字符数组。当遇到第 1 个不在给定组合中出现的字符时，输入就停止。字符串后面会自动加上 NUL 终止符。代码%[abc]表示字符组合包括 a、b 和 c。如果列表中以一个^字符开头，表示字符组合是所列出的字符的补集，所以%[^abc]表示字符组合为 a、b、c 之外的所有字符。右方括号也可以出现在字符列表中，但它必须是列表的第 1 个字符。至于横杠是否用于指定某个范围的字符（例如%[a-z]），则因编译器而异
p	void *	输入预期为一串字符，诸如那些由 printf 函数的%p 格式代码所产生的输出。它的转换方式因编译器而异，但转换结果将和按照上面描述的进行打印所产生的字符的值是相同的
n	int *	到目前为止通过这个 scanf 函数的调用从输入读取的字符数被返回。%n 转换的字符并不计算在 scanf 函数的返回值之内。它本身并不消耗任何输入
%	(无)	这个代码与输入中的一个%相匹配，该%符号将被丢弃

程序 15.4 使用一种更为可靠的方法读取这种类型的输入。这个方法的优点在于现在的输入是逐步处理的。它不可能读入一组起始于某一行但结束于另一行的值。而且，通过尝试转换 5 个值，无论是输入行的值太多还是太少都会被检测出来。

```
/*
** 用 sscanf 处理行定向(line-oriented)的输入
*/
#include <stdio.h>
#define    BUFFER_SIZE    100  /* 我们将要处理的最长行 */

void
function( FILE *input )
{
```

```
                int  a, b, c, d, e;
                char buffer[ BUFFER_SIZE ];

                while( fgets( buffer, BUFFER_SIZE, input ) != NULL ){
                        if( sscanf( buffer, "%d %d %d %d %d",
                            &a, &b, &c, &d, &e ) != 4 ){
                                fprintf( stderr, "Bad input skipped: %s",
                                   buffer );
                                continue;
                        }

                /*
                ** 处理这组输入
                */
                }
}
```

程序 15.4　用 sscanf 处理行定向的输入　　scanf1.c

一个相关的技巧用于读取可能以几种不同的格式出现的行定向输入。每个输入行先用 fgets 读取，然后用几个 sscanf（每个都使用一种不同的格式）进行扫描。输入行由**第 1 个** sscanf 决定，后者用于转换预期数目的值。例如，程序 15.5 检查一个以前读取的缓冲区的内容。它从一个输入行中提取或者 1 个或者 2 个或者 3 个值并对那些没有输入值的变量赋缺省的值。

```
/*
** 使用 sscanf 处理可变格式的输入
*/
#include <stdio.h>
#include <stdlib.h>

#define   DEFAULT_A 1     /* 或其他 ... */
#define   DEFAULT_B 2     /* 或其他 ... */

void
function( char *buffer )
{
        int   a, b, c;

/*
** 看看 3 个值是否都已给出。
*/
if( sscanf( buffer, "%d %d %d", &a, &b, &c ) != 3 ){
/*
** 否，对 a 使用缺省值，看看其他两个值是否都已给出。
*/
      a = DEFAULT_A;
      if( sscanf( buffer, "%d %d", &b, &c ) != 2 ){
      /*
      ** 也为 b 使用缺省值，寻找剩余的值。
      */
            b = DEFAULT_B;
            if( sscanf( buffer, "%d", &c ) != 1 ){
                  fprintf( stderr, "Bad input: %s",
                  buffer );
```

```
                        exit( EXIT_FAILURE );
                    }
                }
            }
            /*
            ** 处理 a, b, c
            */
}
```

程序 15.5　使用 sscanf 处理可变格式的输入　　scanf2.c

15.10.3　printf 家族

printf 函数家庭用于创建格式化的输出。这个家族共有 3 个函数：fprintf、printf 和 sprintf。它们的原型如下所示。

```
int fprintf( FILE *stream, char const *format, ... );
int printf( char const *format, ... );
int sprintf( char *buffer, char const *format, ... );
```

你在第 1 章就曾见过，printf 根据格式代码和 format 参数中的其他字符对参数列表中的值进行格式化。这个家族的另两个函数的工作过程也类似。使用 printf，结果输出送到标准输出。使用 fprintf，你可以使用任何输出流，而 sprintf 把它的结果作为一个 NUL 结尾的字符串存储到指定的 buffer 缓冲区而不是写入到流中。这 3 个函数的返回值是实际打印或存储的字符数。

警告：

sprintf 是一个潜在的错误根源。缓冲区的大小并不是 sprintf 函数的一个参数，所以如果输出结果很长溢出缓冲区时，就可能改写缓冲区后面内存位置中的数据。要杜绝这个问题，可以采取两种策略。第 1 种是声明一个非常巨大的缓冲区，但这个方案很浪费内存，而且尽管大型缓冲区能够减少溢出的可能性，但它并不能根除这种可能性。第 2 种方法是对格式进行分析，看看最大可能出现的值被转换后的结果输出将有多长。例如，在 4 位整型的机器上，最大的整数有 11 位（包括一个符号位），所以缓冲区至少能容纳 12 个字符（包括结尾的 NUL 字节）。字符串的长度并没有限制，但函数所生成的字符串的字符数目可以用格式代码中一个可选的字段来限制。

警告：

printf 函数家族的格式代码和 scanf 函数家族的格式代码用法不同。所以你必须小心谨慎，防止误用。两者的格式代码中的有些可选字段看上去是相同的，这使得问题变得更为困难。不幸的是，许多常见的格式代码，如%d 就属于这一类。

警告：

另一个错误来源是函数的参数类型与对应的格式代码不匹配。通常这个错误将导致输出结果是垃圾，但这种不匹配也可能导致程序失败。和 scanf 函数家族一样，这些函数无法验证一个值是否具有格式码所表示的正确类型，所以保证它们相互匹配是程序员的责任。

15.10.4　printf 格式代码

printf 函数原型中的 format 字符串可能包含格式代码，它使参数列表的下一个值根据指定的方

式进行格式化，至于其他的字符则原样逐字打印。格式代码由一个百分号开头，后面跟(1)零个或多个标志字符，用于修改有些转换的执行方式，(2)一个可选的最小字段宽度，(3)一个可选的精度，(4)一个可选的修改符，(5)转换类型。

标志和其他字段的准确含义取决于使用何种转换。表 15.5 描述了转换类型代码，表 15.6 描述了标志字符和它们的含义。

表 15.5　　printf 格式代码

代　码	参　　数	含　　义
c	int	参数被裁剪为 unsigned char 类型并作为字符进行打印
d i	int	参数作为一个十进制整数打印。如果给出了精度而且值的位数少于精度位度，前面就用 0 填充
u o x,X	unsigned int	参数作为一个无符号值打印，u 使用十进制，o 使用八进制，x 或 X 使用十六进制，两者的区别是 x 约定使用 abcdef，而 X 约定使用 ABCDEF
e E	double	参数根据指数形式打印。例如，6.023000e23 是使用代码 e，6.023000E23 是使用代码 E。小数点后面的位数由精度字段决定，缺省值是 6
f	double	参数按照常规的浮点格式打印。精度字段决定小数点后面的位数，缺省值是 6
g G	double	参数以%f 或%e（如 G 则%E）的格式打印，取决于它的值。如果指数大于等于-4 但小于精度字段就使用%f 格式，否则使用指数格式
s	char *	打印一个字符串
p	void *	指针值被转换为一串因编译器而异的可打印字符。这个代码主要是和 scanf 中的%p 代码组合使用
n	int *	这个代码是独特的，因为它并不产生任何输出。相反，到目前为止函数所产生的输出字符数目将被保存到对应的参数中
%	(无)	打印一个%字符

表 15.6　　printf 格式标志

标　志	含　　义
-	值在字段中左对齐，缺省情况下是右对齐
0	当数值为右对齐时，缺省情况下是使用空格填充值左边未使用的列。这个标志表示用零来填充，它可用于 d,i,u,o,x,X,e,E,f,g 和 G 代码。使用 d,i,u,o,x 和 X 代码时，如果给出了精度字段，零标志就被忽略。如果格式代码中出现了负号标志，零标志也没有效果
+	当用于一个格式化某个有符号值的代码时，如果值非负，正号标志就会给它加上一个正号。如果该值为负，就像往常一样显示一个负号。在缺省情况下，正号并不会显示
空格	只用于转换有符号值的代码。当值非负时，这个标志把一个空格添加到它的开始位置。注意这个标志和正号标志是相互排斥的，如果两个同时给出，空格标志便被忽略
#	选择某些代码的另一种转换形式。它们在表 15.8 中描述

字段宽度是一个十进制整数，用于指定将出现在结果中的最小字符数。如果值的字符数少于字段宽度，就对它进行填充以增加长度。标志决定填充是用空白还是零以及它出现在值的左边还是右边。

对于 d、i、u、o、x 和 X 类型的转换，精度字段指定将出现在结果中的最小的数字个数并覆盖零标志。如果转换后的值的位数小于宽度，就在它的前面插入零。如果值为零且精度也为零，则转换结果就不会产生数字。对于 e、E 和 f 类型的转换，精度决定将出现在小数点之后的数字位数。对于 g 和 G 类型的转换，它指定将出现在结果中的最大有效位数。当使用 s 类型的转换时，精度指定将被转换的最多字符数。精度以一个句点开头，后面跟一个可选的十进制整数。如果未

给出整数，精度的缺省值为零。

如果用于表示字段宽度和/或精度的十进制整数由一个星号代替，那么 printf 的下一个参数（必须是个整数）就提供宽度和（或）精度。因此，这些值可以通过计算获得而不必预先指定。

当字符或短整数值作为 printf 函数的参数时，它们在传递给函数之前先转换为整数。有时候转换可以影响函数产生的输出。同样，在一个长整数的长度大于普通整数的环境里，当一个长整数作为参数传递给函数时，printf 必须知道这个参数是个长整数。表 15.7 所示的修改符用于指定整数和浮点数参数的准确长度，从而解决了这个问题。

表 15.7　　printf 格式代码修改符

修　改　符	用于...时	表示参数是...
h	d, i, u, o, x, X	一个（可能是无符号）short 型整数
h	n	一个指向 short 型整数的指针
l	d, i, u, o, x, X	一个（可能是无符号）long 型整数
l	n	一个指向 long 型整数的指针
L	e, E, f, g, G	一个 long double 型值

在有些环境里，int 和 short int 的长度相等，此时 h 修改符就没有效果。否则，当 short int 作为参数传递给函数时，这个被转换的值将升级为（无符号）int 类型。这个修改符在转换发生之前使它被裁剪回原先的 short 形式。在十进制转换中，一般并不需要进行剪裁。但在有些八进制或十六进制的转换中，h 修改符将保证适当位数的数字被打印。

警告：

在 int 和 long int 长度相同的机器上，l 修改符并无效果。在所有其他机器上，需要使用 l 修改符，因为这些机器上的长整型分为两部分传递到运行时堆栈。如果这个修改符并未给出，那就只有第 1 部分被提取用于转换。这样，不仅转换将产生不正确的结果，而且这个值的第 2 部分被解释为一个单独的参数，这样就破坏了后续参数和它们的格式代码之间的对应关系。

#标志可以用于几种 printf 格式代码，为转换选择一种替代形式。这些形式的细节列于表 15.8。

表 15.8　　printf 转换的其他形式

用于...	#标志...
o	保证产生的值以一个零开头
x, X	在非零值前面加 0x 前缀（%X 则为 0X）
e, E, f	确保结果始终包含一个小数点，即使它后面没有数字
g, G	和上面的 e,E 和 f 代码相同。另外，缀尾的 0 并不从小数中去除

提示：

由于有些机器在打印长整数值时要求 l 修改符而另外一些机器可能不需要。所以，当你打印长整数值时，最好坚持使用 l 修改符。这样，当你把程序移植到任何一台机器上时，就不太需要进行改动。

printf 函数可以使用丰富的格式代码、修改符、限定符、替代形式和可选字段，这使得它看上去极为复杂。但是，它们能够在格式化输出时提供极大的灵活性。所以，你应该耐心一些，把它们全部学会要花一些时间！这里有一些例子，帮助你学习它们。

图 15.1 显示了格式化字符串可能产生的一些变型。只有显示出来的字符才被打印。为了避免歧义，符号¤用于表示一个空白。图 15.2 显示了用不同的整数格式代码格式化一些整数值的结果。图 15.3 显示了浮点值被格式化的一些可能方法。最后，图 15.4 显示了用与前图相同的那些格式代码来格式化一个非常大的浮点数的结果。在前两个输出中出现了明显的错误，因为它们所打印的有效数字的位数超出了指定内存位置所能存储的位数。

格式代码	转换后的字符串		
	A	ABC	ABCDEFGH
%s	A	ABC	ABCDEFGH
%5s	¤¤¤¤A	¤¤ABC	ABCDEFGH
%.5s	A	ABC	ABCDE
%5.5s	¤¤¤¤A	¤¤ABC	ABCDE
%-5s	A¤¤¤¤	ABC¤¤	ABCDEFGH

图 15.1　用 printf 格式字符串

格式代码	转换后的数值			
	1	-12	12345	123456789
%d	1	-12	12345	123456789
%6d	¤¤¤¤¤1	¤¤¤-12	¤12345	123456789
%.4d	0001	-0012	12345	123456789
%6.4d	¤¤0001	¤-0012	¤12345	123456789
%-4d	1¤¤¤	-12¤	12345	123456789
%04d	0001	-012	12345	123456789
%+d	+1	-12	+12345	+123456789

图 15.2　用 printf 格式化整数

格式代码	转换后的数值			
	1	.01	.00012345	12345.6789
%f	1.000000	0.010000	0.000123	12345.678900
%10.2f	¤¤¤¤¤¤1.00	¤¤¤¤¤¤0.01	¤¤¤¤¤¤0.00	¤¤12345.68
%e	1.000000e+00	1.000000e-02	1.234500e-04	1.234568e+04
%.4e	1.0000e+00	1.0000e-02	1.2345e-04	1.2346e+04
%g	1	0.01	0.00012345	12345.7

图 15.3　用 printf 格式化浮点值

格式代码	转换后的数值
	6.023e23
%f	6022999999999999975882752.000000
%10.2f	6022999999999999975882752.00
%e	6.023000e+23
%.4e	6.0230e+23
%g	6.023e+23

图 15.4　用 printf 格式化大浮点值

15.11 二进制 I/O

把数据写到文件效率最高的方法是用二进制形式写入。二进制输出避免了在数值转换为字符串过程中所涉及的开销和精度损失。但二进制数据并非人眼所能阅读，所以这个技巧只有当数据将被另一个程序按顺序读取时才能使用。

fread 函数用于读取二进制数据，fwrite 函数用于写入二进制数据。它们的原型如下所示：

```
size_t  fread( void *buffer, size_t size, size_t count,  FILE *stream );
size_t  fwrite( void *buffer, size_t size, size_t count, FILE *stream );
```

buffer 是一个指向用于保存数据的内存位置的指针，size 是缓冲区中每个元素的字节数，count 是读取或写入的元素数，当然 stream 是数据读取或写入的流。

buffer 参数被解释为一个或多个值的数组。count 参数指定数组中有多少个值，所以读取或写入一个标量时，count 的值应为 1。函数的返回值是实际读取或写入的**元素**（并非字节）数目。如果输入过程中遇到了文件尾或者输出过程中出现了错误，这个数字可能比请求的元素数目要小。

让我们观察一个使用这些函数的代码段。

```
struct  VALUE    {
         long     a;
         float    b;
         char     c[SIZE];
} values[ARRAY_SIZE];
...
n_values = fread( values, sizeof( struct VALUE ),
    ARRAY_SIZE, input_stream );
(处理数组中的数据)
fwrite( values, sizeof( struct VALUE ),
    n_values, output_stream );
```

这个程序从一个输入文件读取二进制数据，对它执行某种类型的处理，把结果写入到一个输出文件。前面提到过，这种类型的 I/O 效率很高，因为每个值中的位直接从流读取或向流写入，不需要任何转换。例如，假定数组中的一个长整数的值是 4,023,817。代表这个值的位是 0x003d6609——这些位将被写入到流中。二进制信息非人眼所能阅读，因为这些位并不对应任何合理的字符。如果它们解释为字符，其值将是\0=f\t，这显然不能很好地向我们传达原数的值。

15.12 刷新和定位函数

在处理流时，另外还有一些函数也较为有用。首先是 fflush，它迫使一个输出流的缓冲区内的数据进行物理写入，不管它是不是已经写满。它的原型如下：

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

当我们需要立即把输出缓冲区的数据进行物理写入时，应该使用这个函数。例如，调用 fflush 函数保证调试信息实际打印出来，而不是保存在缓冲区中直到以后才打印。

在正常情况下，数据以线性的方式写入，这意味着后面写入的数据在文件中的位置是在以前所有写入数据的后面。C 同时支持随机访问 I/O，也就是以任意顺序访问文件的不同位置。随机访问是通过在读取或写入先前定位到文件中需要的位置来实现的。有两个函数用于执行这项操作，它们的原型如下：

```
long  ftell( FILE *stream );
int  fseek( FILE *stream, long offset, int from );
```

ftell 函数返回流的当前位置，也就是说，下一个读取或写入将要开始的位置距离文件起始位置的偏移量。这个函数允许你保存一个文件的当前位置，这样你可能在将来会返回到这个位置。在二进制流中，这个值就是当前位置距离文件起始位置之间的字节数。

在文本流中，这个值表示一个位置，但它并不一定准确地表示当前位置和文件起始位置之间的字符数，因为有些系统将对行末字符进行翻译转换。但是，ftell 函数返回的值总是可以用于 fseek 函数中，作为一个距离文件起始位置的偏移量。

fseek 函数允许你在一个流中定位。这个操作将改变下一个读取或写入操作的位置。它的第 1 个参数是需要改变的流。它的第 2 和第 3 个参数标识文件中需要定位的位置。表 15.9 描述了三种第 2 个和第 3 个参数可以使用的方法。

试图定位到一个文件的起始位置之前是一个错误。定位到文件尾之后并进行写入将扩展这个文件。定位到文件尾之后并进行读取将导致返回一条“到达文件尾”的信息。在二进制流中，从 SEEK_END 进行定位可能不被支持，所以应该避免。在文本流中，如果 from 是 SEEK_CUR 或 SEEK_END，offset 必须是零。如果 from 是 SEEK_SET，offset 必须是一个从同一个流中以前调用 ftell 所返回的值。

表 15.9　　fseek 参数

如果 from 是...	你将定位到...
SEEK_SET	从流的起始位置起 offset 个字节，offset 必须是一个非负值
SEEK_CUR	从流的当前位置起 offset 个字节，offset 的值可正可负
SEEK_END	从流的尾部位置起 offset 个字节，offset 的值可正可负。如果它是正值，它将定位到文件尾的后面

之所以存在这些限制，部分原因是文本流所执行的行末字符映射。由于这种映射的存在，文本文件的字节数可能和程序写入的字节数不同。因此，一个可移植的程序不能根据实际写入字符数的计算结果定位到文本流的一个位置。

用 fseek 改变一个流的位置会带来三个副作用。首先，行末指示字符被清除。其次，如果在 fseek 之前使用 ungetc 把一个字符返回到流中，那么这个被退回的字符会被丢弃，因为在定位操作以后，它不再是“下一个字符”。最后，定位允许你从写入模式切换到读取模式，或者回到打开的流以便更新。

程序 15.6 使用 fseek 访问一个学生信息文件。记录数参数的类型是 size_t，这是因为它不可能是个负值。需要定位的文件位置通过将记录数和记录长度相乘得到。只有当文件中的所有记录都是同一长度时，这种计算方法才是可行的。最后，fread 的结果被返回，这样调用程序就可以判断操作是否成功。

另外还有三个额外的函数，用一些限制更严的方式执行相同的任务。它们的原型如下：

```
void  rewind( FILE *stream );
int   fgetpos( FILE *stream, fpos_t *position );
int   fsetpos( FILE *stream, fpos_t const *position );
```

rewind 函数将读/写指针设置回指定流的起始位置。它同时清除流的错误提示标志。fgetpos 和 fsetpos 函数分别是 ftell 和 fseek 函数的替代方案。

它们的主要区别在于这对函数接受一个指向 fpos_t 的指针作为参数。fgetpos 在这个位置存储文件的当前位置，fsetpos 把文件位置设置为存储在这个位置的值。

用 fpos_t 表示一个文件位置的方式并不是由标准定义的。它可能是文件中的一个字节偏移量，也可能不是。因此，使用一个从 fgetpos 函数返回的 fpos_t 类型的值唯一安全的用法是把它作为参数传递给后续的 fsetpos 函数。

```
/*
** 从一个文件读取一个特定的记录。参数分别是进行读取的流、需要读取的记录数和** 指向放置数据的缓冲区的指针。
*/
#include <stdio.h>
#include "student_info.h"

int
read_random_record( FILE *f, size_t rec_number, StudentInfo *buffer )
{
        fseek( f, (long)rec_number * sizeof( StudentInfo ),
                SEEK_SET );
        return fread( buffer, sizeof( StudentInfo ), 1, f );
}
```

程序 15.6　随机文件访问　　rd_rand.c

15.13　改变缓冲方式

在流上执行的缓冲方式有时并不合适，下面两个函数可以用于对缓冲方式进行修改。这两个函数只有当指定的流被打开但还没有在它上面执行任何其他操作前才能被调用。

```
void setbuf( FILE *stream, char *buf );
int  setvbuf( FILE *stream, char *buf, int mode, size_t size );
```

setbuf 设置了另一个数组，用于对流进行缓冲。这个数组的字符长度必须为 BUFSIZ（它在 stdio.h 中定义）。为一个流自行指定缓冲区可以防止 I/O 函数库为它动态分配一个缓冲区。如果用一个 NULL 参数调用这个函数，setbuf 函数将关闭流的所有缓冲方式。字符准确地将程序所规引的方式进行读取和写入[1]。

警告：

为流缓冲区使用一个自动数组是很危险的。如果在流关闭之前，程序的执行流离开了数组声明所在的代码块，流就会继续使用这块内存，但此时它可能已经分配给了其他函数另作它用。

setvbuf 函数更为通用。mode 参数用于指定缓冲的类型。_IOFBF 指定一个完全缓冲的流，_IONBF 指定一个不缓冲的流，_IOLBF 指定一个行缓冲流。所谓行缓冲，就是每当一个换行符写入到缓冲区时，缓冲区便进行刷新。

buf 和 size 参数用于指定需要使用的缓冲区。如果 buf 为 NULL，那么 size 的值必须是 0。一般而言，最好用一个长度为 BUFSIZ 的字符数组作为缓冲区。尽管使用一个非常大的缓冲区可能可以稍稍提高程序的效率，但如果使用不当，它也有可能降低程序的效率。例如，绝大多数操作系统在内部对磁盘的输入/输出进行缓冲操作。如果你自行指定了一个缓冲区，但它的长度却不是操作系统内部使用的缓冲区的整数倍，就可能需要一些额外的磁盘操作，用于读取或写入一个内存块的一部

[1] 在宿主式运行时环境中，操作系统可能执行自己的缓冲方式，不依赖于流。因此，仅仅调用 setbuf 将不允许程序从键盘即输即读入字符，因为操作系统通常对这些字符进行缓冲，用于实现退格编辑。

分。如果你需要使用一个很大的缓冲区，它的长度应该是 BUFSIZ 的整数倍。在 MS-DOS 机器中，缓冲区的大小如果和磁盘簇的大小相匹配，可能会提高一些效率。

15.14　流错误函数

下面的函数用于判断流的状态。

```
int  feof( FILE *stream );
int  ferror( FILE *stream );
void clearerr( FILE *stream );
```

如果流当前处于文件尾，feof 函数返回真。这个状态可以通过对流执行 fseek、rewind 或 fsetpos 函数来清除。ferror 函数报告流的错误状态，如果出现任何读/写错误函数就返回真。最后，clearerr 函数对指定流的错误标志进行重置。

15.15　临时文件

偶尔，为了方便起见，我们会使用一个文件来临时保存数据。当程序结束时，这个文件便被删除，因为它所包含的数据不再有用。tmpfile 函数就是用于这个目的的。

```
FILE *tmpfile( void );
```

这个函数创建了一个文件，当文件被关闭或程序终止时这个文件便自动删除。该文件以 wb+模式打开，这使它可用于二进制和文本数据。

如果临时文件必须以其他模式打开或者由一个程序打开但由另一个程序读取，就不适合用 tmpfile 函数创建。在这些情况下，我们必须使用 fopen 函数，而且当结果文件不再需要时必须使用 remove 函数（稍后描述）显式地删除。

临时文件的名字可以用 tmpnam 函数创建，它的原型如下：

```
char  *tmpnam( char *name );
```

如果传递给函数的参数为 NULL，那么这个函数便返回一个指向静态数组的指针，该数组包含了被创建的文件名。否则，参数便假定是一个指向长度至少为 L_tmpnam 的字符数组的指针。在这种情况下，文件名在这个数组中创建，返回值就是这个参数。

无论哪种情况，这个被创建的文件名保证不会与已经存在的文件名同名[1]。只要调用次数不超过 TMP_MAX 次，tmpnam 函数每次调用时都能产生一个新的不同名字。

15.16　文件操纵函数

有两个函数用于操纵文件但不执行任何输入/输出操作。它们的原型如下所示。如果执行成功，这两个函数都返回零值。如果失败，它们都返回非零值。

---

[1] 注意：这个用于保证唯一性的方法可能会在多程序系统(multiprogramming system)或那些其享一个网络文件服务器的系统中失败。问题的根源是名字被创建和该名字所标识的文件被创建之间的时间延迟。如果几个程序恰好都创建了一个相同的名字，并在任何文件被实际创建之前测试是否存在这个名字的文件，此时测试结果是否定的，于是每个程序都以为这是个唯一的名字。在文件名被创建之后立即创建文件可以减少（但不能根除）这种潜在的冲突。

```
int  remove( char const *filename );
int  rename( char const *oldname, char const *newname );
```

remove 函数删除一个指定的文件。如果当 remove 被调用时文件处于打开状态，其结果则取决于编译器。

rename 函数用于改变一个文件的名字，从 oldname 改为 newname。如果已经有一个名为 newname 的文件存在，其结果取决于编译器。如果这个函数失败，文件仍然可以用原来的名字进行访问。

15.17 总结

标准规定了标准函数库中的函数的接口和操作，这有助于提高程序的可移植性。一种编译器可以在它的函数库中提供额外的函数，但不应修改标准要求提供的函数。

perror 函数提供了一种向用户报告错误的简单方法。当检测到一个致命的错误时，你可以使用 exit 函数终止程序。

stdio.h 头文件包含了使用 I/O 库函数所需要的声明。所有的 I/O 操作都是一种在程序中移进或移出字节的事务。函数库为 I/O 所提供的接口称为流。在缺省情况下，流 I/O 是进行缓冲的。二进制流主要用于二进制数据，字节不经修改地从二进制流读取或向二进制流写入。另一方面，文本流则用于字符。文本流能够允许的最大文本行因编译器而异，但至少允许 254 个字符。根据定义，行由一个换行符结尾。如果宿主操作系统使用不同的约定结束文本行，I/O 函数必须在这种形式和文本行的内部形式之间进行翻译转换。

FILE 是一种数据结构，用于管理缓冲区和存储流的 I/O 状态。运行时环境为每个程序提供了三个流——标准输入、标准输出和标准错误。最常见的情况是把标准输入缺省设置为键盘，其他两个流缺省设置为显示器。错误信息使用一个单独的流，这样即使标准输出的缺省值重定向为其他位置，错误信息仍能够显示在它的缺省位置。FOPEN_MAX 是你能够同时打开的最多文件数，具体数目因编译器而异，但不能小于 8。FILENAME_MAX 是用于存储文件名的字符数组的最大限制长度。如果不存在长度限制，这个值就是推荐最大长度。

为了对一个文件执行流 I/O 操作，首先必须用 fopen 函数打开文件，它返回一个指向 FILE 结构的指针，这个 FILE 结构指派给进行操作的流。这个指针必须在一个 FILE *类型的变量中保存。然后，这个文件就可以进行读取和（或）写入。读写完毕后，应该关闭文件。许多 I/O 函数属于同一个家族，它们在本质上执行相同的任务，但在从何处读取或何处写入方面存在一些微小的差别。通常一个函数家族的各个变型包括接受一个流参数的函数，一个只用于标准流之一的函数以及一个使用内存中的缓冲区而不是流的函数。

流用 fopen 函数打开。它的参数是需要打开的文件名和需要采用的流模式。模式指定流用于读取、写入还是添加，它同时指定流为二进制流还是文本流。freopen 函数用于执行相同的任务，但你可以自己指定需要使用的流。这个函数最常用于重新打开一个标准流。你应该始终检查 fopen 或 freopen 函数的返回值，看看有没有发生错误。在结束了一个流的操作之后，你应该使用 fclose 函数将它关闭。

逐字符的 I/O 由 getchar 和 putchar 函数家族实现。输入函数 fgetc 和 getc 都接受一个流参数，getchar 则只从标准输入读取。第 1 个以函数的方式实现，后两个则以宏的方式实现。它们都返回一个用整型值表示的单字符。除了用于执行输出而不是输入之外，fputc、putc 和 putchar 函数具有和对应的输入函数相同的属性。ungetc 用于把一个不需要的字符退回到流中。这个被退回的字符将是

下一个输入操作所返回的第 1 个字符。改变流的位置（定位）将导致这个退回的字符被丢弃。

行 I/O 既可以是格式化的，也可以是未格式化的。gets 和 puts 函数家族执行未格式化的行 I/O。fgets 和 gets 都从一个指定的缓冲区读取一行。前者接受一个流参数，后者从标准输入读取。fgets 函数更为安全，它把缓冲区长度作为参数之一，因此可以保证一个长输入行不会溢出缓冲区。而且数据并不会丢失——长输入行的剩余部分（超出缓冲区长度的那部分）将被 fgets 函数的下一次调用读取。fputs 和 puts 函数把文本写入到流中。它们的接口类似对应的输入函数。为了保证向后兼容，gets 函数将去除它所读取的行的换行符，puts 函数在写入到缓冲区的文本后面加上一个换行符。

scanf 和 printf 函数家族执行格式化的 I/O 操作。输入函数共有三种，fscanf 接受一个流参数，scanf 从标准输入读取，sscanf 从一个内存中的缓冲区接收字符。printf 家族也有三个函数，它们的属性也类似。scanf 家族的函数根据一个格式字符串对字符进行转换。一个指针参数列表用于提示结果值的存储地点。函数的返回值是被转换的值的个数，如果没有任何值被转换就遇到文件尾，函数就返回 EOF。printf 家族的函数根据一个格式字符串把值转换为字符形式。这些值是作为参数传递给函数的。

使用二进制流写入二进制数据（如整数和浮点数）比使用字符 I/O 效率更高。二进制 I/O 直接读写值的各个位，而不必把值转换为字符。但是，二进制输出的结果非人眼所能阅读。fread 和 fwrite 函数执行二进制 I/O 操作。每个函数都接受 4 个参数：指向缓冲区的指针、缓冲区中每个元素的长度、需要读取或写入的元素个数以及需要操作的流。

在缺省情况下，流是顺序读取的。但是，你可以通过在读取或写入之前定位到一个不同的位置实现随机 I/O 操作。fseek 函数允许你指定文件中的一个位置，它用一个偏移量表示，参考位置可以是文件起始位置，也可以是文件当前位置，还可以是文件的结尾位置。ftell 函数返回文件的当前位置。fsetpos 和 fgetpos 函数是前两个函数的替代方案。但是，fsetpos 函数的参数只有当它是先前从一个作用于同一个流的 fgetpos 函数的返回值时才是合法的。最后，rewind 函数返回到文件的起始位置。

在执行任何流操作之前，调用 setbuf 函数可以改变流所使用的缓冲区。用这种方式指定一个缓冲区可以防止系统为流动态分配一个缓冲区。向这个函数传递一个 NULL 指针作为缓冲区参数表示禁止使用缓冲区。setvbuf 函数更为通用。使用它，你可以指定一个并非标准长度的缓冲区。你也可以选择你所希望的缓冲方式：全缓冲、行缓冲或不缓冲。

ferror 和 clearerr 函数和流的错误状态有关，也就是说，是否出现了任何读/写错误。第 1 个函数返回错误状态，第 2 个函数重置错误状态。如果流当前位于文件的末尾，那么 feof 函数就返回真。

tmpfile 函数返回一个与一个临时文件关联的流。当流被关闭之后，这个文件被自动删除。tmpnam 函数为临时文件创建一个合适的文件名。这个名字不会与现存的文件名冲突。把文件名作为参数传递给 remove 函数可以删除这个文件。rename 函数用于修改一个文件的名字。它接受两个参数，文件的当前名字和文件的新名字。

15.18　警告的总结

1. 忘了在一条调试用的 printf 语句后面跟一个 fflush 调用。
2. 不检查 fopen 函数的返回值。
3. 改变文件的位置将丢弃任何被退回到流的字符。
4. 在使用 fgets 时指定太小的缓冲区。
5. 使用 gets 的输入溢出缓冲区且未被检测到。

6．使用任何 scanf 系列函数时，格式代码和参数指针类型不匹配。

7．在任何 scanf 系列函数的每个非数组、非指针参数前忘了加上&符号。

8．注意在使用 scanf 系列函数转换 double、long double、short 和 long 整型时，在格式代码中加上合适的限定符。

9．sprintf 函数的输出溢出了缓冲区且未检测到。

10．混淆 printf 和 scanf 格式代码。

11．使用任何 printf 系列函数时，格式代码和参数类型不匹配。

12．在有些长整数长于普通整数的机器上打印长整数值时，忘了在格式代码中指定 l 修改符。

13．使用自动数组作为流的缓冲区时应多加小心。

15.19 编程提示的总结

1．在可能出现错误的场合，检查并报告错误。

2．操纵文本行而无需顾及它们的外部表示形式这个能力有助于提高程序的可移植性。

3．使用 scanf 限定符提高可移植性。

4．当你打印长整数时，即使你所使用的机器并不需要，坚持使用 l 修改符可以提高可移植性。

15.20 问题

1．如果对 fopen 函数的返回值不进行错误检查可能会出现什么后果？

2．如果试图对一个从未打开过的流进行 I/O 操作会发生什么情况？

3．如果一个 fclose 调用失败，但程序并未对它的返回值进行错误检查可能会出现什么后果？

4．如果一个程序在执行时它的标准输入已重定向到一个文件，程序如何检测到这个情况？

5．如果调用 fgets 函数时使用一个长度为 1 的缓冲区会发生什么？长度为 2 呢？

6．为了保证下面这条 sprintf 语句所产生的字符串不溢出，缓冲区至少应该有多大？假定你的机器的上整数的长度为 2 个字节。

```
sprintf( buffer, "%d %c %x", a, b, c );
```

7．为了保证下面这条 sprintf 语句所产生的字符串不溢出，缓冲区至少应该有多大？

```
sprintf( buffer, "%s", a );
```

8．%f 格式代码所打印的最后一位数字是经过四舍五入呢？还是未打印的数字被简单地截掉？

9．你如何得到 perror 函数可能打印的所有错误信息列表？

10．为什么 fprintf、fscanf、fputs 和 fclose 函数都接受一个指向 FILE 结构的指针作为参数而不是 FILE 结构本身。

11．你希望打开一个文件进行写入，假定（1）你不希望文件原先的内容丢失，（2）你希望能够写入到文件的任何位置。那么你该怎样设置打开模式呢？

12．为什么需要 freopen 函数？

13. 对于绝大多数程序，你觉得有必要考虑 fgetc(stdin)或 getchar 哪个更好吗？

14. 在你的系统上，下面的语句将打印什么内容？

```
printf("%d\n", 3.14 );
```

15. 请解释使用%-6.10s 格式代码将打印出什么形式的字符串。

16. 当一个特定的值用格式代码%.3f 打印时，其结果是 1.405。但这个值用格式代码%.2f 打印时，其结果是 1.40。似乎出现了明显错误，请解释其原因。

15.21　编程练习

★1. 编写一个程序，把标准输入的字符逐个复制到标准输出。

★★2. 修改你对练习 1 的解决方案，使它每次读写一整行。你可以假定文件中每一行所包含的字符数不超过 80 个（不包括结尾的换行符）。

★★3. 修改你对练习 2 的解决方案，去除每行 80 个字符的限制。处理这个文件时，你仍应该每次处理一行，但对于那些长于 80 个字符的行，你可以每次处理其中的一段。

★★★4. 修改你对练习 3 的解决方案，提示用户输入两个文件名，并从标准输入读取它们。第 1 个作为输入文件，第 2 个作为输出文件。这个修改后的程序应该打开这两个文件并把输入文件的内容按照前面的方式复制到输出文件。

★★★5. 修改你对练习 4 的解决方案，使它寻找那些以一个整数开始的行。这些整数值应该进行求和，其结果应该写入到输出文件的末尾。除了这个修改之外，这个修改后的程序的其他部分应该和练习 4 一样。

★★6. 在第 9 章，你编写了一个称为 palindrome 的函数，用于判断一个字符串是否是一个回文。在这个练习中，你需要编写一个函数，判断一个整型变量的值是不是回文。例如，245 不是回文，但 14741 却是回文。这个函数的原型应该如下：

```
int  numeric_palindrome( int value );
```

如果 value 是回文，函数返回真，否则返回假。

★★★7. 某个数据文件包含了家庭成员的年龄。一个家庭各个成员的年龄都位于同一行，由空格分隔。例如，下面的数据

```
45  42  22
36  35  7  3  1
22   20
```

描述了三个家庭的所有成员的年龄，它们分别有 3 个、5 个和 2 个成员。

编写一个程序，计算用这种文件表示的每个家庭所有成员的平均年龄。程序应该用格式代码%5.2f 打印出平均年龄，后面是一个冒号和输入数据。你可以假定每个家庭的成员数量都不超过 10 个。

★★★★8. 编写一个程序，产生一个文件的十六进制倾印码(dump)。它应该从命令行接受单个参数，也就是需要进行倾印的文件名。如果命令行中未给出参数，程序就打印标准输入的倾印码。

倾印码的每行都应该具有下面的格式。

列	内　　容
1-6	文件的当前偏移位置，用十六进制表示，前面用零填充
9-43	文件接下来 16 个字节的十六进制表示形式。它们分成 4 组，每组由 8 个十六进制数字组成，每组之间以一个空格分隔
46	一个星号
47-62	文件中上述 16 个字节的字符表示形式。如果某个字符是不可打印字符或空白，就打印一个句点
63	一个星号

所有的十六进制数应该使用大写的 A-F 而不是小写的 a-f。

下面是一些样例行，用于说明这种格式。

```
000200  D405C000 82102004 91D02000 9010207F  *...... ... ... .*
000210  82102001 91D02000 0001C000 2F757372  *.. ... ...../usr*
000220  2F6C6962 2F6C642E 736F002F 6465762F  */lib/ld.so./dev/*
```

★★★ 9. UNIX 的 fgrep 程序从命令行接受一个字符串和一系列文件名作为参数。然后，它逐个查看每个文件的内容。对于文件中每个包含命令行中给定字符串的文本行，程序将打印出它所在的文件名、一个冒号和包含该字符串的行。

编写这个程序。首先出现的是字符串参数，它不包含任何换行字符。然后是文件名参数。如果没有给出任何文件名，程序应该从标准输入读取。在这种情况下，程序所打印的行不包括文件名和冒号。你可以假定各文件所有文本行的长度都不会超过 510 个字符。

★★★★ 10. 编写一个程序，计算文件的检验和(checksum)。该程序按照下面的方式进行调用：

```
$ sum [ -f ] [ file ... ]
```

其中，-f 选项是可选的。稍后我将描述它的含义。

接下来是一个可选的文件名列表，如果未给出任何文件名，程序就处理标准输入。否则，程序根据各个文件在命令行中出现的顺序逐个对它们进行处理。“处理文件”就是计算和打印文件的检验和。

计算检验和的算法是很简单的。文件中的每个字符都和一个 16 位的无符号整数相加，其结果就是检验和的值。不过，虽然它很容易实现，但这个算法可不是个优秀的错误检测方法。在文件中对两个字符进行互换将不会被这种方法检测出是个错误。

正常情况下，当到达每个文件的文件尾时，检验和就写入到标准输出。如果命令行中给出了-f 选项，检验和就写入到一个文件而不是标准输出。如果输入文件的名字是 file，那么这个输出文件的名字应该是 file.cks。当程序从标准输入读取时，这个选项是非法的，因为此时并不存在输入文件名。

下面是这个程序运行的几个例子。它们在那些使用 ASCII 字符集的系统中是有效的。文件 hw 包含了文本行“Hello, World!”，后面跟一个换行符。文件 hw2 包含了两个这样的文本行。所有的输入都不包含任何缀尾的空格或制表符。

```
$ sum
hi
^D
```

```
219
$ sum hw
1095
$ sum -f
-f illegal when reading standard input
$ sum -f hw2
$
```

(File hw2.cks now contains 2190)

★★★★★ 11. 编写一个程序，保存零件和它们的价值的存货记录。每个零件都有一份描述信息，其长度为 1～20 个字符。当一个新零件被添加到存货记录文件时，程序将下一个可用的零件号指定给它。第 1 个零件的零件号为 1。程序应该存储每个零件的当前数量和总价值。

这个程序应该从命令行接受单个参数，也就是存货记录文件的名字。如果这个文件并不存在，程序就创建一个空的存货记录文件。然后程序要求用户输入需要处理的事务类型并逐个对它们进行处理。

程序允许处理下列交易。

```
new description, quantity, cost-each
```

new 交易向系统添加一个新零件。descrption 是该零件的描述信息，它的长度不超过 20 个字符。quantity 是保存到存货记录文件中该零件的数量，它不可以是个负数。cost-each 是每个零件的单价。一个新零件的描述信息如果和一个现有的零件相同并不是错误。程序必须计算和保存这些零件的总价值。对于每个新增加的零件，程序为其指定下一个可用的零件号。零件号从 1 开始，线性递增。被删除零件的零件号可以重新分配给新添加的零件。

```
buy part-number, quantity, cost-each
```

buy 交易为存货记录中一个现存的零件增加一定的数量。part-number 是该零件的零件号，quantity 是购入的零件数量（它不能是负数），cost-each 是每个零件的单价。程序应该把新的零件数量和总价值添加到原先的存货记录中。

```
sell part-number, quantity, price-each
```

sell 交易从存货记录中一个现存的零件减去一定的数量。part-number 是该零件的零件号，quantity 是出售的零件数量（它不能是负数，也不能超过该零件的现有数量），price-each 是每个零件出售所获得的金额。程序应该从存货记录中减去这个数量，并减少该零件的总价值。然后，它应该计算销售所获得的利润，也就是零件的购买价格和零件的出售价格之间的差价。

```
delete part-number
```

这个交易从存货记录文件中删除指定的零件。

```
print part-number
```

这个交易打印指定零件的信息，包括描述信息、现存数量和零件的总价值。

```
print all
```

这个交易以表格的形式打印记录中所有零件的信息。

```
total
```

* 这个交易计算和打印记录中所有零件的总价值。

```
end
```

这个交易终止程序的执行。

当零件以不同的购买价格获得时，计算存货记录的真正价值将变得很复杂，而且取决于首先使用的是最便宜的零件还是最昂贵的零件。这个程序所使用的方法比较简单：只保存每种零件的总价值，每种零件的单价被认为是相等的。例如，假定 10 个纸夹原先以每个$1.00 的价格购买。这个零件的总价值便是$10.00。以后，又以每个$1.25 的价格购入另外 10 个纸夹，这样这个零件的总价值便成了$22.50。此时，每个纸夹的当前单价便是$1.125。存货记录并不保存每批零件的购买记录，即使它们的购买价格不同。当纸夹出售时，利润根据上面计算所得的当前单价进行计算。

这里有一些关于设计这个程序的提示。首先，使用零件号判断存货记录文件中一个零件的写入位置。第 1 个零件号是 1，这样记录文件中零件号为 0 的位置可以用于保存一些其他信息。其次，你可以在删除零件时把它的的描述信息设置为空字符串，便于以后检测该零件是否已被删除。