# 第六章 文件操作

## 基本知识：

## 一、文件

文件是程序设计中的一个重要概念。所谓“文件”一般是指存储在外部介质上数据的集合。一批文件是以数据的形式存放在外部介质（如磁盘）上的。操作系统是以文件为单位对数据进行管理的，也就是说，如果想找存在外部介质上的数据，必须先按文件名找到指定的文件，然后再从该文件中读取数据。要向外部介质上存储数据也必须先建立一个文件（以文件名标识），才能向它输出数据。

从操作系统的角度来看，每一个与主机相连的输出输入设备都看作是一个文件。

在程序运行时，常常需要将一些数据（运行的最终结果或中间数据）输出到磁盘上存放起来，以后需要时再从磁盘中输入到计算机的内存。这就要用到磁盘文件。

C语言将文件看作是一个[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6" \t "_blank)（[字节](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82)）的序列，即一个一个字符（字节）的数据顺序组成。根据数据的组成形式，可分为ASCII文件和二进制文件。ASCII文件又称文本（text)文件,它的每一个字节可放一个ASCII码，代表一个字符。二进制文件是把内存中的数据按其在内存中的存储形式按原样输出到磁盘上存放。因而一个C文件就是一个字节流或二进制流。它把数据看作是一连串的字符（字节），而不考虑记录的界限。换句话说，C语言中文件不是由记录（record)组成的（这是和PASCAL或其他高级语言不同的）。在C语言中对文件的存取是以字符（字节）为单位的。输出输入的数据流的开始和结束仅受程序控制而不受物理符号（如回车换行符）控制。也就是说，在输出时不会自动增加回车换行符作为记录结束的标志，输入时不以回车换行符作为记录的间隔（事实上C文件并不是由记录构成的）。把这种文件称为流式文件。C语言允许对文件存取一个字符，这就增加了处理的灵活性。

## 二、标准I/O文件操作

fopen、fclose、fwrite、fread、fgetc、fgets、fseek、ftell、 feof、 fflush

### 2.1 fopen/fclose

在操作文件之前要用fopen打开文件，操作完毕要用fclose关闭文件。打开文件就是在操作系统中分配一些资源用于保存该文件的状态信息，并得到该文件的标识，以后用户程序就可以用这个标识对文件做各种操作，关闭文件则释放文件在操作系统中占用的资源，使文件的标识失效，用户程序就无法再操作这个文件了。

#include <stdio.h>

FILE \*fopen(const char \*path, const char \*mode);

返回值：成功返回文件指针，出错返回NULL并设置errno

path是文件的路径名，mode表示打开方式。如果文件打开成功，就返回一个FILE \*文件指针来标识这个文件。以后调用其它函数对文件做读写操作都要提供这个指针，以指明对哪个文件进行操作。FILE是C标准库中定义的结构体类型，其中包含该文件在内核中标识、I/O缓冲区和当前读写位置等信息，但调用者不必知道FILE结构体都有哪些成员，我们很快就会看到，调用者只是把文件指针在库函数接口之间传来传去，而文件指针所指的FILE结构体的成员在库函数内部维护，调用者不应该直接访问这些成员，这种编程思想在面向对象方法论中称为封装（Encapsulation）。像FILE \*这样的指针称为不透明指针（Opaque Pointer）或者叫句柄（Handle），FILE \*指针就像一个把手（Handle），抓住这个把手就可以打开门或抽屉，但用户只能抓这个把手，而不能直接抓门或抽屉。

下面说说参数path和mode，path可以是相对路径也可以是绝对路径，mode表示打开方式是读还是写。比如fp = fopen("/tmp/file2", "w");表示打开绝对路径/tmp/file2，只做写操作，path也可以是相对路径，比如fp = fopen("file.a", "r");表示在当前工作目录下打开文件file.a，只做读操作，再比如fp = fopen("../a.out", "r");只读打开当前工作目录上一层目录下的a.out，fp = fopen("Desktop/file3", "w");只写打开当前工作目录下子目录Desktop下的file3。相对路径是相对于当前工作目录（Current Working Directory）的路径，每个进程都有自己的当前工作目录，Shell进程的当前工作目录可以用pwd命令查看：

$ pwd

/home/akaedu

通常Linux发行版都把Shell配置成在提示符前面显示当前工作目录，例如~$表示当前工作目录是主目录，/etc$表示当前工作目录是/etc。用cd命令可以改变Shell进程的当前工作目录。在Shell下敲命令启动新的进程，则该进程的当前工作目录继承自Shell进程的当前工作目录，该进程也可以调用chdir(2)函数改变自己的当前工作目录。

mode参数是一个字符串，由rwatb+六个字符组合而成，r表示读，w表示写，a表示追加（Append），在文件末尾追加数据使文件的尺寸增大。t表示文本文件，b表示二进制文件，有些操作系统的文本文件和二进制文件格式不同，而在UNIX系统中，无论文本文件还是二进制文件都是由一串字节组成，t和b没有区分，用哪个都一样，也可以省略不写。如果省略t和b，rwa+四个字符有以下6种合法的组合：

"r"

只读，文件必须已存在

"w"

只写，如果文件不存在则创建，如果文件已存在则把文件长度截断（Truncate）为0字节再重新写，也就是替换掉原来的文件内容

"a"

只能在文件末尾追加数据，如果文件不存在则创建

"r+"

允许读和写，文件必须已存在

"w+"

允许读和写，如果文件不存在则创建，如果文件已存在则把文件长度截断为0字节再重新写

"a+"

允许读和追加数据，如果文件不存在则创建

在打开一个文件时如果出错，fopen将返回NULL并设置errno，errno稍后介绍。在程序中应该做出错处理，通常这样写：

if ( (fp = fopen("/tmp/file1", "r")) == NULL) {

printf("error open file /tmp/file1!\n");

exit(1);

}

比如/tmp/file1这个文件不存在，而r打开方式又不会创建这个文件，fopen就会出错返回。

再说说fclose函数。

#include <stdio.h>

int fclose(FILE \*fp);

返回值：成功返回0，出错返回EOF并设置errno

把文件指针传给fclose可以关闭它所标识的文件，关闭之后该文件指针就无效了，不能再使用了。如果fclose调用出错（比如传给它一个无效的文件指针）则返回EOF并设置errno，errno稍后介绍，EOF在stdio.h中定义：

/\* End of file character.

Some things throughout the library rely on this being -1. \*/

#ifndef EOF

# define EOF (-1)

#endif

它的值是-1。fopen调用应该和fclose调用配对，打开文件操作完之后一定要记得关闭。如果不调用fclose，在进程退出时系统会自动关闭文件，但是不能因此就忽略fclose调用，如果写一个长年累月运行的程序（比如网络服务器程序），打开的文件都不关闭，堆积得越来越多，就会占用越来越多的系统资源。

### 2.2 errno与perror函数

很多系统函数在错误返回时将错误原因记录在libc定义的全局变量errno中，每种错误原因对应一个错误码，请查阅errno(3)的Man Page了解各种错误码，errno在头文件errno.h中声明，是一个整型变量，所有错误码都是正整数。

如果在程序中打印错误信息时直接打印errno变量，打印出来的只是一个整数值，仍然看不出是什么错误。比较好的办法是用perror或strerror函数将errno解释成字符串再打印。

#include <stdio.h>

void perror(const char \*s);

perror函数将错误信息打印到标准错误输出，首先打印参数s所指的字符串，然后打印:号，然后根据当前errno的值打印错误原因。例如：

**例 25.4. perror**

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE \*fp = fopen("abcde", "r");

if (fp == NULL) {

perror("Open file abcde");

exit(1);

}

return 0;

}

如果文件abcde不存在，fopen返回-1并设置errno为ENOENT，紧接着perror函数读取errno的值，将ENOENT解释成字符串No such file or directory并打印，最后打印的结果是Open file abcde: No such file or directory。虽然perror可以打印出错误原因，传给perror的字符串参数仍然应该提供一些额外的信息，以便在看到错误信息时能够很快定位是程序中哪里出了错，如果在程序中有很多个fopen调用，每个fopen打开不同的文件，那么在每个fopen的错误处理中打印文件名就很有帮助。

如果把上面的程序改成这样：

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

int main(void)

{

FILE \*fp = fopen("abcde", "r");

if (fp == NULL) {

perror("Open file abcde");

printf("errno: %d\n", errno);

exit(1);

}

return 0;

}

则printf打印的错误号并不是fopen产生的错误号，而是perror产生的错误号。errno是一个全局变量，很多系统函数都会改变它，fopen函数Man Page中的ERRORS部分描述了它可能产生的错误码，perror函数的Man Page中没有ERRORS部分，说明它本身不产生错误码，但它调用的其它函数也有可能改变errno变量。大多数系统函数都有一个Side Effect，就是有可能改变errno变量（当然也有少数例外，比如strcpy），所以一个系统函数错误返回后应该马上检查errno，在检查errno之前不能再调用其它系统函数。

strerror函数可以根据错误号返回错误原因字符串。

#include <string.h>

char \*strerror(int errnum);

返回值：错误码errnum所对应的字符串

这个函数返回指向静态内存的指针。以后学线程库时我们会看到，有些函数的错误码并不保存在errno中，而是通过返回值返回，就不能调用perror打印错误原因了，这时strerror就派上了用场：

fputs(strerror(n), stderr);

### 2.3. 以字节为单位的I/O函数

fgetc函数从指定的文件中读一个字节，getchar从标准输入读一个字节，调用getchar()相当于调用fgetc(stdin)。

#include <stdio.h>

int fgetc(FILE \*stream);

int getchar(void);

返回值：成功返回读到的字节，出错或者读到文件末尾时返回EOF

注意在Man Page的函数原型中FILE \*指针参数有时会起名叫stream，这是因为标准I/O库操作的文件有时也叫做流（Stream），文件由一串字节组成，每次可以读或写其中任意数量的字节，以后介绍TCP协议时会对流这个概念做更详细的解释。

对于fgetc函数的使用有以下几点说明：

要用fgetc函数读一个文件，该文件的打开方式必须是可读的。

系统对于每个打开的文件都记录着当前读写位置在文件中的地址（或者说距离文件开头的字节数），也叫偏移量（Offset）。当文件打开时，读写位置是0，每调用一次fgetc，读写位置向后移动一个字节，因此可以连续多次调用fgetc函数依次读取多个字节。

fgetc成功时返回读到一个字节，本来应该是unsigned char型的，但由于函数原型中返回值是int型，所以这个字节要转换成int型再返回，那为什么要规定返回值是int型呢？因为出错或读到文件末尾时fgetc将返回EOF，即-1，保存在int型的返回值中是0xffffffff，如果读到字节0xff，由unsigned char型转换为int型是0x000000ff，只有规定返回值是int型才能把这两种情况区分开，如果规定返回值是unsigned char型，那么当返回值是0xff时无法区分到底是EOF还是字节0xff。如果需要保存fgetc的返回值，一定要保存在int型变量中，如果写成unsigned char c = fgetc(fp);，那么根据c的值又无法区分EOF和0xff字节了。注意，fgetc读到文件末尾时返回EOF，只是用这个返回值表示已读到文件末尾，并不是说每个文件末尾都有一个字节是EOF（根据上面的分析，EOF并不是一个字节）。

fputc函数向指定的文件写一个字节，putchar向标准输出写一个字节，调用putchar(c)相当于调用fputc(c, stdout)。

#include <stdio.h>

int fputc(int c, FILE \*stream);

int putchar(int c);

返回值：成功返回写入的字节，出错返回EOF

对于fputc函数的使用也要说明几点：

要用fputc函数写一个文件，该文件的打开方式必须是可写的（包括追加）。

每调用一次fputc，读写位置向后移动一个字节，因此可以连续多次调用fputc函数依次写入多个字节。但如果文件是以追加方式打开的，每次调用fputc时总是将读写位置移到文件末尾然后把要写入的字节追加到后面。

下面的例子演示了这四个函数的用法，从键盘读入一串字符写到一个文件中，再从这个文件中读出这些字符打印到屏幕上。

**例 25.5. 用fputc/fget读写文件和终端**

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE \*fp;

int ch;

if ( (fp = fopen("file2", "w+")) == NULL) {

perror("Open file file2\n");

exit(1);

}

while ( (ch = getchar()) != EOF)

fputc(ch, fp);

rewind(fp);

while ( (ch = fgetc(fp)) != EOF)

putchar(ch);

fclose(fp);

return 0;

}

从终端设备读有点特殊。当调用getchar()或fgetc(stdin)时，如果用户没有输入字符，getchar函数就阻塞等待，所谓阻塞是指这个函数调用不返回，也就不能执行后面的代码，这个进程阻塞了，操作系统可以调度别的进程执行。从终端设备读还有一个特点，用户输入一般字符并不会使getchar函数返回，仍然阻塞着，只有当用户输入回车或者到达文件末尾时getchar才返回[[34](file:///D:\studydir\%E4%B9%A6%E7%B1%8D%E5%A4%A7%E5%85%A810G\%E8%B5%84%E6%96%99\Linux%20C%E7%BC%96%E7%A8%8B%E4%B8%80%E7%AB%99%E5%BC%8F%E5%AD%A6%E4%B9%A0\ch25s02.html#ftn.id2831641)]。这个程序的执行过程分析如下：

$ ./a.out

hello（输入hello并回车，这时第一次调用getchar返回，读取字符h存到文件中，然后连续调用getchar五次，读取ello和换行符存到文件中，第七次调用getchar又阻塞了）

hey（输入hey并回车，第七次调用getchar返回，读取字符h存到文件中，然后连续调用getchar三次，读取ey和换行符存到文件中，第11次调用getchar又阻塞了）

（这时输入Ctrl-D，第11次调用getchar返回EOF，跳出循环，进入下一个循环，回到文件开头，把文件内容一个字节一个字节读出来打印，直到文件结束）

hello

hey

从终端设备输入时有两种方法表示文件结束，一种方法是在一行的开头输入Ctrl-D（如果不在一行的开头则需要连续输入两次Ctrl-D），另一种方法是利用Shell的Heredoc语法：

$ ./a.out <<END

> hello

> hey

> END

hello

hey

<<END表示从下一行开始是标准输入，直到某一行开头出现END时结束。<<后面的结束符可以任意指定，不一定得是END，只要和输入的内容能区分开就行。

在上面的程序中，第一个while循环结束时fp所指文件的读写位置在文件末尾，然后调用rewind函数把读写位置移到文件开头，再进入第二个while循环从头读取文件内容。

### 2.4. 操作读写位置的函数

我们在上一节的例子中看到rewind函数把读写位置移到文件开头，本节介绍另外两个操作读写位置的函数，fseek可以任意移动读写位置，ftell可以返回当前的读写位置。

#include <stdio.h>

int fseek(FILE \*stream, long offset, int whence);

返回值：成功返回0，出错返回-1并设置errno

long ftell(FILE \*stream);

返回值：成功返回当前读写位置，出错返回-1并设置errno

void rewind(FILE \*stream);

fseek的whence和offset参数共同决定了读写位置移动到何处，whence参数的含义如下：

SEEK\_SET

从文件开头移动offset个字节

SEEK\_CUR

从当前位置移动offset个字节

SEEK\_END

从文件末尾移动offset个字节

offset可正可负，负值表示向前（向文件开头的方向）移动，正值表示向后（向文件末尾的方向）移动，如果向前移动的字节数超过了文件开头则出错返回，如果向后移动的字节数超过了文件末尾，再次写入时将增大文件尺寸，从原来的文件末尾到fseek移动之后的读写位置之间的字节都是0。

先前我们创建过一个文件textfile，其中有五个字节，5678加一个换行符，现在我们拿这个文件做实验。

例 25.6. fseek

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE\* fp;

if ( (fp = fopen("textfile","r+")) == NULL) {

perror("Open file textfile");

exit(1);

}

if (fseek(fp, 10, SEEK\_SET) != 0) {

perror("Seek file textfile");

exit(1);

}

fputc('K', fp);

fclose(fp);

return 0;

}

运行这个程序，然后查看文件textfile的内容：

$ ./a.out

$ od -tx1 -tc -Ax textfile

000000 35 36 37 38 0a 00 00 00 00 00 4b

5 6 7 8 \n \0 \0 \0 \0 \0 K

00000b

fseek(fp, 10, SEEK\_SET)将读写位置移到第10个字节处（其实是第11个字节，从0开始数），然后在该位置写入一个字符K，这样textfile文件就变长了，从第5到第9个字节自动被填充为0。

### 2.5. 以字符串为单位的I/O函数fgets、gets

fgets从指定的文件中读一行字符到调用者提供的缓冲区中，gets从标准输入读一行字符到调用者提供的缓冲区中。

#include <stdio.h>

char \*fgets(char \*s, int size, FILE \*stream);

char \*gets(char \*s);

返回值：成功时s指向哪返回的指针就指向哪，出错或者读到文件末尾时返回NULL

gets函数无需解释，Man Page的BUGS部分已经说得很清楚了：Never use gets()。gets函数的存在只是为了兼容以前的程序，我们写的代码都不应该调用这个函数。gets函数的接口设计得很有问题，就像strcpy一样，用户提供一个缓冲区，却不能指定缓冲区的大小，很可能导致缓冲区溢出错误，这个函数比strcpy更加危险，strcpy的输入和输出都来自程序内部，只要程序员小心一点就可以避免出问题，而gets读取的输入直接来自程序外部，用户可能通过标准输入提供任意长的字符串，程序员无法避免gets函数导致的缓冲区溢出错误，所以唯一的办法就是不要用它。

现在说说fgets函数，参数s是缓冲区的首地址，size是缓冲区的长度，该函数从stream所指的文件中读取以'\n'结尾的一行（包括'\n'在内）存到缓冲区s中，并且在该行末尾添加一个'\0'组成完整的字符串。

如果文件中的一行太长，fgets从文件中读了size-1个字符还没有读到'\n'，就把已经读到的size-1个字符和一个'\0'字符存入缓冲区，文件中剩下的半行可以在下次调用fgets时继续读。

如果一次fgets调用在读入若干个字符后到达文件末尾，则将已读到的字符串加上'\0'存入缓冲区并返回，如果再次调用fgets则返回NULL，可以据此判断是否读到文件末尾。

注意，对于fgets来说，'\n'是一个特别的字符，而'\0'并无任何特别之处，如果读到'\0'就当作普通字符读入。如果文件中存在'\0'字符（或者说0x00字节），调用fgets之后就无法判断缓冲区中的'\0'究竟是从文件读上来的字符还是由fgets自动添加的结束符，所以fgets只适合读文本文件而不适合读二进制文件，并且文本文件中的所有字符都应该是可见字符，不能有'\0'。

fputs向指定的文件写入一个字符串，puts向标准输出写入一个字符串。

#include <stdio.h>

int fputs(const char \*s, FILE \*stream);

int puts(const char \*s);

返回值：成功返回一个非负整数，出错返回EOF

缓冲区s中保存的是以'\0'结尾的字符串，fputs将该字符串写入文件stream，但并不写入结尾的'\0'。与fgets不同的是，fputs并不关心的字符串中的'\n'字符，字符串中可以有'\n'也可以没有'\n'。puts将字符串s写到标准输出（不包括结尾的'\0'），然后自动写一个'\n'到标准输出。

### 2.6. 以记录为单位的I/O函数fread、fwrite

#include <stdio.h>

size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

返回值：读或写的记录数，成功时返回的记录数等于nmemb，出错或读到文件末尾时返回的记录数小于nmemb，也可能返回0

fread和fwrite用于读写记录，这里的记录是指一串固定长度的字节，比如一个int、一个结构体或者一个定长数组。参数size指出一条记录的长度，而nmemb指出要读或写多少条记录，这些记录在ptr所指的内存空间中连续存放，共占size \* nmemb个字节，fread从文件stream中读出size \* nmemb个字节保存到ptr中，而fwrite把ptr中的size \* nmemb个字节写到文件stream中。

nmemb是请求读或写的记录数，fread和fwrite返回的记录数有可能小于nmemb指定的记录数。例如当前读写位置距文件末尾只有一条记录的长度，调用fread时指定nmemb为2，则返回值为1。如果当前读写位置已经在文件末尾了，或者读文件时出错了，则fread返回0。如果写文件时出错了，则fwrite的返回值小于nmemb指定的值。下面的例子由两个程序组成，一个程序把结构体保存到文件中，另一个程序和从文件中读出结构体。

**例 25.7. fread/fwrite**

/\* writerec.c \*/

#include <stdio.h>

#include <stdlib.h>

struct record {

char name[10];

int age;

};

int main(void)

{

struct record array[2] = {{"Ken", 24}, {"Knuth", 28}};

FILE \*fp = fopen("recfile", "w");

if (fp == NULL) {

perror("Open file recfile");

exit(1);

}

fwrite(array, sizeof(struct record), 2, fp);

fclose(fp);

return 0;

}

/\* readrec.c \*/

#include <stdio.h>

#include <stdlib.h>

struct record {

char name[10];

int age;

};

int main(void)

{

struct record array[2];

FILE \*fp = fopen("recfile", "r");

if (fp == NULL) {

perror("Open file recfile");

exit(1);

}

fread(array, sizeof(struct record), 2, fp);

printf("Name1: %s\tAge1: %d\n", array[0].name, array[0].age);

printf("Name2: %s\tAge2: %d\n", array[1].name, array[1].age);

fclose(fp);

return 0;

}

$ gcc writerec.c -o writerec

$ gcc readrec.c -o readrec

$ ./writerec

$ od -tx1 -tc -Ax recfile

000000 4b 65 6e 00 00 00 00 00 00 00 00 00 18 00 00 00

K e n \0 \0 \0 \0 \0 \0 \0 \0 \0 030 \0 \0 \0

000010 4b 6e 75 74 68 00 00 00 00 00 00 00 1c 00 00 00

K n u t h \0 \0 \0 \0 \0 \0 \0 034 \0 \0 \0

000020

$ ./readrec

Name1: Ken Age1: 24

Name2: Knuth Age2: 28

我们把一个struct record结构体看作一条记录，由于结构体中有填充字节，每条记录占16字节，把两条记录写到文件中共占32字节。该程序生成的recfile文件是二进制文件而非文本文件，因为其中不仅保存着字符型数据，还保存着整型数据24和28（在od命令的输出中以八进制显示为030和034）。注意，直接在文件中读写结构体的程序是不可移植的，如果在一种平台上编译运行writebin.c程序，把生成的recfile文件拷到另一种平台并在该平台上编译运行readbin.c程序，则不能保证正确读出文件的内容，因为不同平台的大小端可能不同（因而对整型数据的存储方式不同），结构体的填充方式也可能不同（因而同一个结构体所占的字节数可能不同，age成员在name成员之后的什么位置也可能不同）。

### 2.7. C标准库的I/O缓冲区

用户程序调用C标准I/O库函数读写文件或设备，而这些库函数要通过系统调用把读写请求传给内核（以后我们会看到与I/O相关的系统调用），最终由内核驱动磁盘或设备完成I/O操作。C标准库为每个打开的文件分配一个I/O缓冲区以加速读写操作，通过文件的FILE结构体可以找到这个缓冲区，用户调用读写函数大多数时候都在I/O缓冲区中读写，只有少数时候需要把读写请求传给内核。以fgetc/fputc为例，当用户程序第一次调用fgetc读一个字节时，fgetc函数可能通过系统调用进入内核读1K字节到I/O缓冲区中，然后返回I/O缓冲区中的第一个字节给用户，把读写位置指向I/O缓冲区中的第二个字符，以后用户再调fgetc，就直接从I/O缓冲区中读取，而不需要进内核了，当用户把这1K字节都读完之后，再次调用fgetc时，fgetc函数会再次进入内核读1K字节到I/O缓冲区中。在这个场景中用户程序、C标准库和内核之间的关系就像CPU、Cache和内存之间的关系一样，C标准库之所以会从内核预读一些数据放在I/O缓冲区中，是希望用户程序随后要用到这些数据，C标准库的I/O缓冲区也在用户空间，直接从用户空间读取数据比进内核读数据要快得多。另一方面，用户程序调用fputc通常只是写到I/O缓冲区中，这样fputc函数可以很快地返回，如果I/O缓冲区写满了，fputc就通过系统调用把I/O缓冲区中的数据传给内核，内核最终把数据写回磁盘。有时候用户程序希望把I/O缓冲区中的数据立刻传给内核，让内核写回设备，这称为Flush操作，对应的库函数是fflush，fclose函数在关闭文件之前也会做Flush操作。

下图以fgets/fputs示意了I/O缓冲区的作用，使用fgets/fputs函数时在用户程序中也需要分配缓冲区（图中的buf1和buf2），注意区分用户程序的缓冲区和C标准库的I/O缓冲区。

C标准库的I/O缓冲区有三种类型：全缓冲、行缓冲和无缓冲。当用户程序调用库函数做写操作时，不同类型的缓冲区具有不同的特性。

全缓冲

如果缓冲区写满了就写回内核。常规文件通常是全缓冲的。

行缓冲

如果用户程序写的数据中有换行符就把这一行写回内核，或者如果缓冲区写满了就写回内核。标准输入和标准输出对应终端设备时通常是行缓冲的。

无缓冲

用户程序每次调库函数做写操作都要通过系统调用写回内核。标准错误输出通常是无缓冲的，这样用户程序产生的错误信息可以尽快输出到设备。

下面通过一个简单的例子证明标准输出对应终端设备时是行缓冲的。

#include <stdio.h>

int main()

{

printf("hello world");

while(1);

return 0;

}

运行这个程序，会发现hello world并没有打印到屏幕上。用Ctrl-C终止它，去掉程序中的while(1);语句再试一次：

$ ./a.out

hello world$

hello world被打印到屏幕上，后面直接跟Shell提示符，中间没有换行。

我们知道main函数被启动代码这样调用：exit(main(argc, argv));。main函数return时启动代码会调用exit，exit函数首先关闭所有尚未关闭的FILE \*指针（关闭之前要做Flush操作），然后通过\_exit系统调用进入内核退出当前进程[[35](file:///D:\studydir\%E4%B9%A6%E7%B1%8D%E5%A4%A7%E5%85%A810G\%E8%B5%84%E6%96%99\Linux%20C%E7%BC%96%E7%A8%8B%E4%B8%80%E7%AB%99%E5%BC%8F%E5%AD%A6%E4%B9%A0\ch25s02.html#ftn.id2834688)]。

在上面的例子中，由于标准输出是行缓冲的，printf("hello world");打印的字符串中没有换行符，所以只把字符串写到标准输出的I/O缓冲区中而没有写回内核（写到终端设备），如果敲Ctrl-C，进程是异常终止的，并没有调用exit，也就没有机会Flush I/O缓冲区，因此字符串最终没有打印到屏幕上。如果把打印语句改成printf("hello world\n");，有换行符，就会立刻写到终端设备，或者如果把while(1);去掉也可以写到终端设备，因为程序退出时会调用exitFlush所有I/O缓冲区。在本书的其它例子中，printf打印的字符串末尾都有换行符，以保证字符串在printf调用结束时就写到终端设备。

我们再做个实验，在程序中直接调用\_exit退出。

#include <stdio.h>

#include <unistd.h>

int main()

{

printf("hello world");

\_exit(0);

}

结果也不会把字符串打印到屏幕上，如果把\_exit调用改成exit就可以打印到屏幕上。

除了写满缓冲区、写入换行符之外，行缓冲还有一种情况会自动做Flush操作。如果：

用户程序调用库函数从无缓冲的文件中读取

或者从行缓冲的文件中读取，并且这次读操作会引发系统调用从内核读取数据

那么在读取之前会自动Flush所有行缓冲。例如：

#include <stdio.h>

#include <unistd.h>

int main()

{

char buf[20];

printf("Please input a line: ");

fgets(buf, 20, stdin);

return 0;

}

虽然调用printf并不会把字符串写到设备，但紧接着调用fgets读一个行缓冲的文件（标准输入），在读取之前会自动Flush所有行缓冲，包括标准输出。

如果用户程序不想完全依赖于自动的Flush操作，可以调fflush函数手动做Flush操作。

#include <stdio.h>

int fflush(FILE \*stream);

返回值：成功返回0，出错返回EOF并设置errno

对前面的例子再稍加改动：

#include <stdio.h>

int main()

{

printf("hello world");

fflush(stdout);

while(1);

}

虽然字符串中没有换行，但用户程序调用fflush强制写回内核，因此也能在屏幕上打印出字符串。fflush函数用于确保数据写回了内核，以免进程异常终止时丢失数据。作为一个特例，调用fflush(NULL)可以对所有打开文件的I/O缓冲区做Flush操作。

## 实训任务：

### 小写转大写

从键盘输入一个字符串，将其中的小写字母全部转换成大写字母，然后输出到一个磁盘文件test中保存。

### 合并文件（1）

有两个磁盘文件“A”和‘’B“，各存放一行字母，要求把这两个文件中的信息合并（按字母顺序排列），输出到一个新文件”C“中。

### 合并文件（2）

有两个文本文档，第一个文本文档的内容是：“书中自有黄金屋“，第二个文本文档的内容是：”书中自有颜如玉“，编程实现合并两个文件信息，即将文档2的内容合并到文档1内容的后面。

### 将数据写入磁盘文件

编程实现将数据写入磁盘文件，即在任意路径下新建一个文本文档，向该文档中写入“好好学习，天天向上“，并以”#“号结束字符串的输入。

### 文档复制

编程实现将现有文本文档的内容复制到新建的文本文档中。

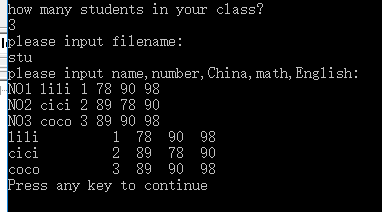
### 结构数据存储

有5个学生，每个学生有3门课的成绩，从键盘输入数据（包括学号、姓名、3门课的成绩），计算出平均成绩，将原有数据和计算出的平均分数存放在磁盘文件stdu中。

### 成块读写数据

编程实现学生成绩信息的输入与输出，通过键盘输入学生成绩信息，保存到指定磁盘文件中，输入全部信息后，将磁盘文件中保存的信息输出到屏幕。

运行：



### 排序并写入文件

将上题所得到的stud文件中的学生数据按平均分进行排序处理，将已排序:的学生数据存入一个新文件stu\_sort中。

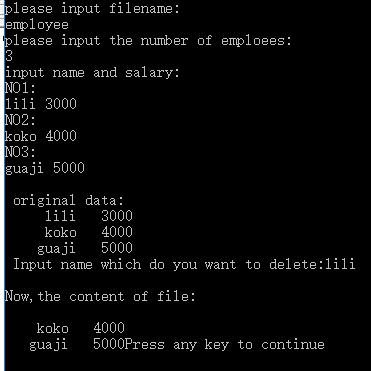
### 插入处理

将上题中已排序的学生成绩文件进行插入处理。插入一个学生的3门课程成绩，程序先计算新插入学生的平均成绩，然后将它按平均成绩高低顺序插入，插入后重新写入文件。

### 删除文件中的记录

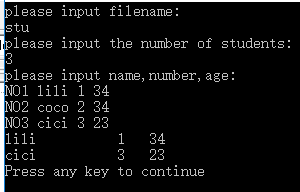
编程实现对记录中职工工资信息的删除。具体要求如下：输入路径及文件名打开文件，输入员工姓名及工资，输入完毕显示文件中的内容，输入要删除的员工姓名，进行删除操作，最后将删除后的内容显示在屏幕上。

运行结果：



### 随机读写文件

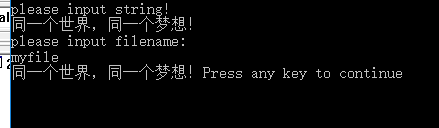
输入若干学生信息，保存到指定磁盘文件中，要求将奇数条学生信息从磁盘中读入并显示到屏幕上。

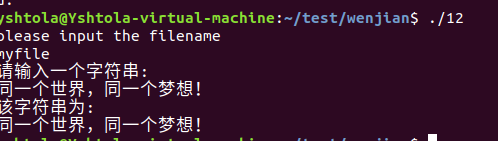


### 以行为单位读写文件

从键盘中输入字符串“同一个世界，同一个梦想！“，要求将字符串内容输出到磁盘文件，再从磁盘文件中读取字符串到数组,最终将其输出到屏幕上。

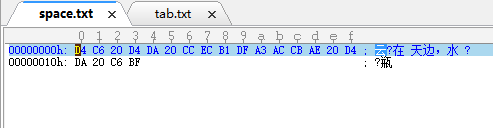
运行：

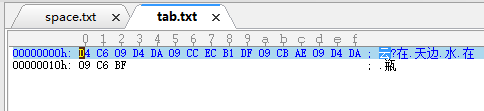




### 错误处理

编程实现将文件中的制表符换成恰当数目的空格，要求每次读写操作后都调用ferror()函数检查错误



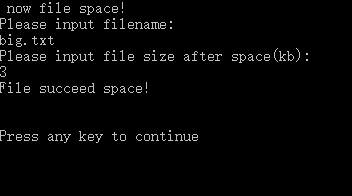
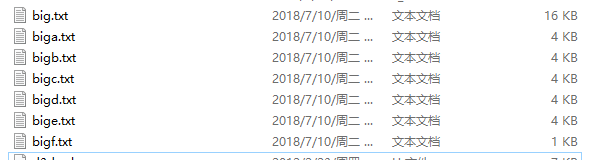


### 附加题：

### 文件分割

编程实现将一个较大的文件分割成若干个较小的文件，要求分割成的文件不改变原有文件内容。

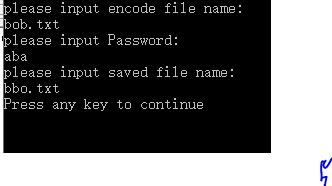
运行结果：

### 文件加密

编程实现文件加密，具体要求如下：先从键盘输入要加密的文件所在的路径及名称，再输入密码，最后输入加密后的文件要存储的路径及名称。

运行结果:



### 文秘系统管理

输入选项，实现如下功能：

1：文件拆分：由用户输入拆分的文件名，需要拆分成几个文件，根据用户输入拆分成文件的个数计算每个文件的大小。

2：文件合并：由用户循环输入需要合并的文件，输入结束后合并到第一个文件。

3：退出系统

运行结果：

