

# 计算机程序设计基础(1)

## --- C语言程序设计(12)

孙甲松

[sunjiasong@tsinghua.edu.cn](mailto:sunjiasong@tsinghua.edu.cn)

电子工程系 信息认知与智能系统研究所  
罗姆楼6-104

电话: 13901216180/62796193

2022.12.

# 第12章 文件

## 12.1 文件的基本概念

### 12.1.1 文本文件与二进制文件

### 12.1.2 缓冲文件系统

### 12.1.3 文件类型指针

## 12.2 文件的基本操作

### 12.2.1 文件的打开与关闭

### 12.2.2 文件的读写

### 12.2.3 文件的定位

### 12.2.4 文件缓冲区的清除

### 12.2.5 文件指针错误状态的清除

## 12.3 程序举例

## 12.1 文件的基本概念

文件是指存储在外存储器上的数据的集合，可永久保存、复制、传输。

每个文件都有一个名字，称为文件名。文件名的命名规则比较随意，且与操作系统密切相关，Windows操作系统的文件名中一般不能出现\*，?等。

一般来说，同一个文件目录下的不同的文件有不同的文件名，计算机操作系统就是根据文件名对各种文件进行存取，并进行处理。

### 12.1.1 文本文件与二进制文件

存储形式

- 文本文件又称为ASCII文件。在这种文件中，每个字节存放一个字符的ASCII码值。每个整数或实数是由每位数字的ASCII字符组成的。`.c` `.cpp`等源程序文件都是文本文件。

- 二进制文件中的数据与该数据在内存中的二进制形式是一致的，是把整数的补码、浮点数float和double的IEEE 754表示直接写入文件中，其中一个字节并不代表一个字符。可执行文件都是二进制文件。`.jpg`、`.mp4`、`.pdf`、`.pptx`、`.docx`等也都是二进制文件。

● 用一般的编辑器能编辑、人能直接读懂的文件是文本文件。是由ASCII字节流组成的，是流式文件。例如有程序：

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

```
main( )
```

```
{ int a=12345, b=567890;
```

```
float f=97.6875f;
```

```
double f2=97.6875;
```

```
FILE *fp;
```

```
fp = fopen("data1.txt", "w"); /* 文本文件 */
```

```
fprintf(fp, "%d %d %f %f\n", a, b, f, f2);
```

```
fclose(fp);
```

```
fp = fopen("data2.txt", "wb"); /* 二进制文件 */
```

```
fwrite(&a, sizeof(int), 1, fp);
```

```
fwrite(&b, sizeof(int), 1, fp);
```

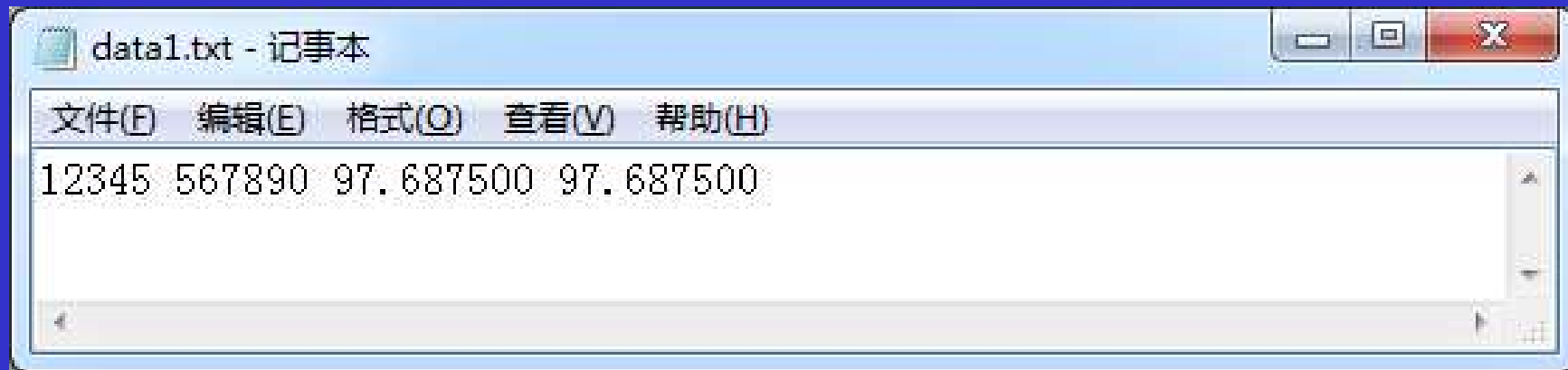
```
fwrite(&f, sizeof(float), 1, fp);
```

```
fwrite(&f2, sizeof(double), 1, fp);
```

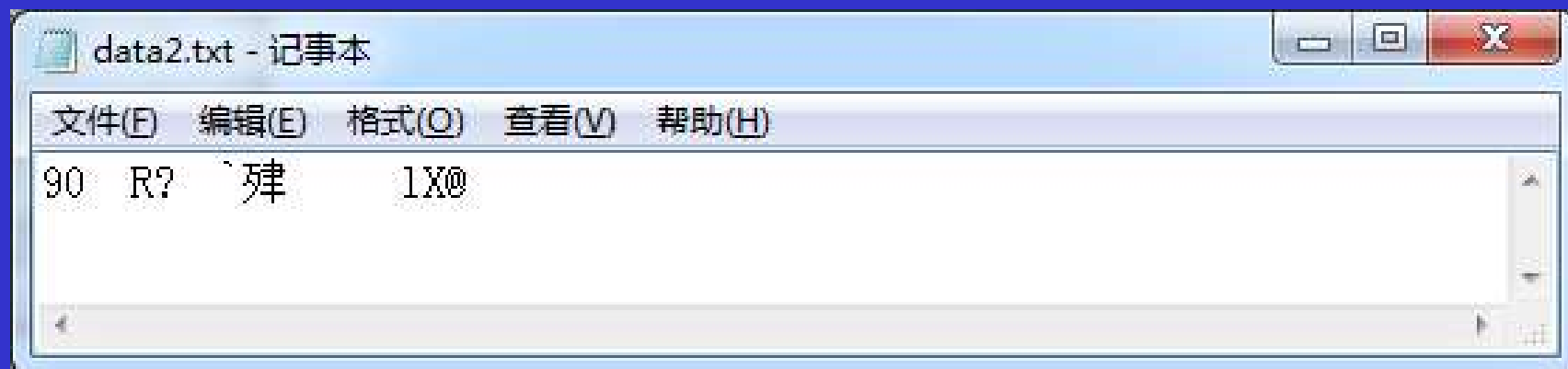
```
fclose(fp);
```

```
}
```

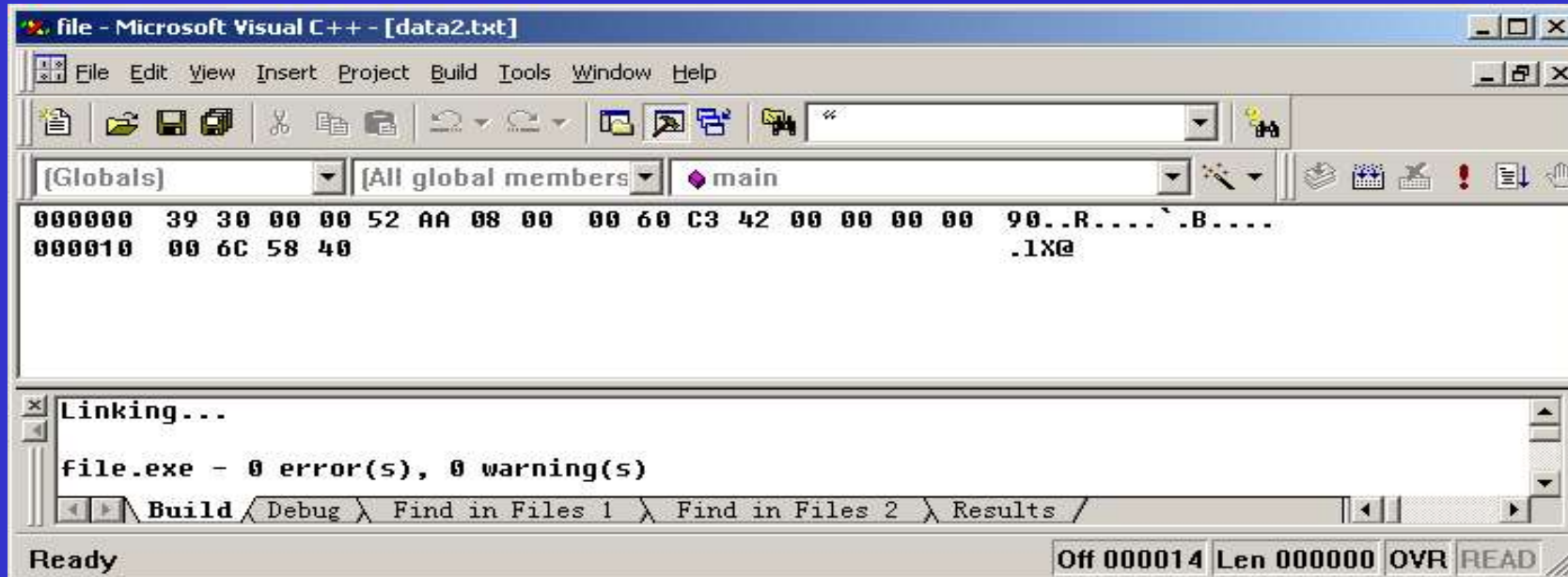
该程序将 `int a=12345,b=567890; float f=97.6875f; double f2=97.6875;` 写到文本文件 **data1.txt** 中, 二进制文件 **data2.txt** 中, 用 Windows 的记事本打开 **data1.txt** (文件长度 34 字节), 看到的是:



**12345 567890 97.687500 97.687500** 是字符型的 ASCII 字符串。  
用记事本打开 **data2.txt**, 看到的是: (此文件长度 20 字节)



# 用VS2008编辑器打开data2.txt: (每个字节的十六进制)



$12345_{10} = 3039_{16} = 00\ 00\ 30\ 39_{16}$  对应  $39\ 30\ 00\ 00$  (4字节颠倒)

$567890_{10} = 8AA52_{16} = 00\ 08\ AA\ 52_{16}$  对应  $52\ AA\ 08\ 00$

$97.687510_{10} = (0.\underline{1}1000011011)_2 * 2^7$  的float =  $42\ C3\ 60\ 00_{16}$  对应  $00\ 60\ C3\ 42$   
 $(0\underline{1}00\ 0010\ 1\ \underline{100\ 0011\ 0110\ 0000\ 0000\ 0000})_2$

97.6875的double =  $40\ 58\ 6C\ 00\ 00\ 00\ 00\ 00$

对应  $00\ 00\ 00\ 00\ 00\ 6C\ 58\ 40$  (参考书p.42结果)

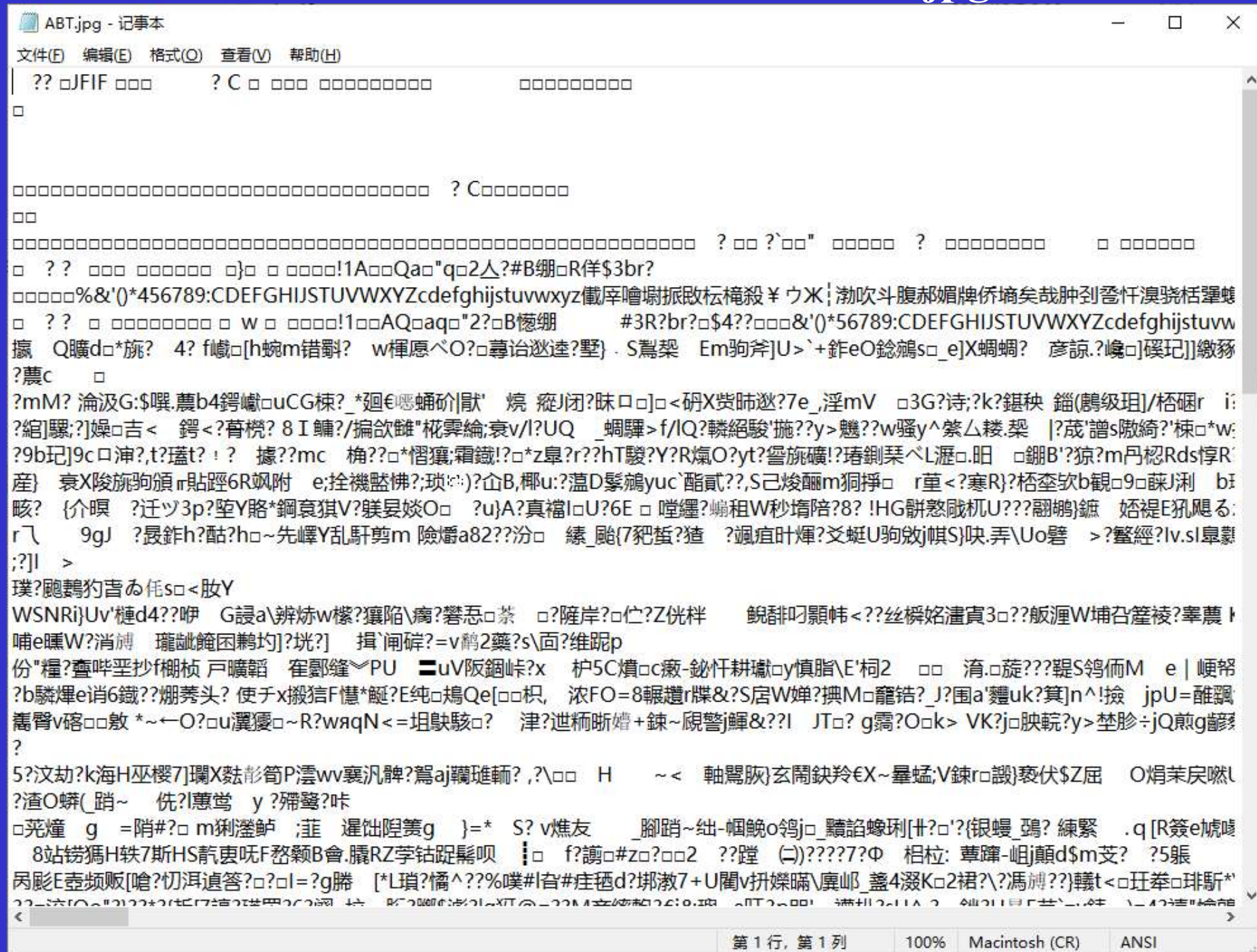
$(0\underline{1}00\ 0000\ 0101\ \underline{1000\ 0110\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000})_2$

float    1位符号位    8位阶码(定点整数-2的偏移码)    23位尾数原码(去第1位)

double   1位符号位   11位阶码(定点整数-2的偏移码)   52位尾数原码(去第1位)



## 用Windows记事本打开一个二进制文件ABT.jpg:



用图片浏览器打开文件ABT.jpg是一幅自然风景照片：





## 12.1.2 缓冲文件系统

- 在C语言中，对文件的操作都是通过库函数来实现的。

缓冲文件系统

高级文件系统

- 缓冲文件系统指系统自动为正在被使用的文件在内存中开辟输入输出缓冲区。
- 当需要从内存向外存储器中的文件输出数据时，先将数据送到为该文件开辟的输出缓冲区中，当输出缓冲区满或收到强制命令后才一起送到外存储器的文件中。
- 当需要从外存储器中的文件读入数据到内存中进行处理时，首先一次从外存储器将一批数据读入输入缓冲区中（将缓冲区填满），然后再从输入缓冲区中将数据逐个读出进行处理。
- 因此，在缓冲文件系统中，对文件的输入输出是通过为该文件开辟的输入输出缓冲区进行的，对文件中数据的处理也是在该输入输出缓冲区中进行的。

## 12.1.2 缓冲文件系统

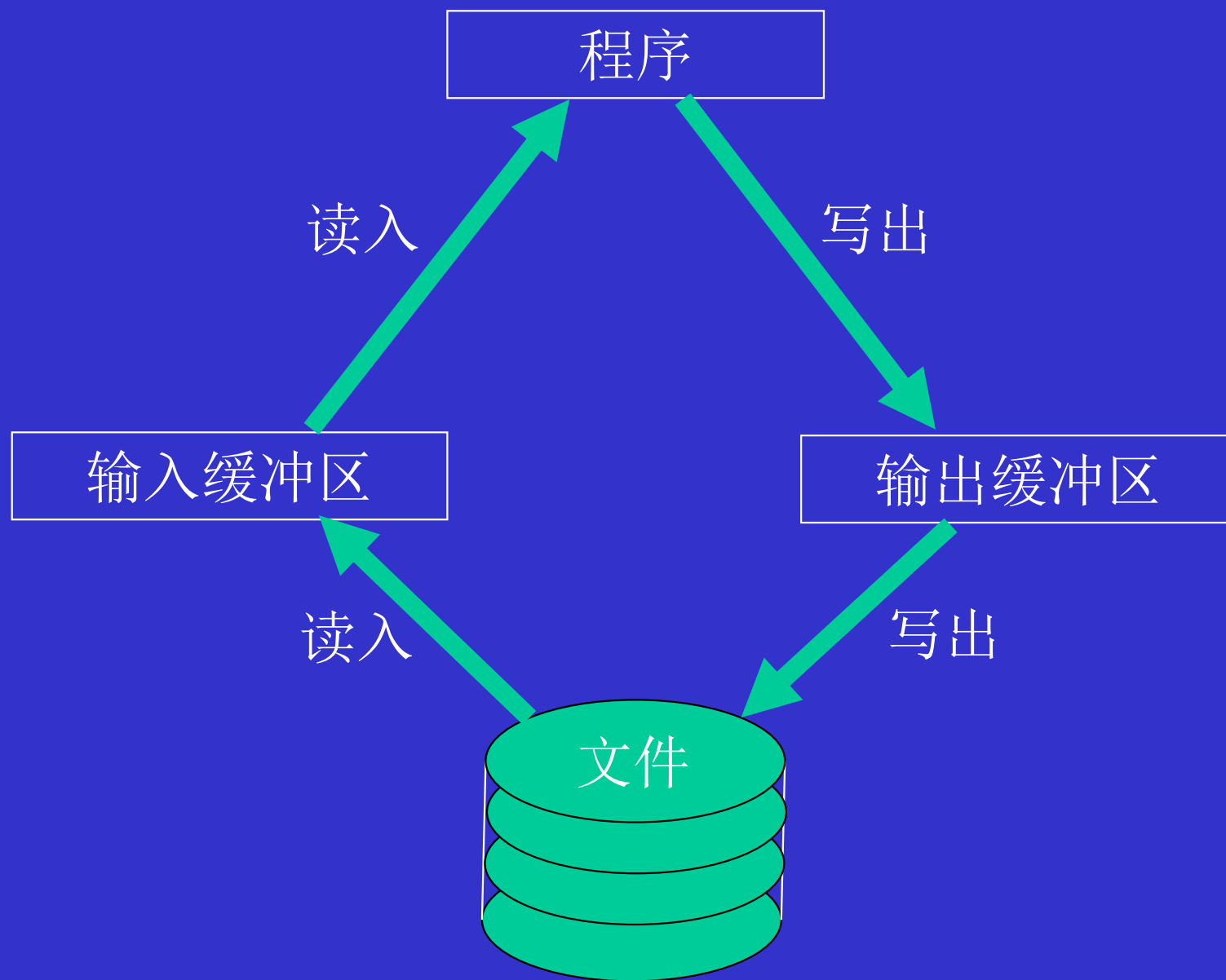
- 设置文件输入输出缓冲区的优点是：

减少程序直接进行I/O操作的次数，将每次读写一个数就进行一次I/O操作，合并为多次读写仅仅进行一次I/O操作，从而提高效率和整个程序的执行速度。

因为读写硬盘的I/O操作是机械操作，不可能每秒钟操作成千上万次，过多的I/O操作会影响CPU和整个程序的执行效率。

- 设置文件输入输出缓冲区的缺点是：

由于多次读写合并为一次I/O操作，可能会出现，要写出的数据因为先写进输出缓冲区，还没有真正写到磁盘U盘固盘等外部存储介质中，如果此时程序非正常终止，会出现输出缓冲区中的数据丢失，没有写到文件中去的现象。



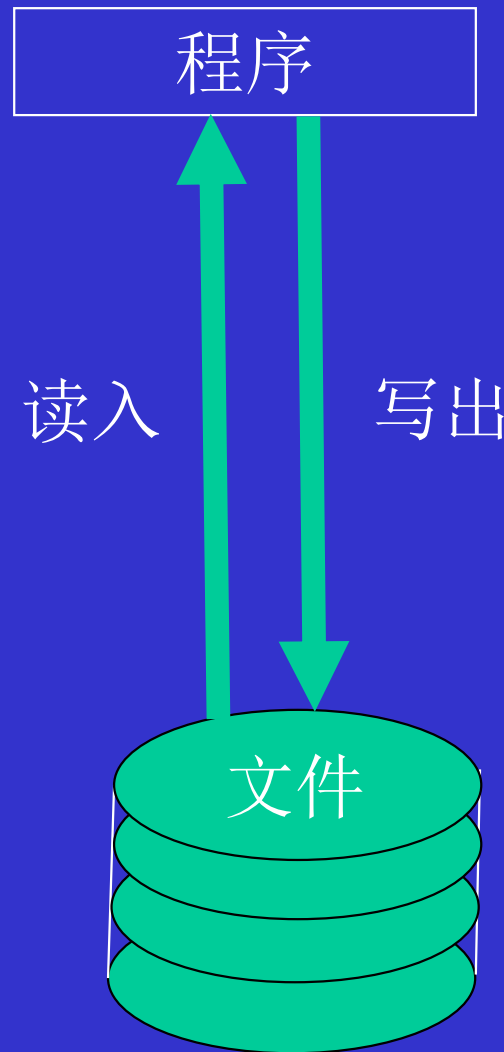
## 非缓冲文件系统

非缓冲文件系统

低级文件系统

● 是指系统不自动为文件开辟输入输出缓冲区，而是由用户程序自己为文件设定输入输出缓冲区。这种情况下，程序的每次I/O读写，都直接访问磁盘U盘等介质。

● 实例就是每次开机时，加载或恢复操作系统的过程，就是非缓冲文件操作。





### 12.1.3 文件类型指针

- 在C语言的缓冲文件系统中, 用文件类型指针来标识打开的文件

定义文件类型指针的一般形式为:

**FILE** \*指针变量名;

系统将该结构体类型定义为 **FILE** (注意: 是英文大写字母), 简称文件类型。

其中指针变量名用于指向一个文件, 实际上是指向用于存放文件数据缓冲区的首地址。

- 一般来说, 文件操作有以下三个步骤:

- 1) 打开文件。在计算机内存中开辟一个软通道(缓冲区), 用于存放被打开文件的有关信息。
- 2) 文件处理。包括在缓冲区中读写数据以及定位等操作。
- 3) 关闭文件。将缓冲区中的内容写回到外部存储器 (磁盘, 固盘, U盘等) 中, 并释放软通道(缓冲区)。

打开 stdio.h 文件，可以看到其中有一段程序：

```
struct _iobuf {  
    char *_ptr;  
    int  _cnt;  
    char *_base;  
    int  _flag;  
    int  _file;  
    int  _charbuf;  
    int  _bufsiz;  
    char *_tmpfname;  
};  
typedef struct _iobuf FILE;  
这就是FILE类型的定义。
```

## 12.2 文件的基本操作

### 12.2.1 文件的打开与关闭

#### ① 文件的打开

```
FILE *fp(或其他指针变量名);
```

```
...
```

```
fp=fopen("文件名", "文件打开方式");
```

#### **fopen( )函数**

- 文件名 ..... 要打开的文件的名字
- 文件打开方式 ..... 以何种方式打开文件

文件名 与 文件打开方式 这两者均是字符串形式

- 主要功能是为需要打开的文件分配一个缓冲区，并返回该缓冲区的首地址。

- 文件名字符串中可以带文件存储路径，例如在Windows中："C:\\data\\abc.txt"，用两个\\的转义符表示一个目录符\\

## ● 文本文件打开方式: "r", "w", "a", "r+", "w+", "a+"

- r** 只读 为读打开一个文件。若指定的文件不存在, 则返回空指针值 **NULL**。这种打开方式只能读不能写。
- w** 只写 为写打开一个新文件。若指定的文件已存在, 则其中原有内容被删去; 否则创建一个新文件, 若创建不成功, 则返回空指针值 **NULL**。这种打开方式只能写不能读。
- a** 追加写 向文件尾增加数据, 不能修改其它数据, 也不能读。若指定的文件不存在, 则创建一个新文件, 若创建不成功, 则返回空指针值 **NULL**。
- r+** 读写 为读写打开一个文件。若指定的文件不存在, 则返回空指针值 **NULL**。这种打开方式可以从文件中读入数据, 也可以写数据到文件中。
- w+** 写读 为写读打开一个新文件。若指定的文件已存在, 则其中原有内容被删去; 否则创建一个新文件, 若创建不成功, 则返回空指针值 **NULL**。写到文件中的数据, 可以移动文件指针再从文件中读入。
- a+** 读与追加写 为读与向文件尾增加数据打开一个文件。若指定的文件不存在, 则创建一个新文件, 若创建不成功, 则返回空指针值 **NULL**。

- 以上方式打开的是文本文件，也可以在后面加"**t**":

**"rt", "wt", "at", "r+t", "w+t", "a+t".....**

但通常省略"**t**"。

如果在后面附加"**b**"，则表示打开的是二进制文件，否则默认为打开的是文本文件。

- 二进制文件打开方式:

**"rb", "wb", "ab", "r+b", "w+b", "a+b"**

其中的 **"r+b", "w+b", "a+b"**

也可以写为: **"rb+", "wb+", "ab+"**



- rb** 只读 为读打开一个二进制文件。若指定的文件不存在，则返回空指针值 **NULL**。这种打开方式只能读不能写。
- wb** 只写 为写打开一个新二进制文件。若指定的文件已存在，则其中原有内容被删去；否则创建一个新文件，若创建不成功，则返回空指针值 **NULL**。这种打开方式只能写不能读。
- ab** 追加写 向二进制文件尾添加数据，不能修改其它数据，也不能读。若指定的文件不存在，则创建一个新文件，若创建不成功，则返回空指针值 **NULL**。
- r+b** 读写 为读写打开一个二进制文件。若指定的文件不存在，则返回空指针值 **NULL**。可以从文件中读入二进制数据，也可以写二进制数据到文件中。
- w+b** 写读 为写读打开一个新二进制文件。若指定的文件已存在，则其中原有内容被删去；否则创建一个新文件，若创建不成功，则返回空指针值 **NULL**。写到文件中的二进制数据，可以移动文件指针再从文件中读入。
- a+b** 读与追加写 为读与向文件尾添加数据打开一个二进制文件。若指定的文件不存在，则创建一个新文件，若创建不成功，则返回空指针值 **NULL**。可以向文件尾添加二进制数据，也可以移动文件指针读入二进制数据，但不能修改文件中已有的二进制数据。

- 在打开一个文件时，有时会出错，可能各种原因引起：文件不存在，不允许打开，不能生成新文件，..... 等等。

- 在C程序中，常采用以下方式来打开文件：

```
FILE *fp;
```

```
if ((fp=fopen("文件名", "文件打开方式")) == NULL)
```

```
{ printf("cannot open this file !\n");
```

```
    exit(0); /*终止调用过程*/
```

```
} /* 注意：由于=的执行优先级低于==，因此上面的()不能省略 */
```

- 在上述方式打开文件时，如果出现“打开”错误，**fopen()**函数返回空指针**NULL**，程序就显示以下信息：

```
cannot open this file !
```

并执行**exit**函数强制退出当前的程序执行过程。

- 打开文件也可以改写为:

```
FILE *fp;
```

```
fp= fopen("文件名", "文件打开方式");
```

```
if (fp == NULL)
```

```
{ printf("cannot open this file !\n");
```

```
    exit(0); /*终止调用过程 */
```

```
}
```

## ② 文件的关闭

```
fclose(fp);
```

- **fclose()**函数的主要功能是将由**fp**指向的缓冲区中的数据存放到外部存储器的文件中，然后释放该缓冲区。
- 当文件被关闭后，如果想再对该文件进行操作，则必须重新打开它。
- 虽然允许同时打开多个文件，但同时可以打开的文件个数是有限的。
- 因此，如果不关闭已经处理完的文件，当打开的文件个数很多时，会影响对其他文件的打开操作。
- 建议：当一个文件使用完后应立即关闭它。

## 12.2.2 文件的读写

### 1. 字符读写函数 主要适用于文本文件的读写。

#### ① 读操作

从指定的文件向程序输入数据。

#### ● 读字符函数 **fgetc()**

读字符函数的一般形式为:

**int fgetc(FILE \*fp);**

其中**fp**为文件类型的指针，指向已打开的文件。该函数的功能是，从指定的文件读入一个字符并返回（注意：返回值的类型是**int**）。如果读到文件末尾或者读取出错时返回**EOF**（值为-1）。



**【例12-1】** 从文本文件a.txt中顺序读入字符并在屏幕上显示。

```
#include <stdio.h>
#include <stdlib.h> /* 要使用exit() 函数必须引用 */
main()
{ FILE *fin;
  char c;
  if ((fin=fopen("a.dat","r"))==NULL)
  { printf("cannot open this file !\n");
    exit(0);
  }
  c=fgetc(fin); /*从文件读取一个字符*/
  while(c!=EOF) /* EOF为文件的结束标志，值为-1 */
  { putchar(c); /*在屏幕上显示字符*/
    c=fgetc(fin); /*继续从文件读取一个字符*/
  }
  fclose(fin);/*关闭文件*/
}
```

a.dat : abcdefg  
12345

运行结果: abcdefg  
12345请按任意键继续...

## 12.2.2 文件的读写

### ② 写操作

将程序中处理好的数据写到指定的文件中。

#### ● 写字符函数 **fputc()**

一般形式为：

**int fputc(int ch, FILE \*fp);**

- ✓ 其中**fp**为文件类型的指针，指向已打开的文件；这里的**ch**可以是字符型变量，也可以是字符型常量与字符型表达式。但形参**ch**的类型是**int**。
- ✓ 该函数的功能是：将一个字符写到指定的文件中。若写成功，则返回已输出的字符，否则返回文件结束标志**EOF**。

【例12-2】从键盘输入的文本原样写到名为**abc.txt**文件中，以输入字符#作为键盘输入结束标志。

C程序如右边所示，若从键盘输入：

akddndn

dldlld

asss#

则文件**abc.txt**的内容将是：

akddndn

dldlld

asss

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fout;
  char c;
  if ((fout=fopen("abc.txt", "w"))==NULL)
  { printf("cannot open this file !\n");
    exit(0);
  }
  c=getchar( ); /*从键盘输入一个字符*/
  while(c!='#') /* '#'为输入的结束标志*/
  { fputc(c, fout); /*将字符写入文件*/
    c=getchar( ); /*继续从键盘输入一个字符*/
  }
  fclose(fout); /*关闭文件*/
}
```

### ③ 读字符串函数fgets()

一般形式为:

```
char *fgets( char *string, int n, FILE *fp );
```

其中fp为文件类型的指针，指向已打开的文件；string是一个字符串指针；n是一个整型变量，也可以是整型常量或整型表达式。当读入正确时，函数返回值是与string相同的指针值，函数返回值为NULL则表示读入不成功。

#### 功能

从指定的文件读入一行字符（不超过n-1个字符）存放到由string指向的存储空间中，读入结束后，将自动在字符串最后加一个字符串结束符'\0'。

- 需要指出的是，在执行fgets()的过程中，如果在未读满n-1个字符时，就已经读到一个换行符或文件结束标志EOF，则将结束本次读操作，此时读入的字符就不够n-1个，但包括回车符也被读入到字符串string中。若fgets读入不成功，将返回NULL。

例如有程序：

```
#include <stdio.h>
#include <string.h>
main( )
{ char a[20];
  int n;
  FILE *fp;
  fp = fopen("d:\\1.txt", "r");
  for (n=0; n<3; n++)
  { fgets(a,20, fp );
    printf("%s", a);
    printf(" %d\n", strlen(a));
  }
  fclose(fp);
}
```

1. txt文件中内容为：(最后一行有回车符)  
abcdefghijklmnopqr  
abcdefghijklmnopqrstuvwxy

运行结果为：

```
abcdefghijklmnopqr
19
abcdefghijklmnopqrs 19
tuvwxyz
8
请按任意键继续...
```

结果分析：

- 1. txt中第1行18个不足19个字符，因此fgets连回车符也读入到字符串中并输出到了屏幕上。
- 第2行超过19个字符，因此fgets只读入前19个字符，fgets下一次读入了第2行剩余的字符连同回车符。



#### ④ 写字符串函数fputs()

写字符串函数的一般形式为：

```
int fputs( const char *string, FILE *fp );
```

其中**fp**为文件类型的指针，指向已打开的文件，**string**可以是一个字符串常量，也可以是一个指向字符串的指针，还可以是存放字符串的数组名。

##### 功能

将指定的字符串写到文件**fp**中。若写成功，则返回非负值；若写不成功，则返回**EOF**。

- 需要指出的是，在利用函数**fputs()**将字符串写到文件的过程中，字符串中最后的字符串结束符'\0'并不写到文件中，也不自动加换行符'\n'。因此，为了便于以后的读入，在写字符串到文件时，必要时可以人为加入如'\n'这样的字符。

例如有程序：

```
#include <stdio.h>
main( )
{ char a[20];
  int n;
  FILE *fp,*fpout;
  fp = fopen("1.txt", "r");
  fpout = fopen("2.txt", "w");
  for (n=0; n<3; n++)
  { fgets(a, 20, fp);
    fputs(a, fpout);
  }
  fclose(fp);
  fclose(fpout);
}
```

1. txt文件中内容为：(最后一行有回车符)  
abcdefghijklmnopqr  
abcdefghijklmnopqrstuvwxyz

●程序的功能是，循环用读字符串函数fgets从文件1. txt读入数据到字符串a，然后用写字符串函数fputs写入到2. txt文件中。

●实际上是文件2. txt完全复制了文件1. txt。可以通过打开2. txt查看验证。

2. txt文件中内容为：  
abcdefghijklmnopqr  
abcdefghijklmnopqrstuvwxyz  
(注：最后一行有回车符)

## 2. 数据块读写函数

主要适用于二进制文件的读写

### ① 判断文件是否结束的函数feof()

在C语言中，只有文本文件才是以读入EOF作为文件结束标志的，而二进制文件不是以EOF作为文件结束标志。为此，C语言提供了一个feof()函数，专门用来判断文件是否结束（文件指针在文件尾部）。

- 判断文件是否结束函数的一般形式为：

**int feof(FILE \*fp)**

其中fp指向已打开的文件。

### 功能

在读fp指向的文件时判断是否遇到文件结束。如果遇到文件结束，则函数feof(fp)的返回值为非0；否则返回值为0。

- feof()函数既可以用来判断二进制文件,也可以用来判断文本文件的文件指针是否到了文件尾。

使用**feof()** 改写【例12-1】：

```
#include <stdio.h>
#include <stdlib.h>
main()
{ FILE *fin;
  char c;
  if ((fin=fopen("a.txt", "r+"))==NULL)
  {   printf("cannot open this file!\n");
      exit(0);
  }
  c=fgetc(fin);    /* 从文件读取一个字符 */
  while(!feof(fin)) /* 未读到文件末尾继续循环 */
  {   putchar(c);  /* 在屏幕上显示字符 */
      c=fgetc(fin); /* 继续从文件读取一个字符 */
  }
  fclose(fin);    /* 关闭文件 */
}
```

又例如，有程序：

```
#include <stdio.h>
#include <stdlib.h>
main()
{ FILE *fin, *fout;
  char a[80];
  if ((fin=fopen("a.dat","r"))==NULL)
  { printf("cannot open this file !\n");    exit(0); }
  if ((fout=fopen("b.dat","w"))==NULL)
  { printf("cannot open this file !\n");    exit(0); }
  while( !feof(fin) )    /* 判断文件是否结束 */
  { fgets(a, 80, fin); /* 从文件读取一行字符 */
    fputs(a, fout);    /* 写到另一个文件中 */
  }
  fclose(fout); /*关闭文件*/
  fclose(fin);  /*关闭文件*/
}
```

若文本文件**a.dat**的内容是:

abcdefg

12345

998e929292

ddl1ddl1ddl1ddl1ddl1ddl1ddl1ddl1ddl1

最后一行有一个回车符。

## 所生成的输出文件**b.dat**的内容会是什么？？？？

## b.dat的内容是:

abcdefg

12345

998e929292

**ddl**

dd1dd1dd1dd1dd1dd1dd1dd1dd1

## 看起来最后一行重复了一次，为什么？

- 因为feof(fin)只有当读不成功才返回非0值，因此老是慢一拍！上面循环最后一次读不成功，但随后的fputs把上次读入的字符串又写了一次。
- 我们可以通过fgets函数的返回值是否为NULL来确定是否成功读入，读入不成功立即终止执行。因此为了避免出现最后一行重复写的问题。程序应该改为：



```
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fin, *fout;
  char a[80];
  if ((fin=fopen("a.dat","r"))==NULL)
  { printf("cannot open this file !\n");    exit(0);    }
  if ((fout=fopen("b.dat","w"))==NULL)
  { printf("cannot open this file !\n");    exit(0);    }
  while(!feof(fin) ) /* 判断文件是否结束,可以改为: while(1) */
  {   if (fgets(a, 80, fin) == NULL) /*若读不成功,立刻终止 */
      break;          /* 从文件读取一行字符 */
      fputs(a, fout);  /* 写到另一个文件中 */
  }
  fclose(fout); /*关闭文件*/
  fclose(fin); /*关闭文件*/
}
```

## ② 数据块读函数 **fread()**

### 功能

数据块读函数的功能是：从指定的文件中以二进制格式读入一组数据到指定的内存区。若成功读入，则返回实际读取到的项数（小于或等于count）；如果不成功或读到文件末尾返回0。

其形式为：**int fread(buffer, size, count, fp);**

其中：**buffer** 存放读入数据的内存首地址。

**size** 每个数据项的字节数，通常用：

**sizeof(数据项的数据类型)**

**count** 读入的数据项个数。

**fp** 文件类型指针，指向已打开的二进制文件。

**【例12-3】** 下列C程序从二进制文件b.dat中读入4个整数存放到整型数组x中，并将4个整数输出到屏幕上。

```
#include <stdio.h>
#include <stdlib.h>
main()
{ FILE *fp;
  int x[4];
  if ((fp=fopen("b.dat", "rb"))==NULL)
  { printf("cannot open this file !\n");
    exit(0);
  }
  fread(x, sizeof(int), 4, fp);
  /* 也可写为: fread(&x[0], sizeof(int), 4, fp); */
  printf("%d %d %d %d\n", x[0],x[1],x[2],x[3]);
  fclose(fp);
}
```

### ③ 数据块写函数fwrite( )

#### 功能

数据块写函数fwrite的功能是：将一组数据以二进制格式写到指定的文件中。若成功写出，则函数返回值是实际写出的项数；若函数返回值是0，则表示本次写不成功。

其形式为： **int fwrite(buffer, size, count, fp);**

其中： **buffer**     输出数据的内存首地址。

**size**            每个数据项的字节数，通常用：

**sizeof(数据项的数据类型)**

**count**        写出的数据项个数。

**fp**            文件类型指针，指向已打开的二进制文件。

**【例12-4】** 下列C程序将一维数组x中的5个元素写到二进制文件c.dat中。

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fp;
  double x[5]={1.1, 2.3, 4.5, -3.6, 9.5};
  if ((fp=fopen("c.dat", "wb"))==NULL)
  {   printf("cannot open this file !\n");
      exit(0);
  }
  fwrite(x, sizeof(double), 5, fp);
  /* 也可写为: fwrite(&x[0], sizeof(double), 5, fp); */
  fclose(fp);
}
```

### 3. 格式读写函数

#### ① fscanf( )函数

**功能** 从指定的文本文件中格式化读入数据

其形式为: **int fscanf(文件指针, 格式控制, 地址表);**

- ✓ 若成功读入, 则返回值为读入并进行格式转换且赋值的项数; 返回的值中不包括读入但没有成功进行格式转换并赋值的项数。
- ✓ 若没有任何项被正确读入, 函数值返回0。
- ✓ 若读到文件尾, 返回EOF (即-1)。

- ✓ **fscanf( )**函数与格式输入函数**scanf()**很相似，区别在于：
- ✓ **scanf( )**函数是从键盘输入数据，而**fscanf()**函数是从文件读入数据。
- ✓ 因此在**fscanf( )**函数参数中多了一个文件指针，用于指出从哪个文件读入数据。
- ✓ 由于标准输入的设备名为 **stdin**

因此： **fscanf(stdin, 格式控制, 地址表);**

完全等价于： **scanf(格式控制, 地址表);**

- ✓ 也表示从键盘输入数据。



**【例12-5】** 下列C程序从文本文件ABC.txt中按格式读入两个整数，分别赋给整型变量a与b。

```
#include <stdio.h>
#include <stdlib.h>
main()
{ FILE *fp;
  int a, b;
  if ((fp=fopen("ABC.txt", "r"))==NULL)
  {   printf("cannot open this file !\n");
      exit(0);
  }
  fscanf(fp, "%d%d", &a, &b);
  printf("%d %d\n", a, b);
  fclose(fp);
}
```

ABC.txt文件内容:

12

34

运行结果:

12 34

请按任意键继续...

上面的打开方式"r"也可以写为: "rt" 或 "r+t"或 "rt+"

## ② fprintf( )函数

**功能** 格式化写数据到指定的文本文件中。

其形式为: **int fprintf**(文件指针, 格式控制, 输出表)

- ✓ 若成功写出, 则返回值为写出的项数。
- ✓ 返回负数表示写失败。
- ✓ **fprintf( )**函数与格式输出函数**printf( )**相似, 区别就在于:
- ✓ **printf( )**函数是将数据输出到显示屏幕上, 而**fprintf( )**函数是将数据写到文件中。
- ✓ 因此在**fprintf( )**函数参数中多了一个文件指针, 用于指出将数据写到哪个文件中。

✓ 标准输出的设备名为 **stdout**

因此: **fprintf(stdout, 格式控制, 地址表);**

完全等价于: **printf(格式控制, 地址表);**

✓ 表示将数据输出到显示屏幕上。

● 必须指出的是, **fprintf()** 函数与**fscanf()**函数是对应的, 即在使用**fscanf()**函数从文件读数据时, 其格式应与用**fprintf()**函数将数据写到文件时的格式一致, 否则将会导致读写错误。

【例12-6】 下列C程序将一个整数与一个双精度实数按格式存放  
到文本文件AB. txt中。

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fp;
  int a=10;
  double x=11.4;
  if ((fp=fopen("AB.txt", "w+"))==NULL)
  { printf("cannot open this file!\n");
    exit(0);
  }
  fprintf(fp, "%d,%lf", a, x);
  fclose(fp);
}
```

输出结果是什么？ 用文本编辑器打开AB.txt文件，其内容是：  
10,11.400000

文本文件格式化输入输出综合应用实例：

下面的程序从键盘上读入10个浮点数写入到文件data.txt中，然后再从文件data.txt中逐个读入这10个浮点数写出到屏幕上。

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

```
main( )
```

```
{ double c[10],x;
```

```
FILE *fp;
```

```
int i;
```

```
for (i = 0; i < 10; i++)
```

```
    scanf("%lf", &c[i]);
```

```
fp = fopen("data.txt", "w+");
```

```
for (i = 0; i < 10; i++)
```

```
    fprintf(fp, "%lf", c[i]);
```

```
fseek(fp,0,SEEK_SET);
```

```
for (i = 1; i <= 10; i++)
```

```
{ fscanf(fp, "%lf", &x);
```

```
    printf("%lf ", x);
```

```
}
```

```
fclose(fp);
```

```
}
```

运行结果为:

1 2 3 4 5 6 7 8 9 10

1.000000 0.000000 0.000000 0.000000 0.000001 0.000001 0.000001  
0.000001 0.000000 0.000000 请按任意键继续...

结果令人匪夷所思。如果你此时打开文件data.txt, 看到的将是:

1.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000  
009.00000010.000000

10个实数首尾相连, 这是因为fprintf(fp, "%lf", c[i]); 输出时没考虑输出格式导致的, 这导致后面的fscanf(fp, "%lf", &x);无法正确读入这10个数, 只能按小数点分割数字串, 实际读入的是:

1.0000002 .0000003 .0000004 .0000005 .0000006 .0000007 .0000008 .  
0000009 .00000010 .000000

因为printf("%lf ", x);缺省输出6位小数, 最后一位四舍五入, 才输出了上面看起来很诡异的结果。 如果改为 printf("%.7lf ", x);

输出7位小数, 运行结果将变成:

1 2 3 4 5 6 7 8 9 10

1.00000002 0.00000003 0.00000004 0.00000005 0.00000006 0.00000007  
0.00000008 0.00000009 0.00000001 0.00000000 请按任意键继续...

因此必须牢记：

用fprintf向文本文件中写入数据时，在输出格式中必须加空格、回车、制表符等分隔符，这样前后两个数据才不会黏连在一起，才能保证从此文本文件中能再次正确读入数据。

上面程序中改为：

```
fprintf(fp, "%lf ", c[i]); // 输出格式中加空格
```

或 

```
fprintf(fp, "%lf\n", c[i]); // 输出格式中加回车符
```

程序才能得到正确运行结果：

1 2 3 4 5 6 7 8 9 10

1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000  
8.000000 9.000000 10.000000 请按任意键继续...



### 12.2.3 文件的定位

#### ① `rewind()`函数

**功能** 将文件的读写指针移动到文件的开头。

其形式为: `void rewind(FILE *fp);`

其中**fp**是已经打开的文件指针, 此函数没有返回值。

---

#### ② `fseek()`函数

其形式为: `int fseek( FILE *stream, long offset, int origin );`

**功能** 将文件的读写指针移动到指定的位置。

✓ 其中：

*stream* 为 文件指针

*offset* 为 偏移量

*origin* 为 起始位置

✓ 起始位置是指移动文件读写指针的参考位置，它有以下三个值：

SEEK\_SET 或 0 表示从文件首（开头）

SEEK\_CUR 或 1 表示从当前读写的位置

SEEK\_END 或 2 表示从文件尾

✓ 偏移量是指以“起始位置”为基点，文件指针向后或向前移动的字节数，正整数表示向后移动，负整数表示向前移动。偏移量的类型要求为长整型。

关于fseek的使用，有程序：

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

```
#include <stdlib.h>
```

```
main( )
```

```
{ FILE *fp; int k;
```

```
double x[5]={1.1, 2.3, 4.5, -3.6, 9.5}, y[6]={0};
```

```
if ((fp=fopen("cc.dat","w+b"))==NULL) /* 以二进制写读方式打开文件 */
```

```
{ printf("cannot open this file !\n"); exit(0); }
```

```
fwrite(x, sizeof(double), 5, fp);
```

```
fseek(fp, 0L, SEEK_SET);
```

```
fread(&y[0], sizeof(double), 2, fp);
```

```
fseek(fp, -4L*(long)sizeof(double), SEEK_END);
```

```
fread(&y[2], sizeof(double), 2, fp);
```

```
fseek(fp, - (long)sizeof(double), SEEK_CUR);
```

```
fread(&y[4], sizeof(double), 2, fp);
```

```
fclose(fp);
```

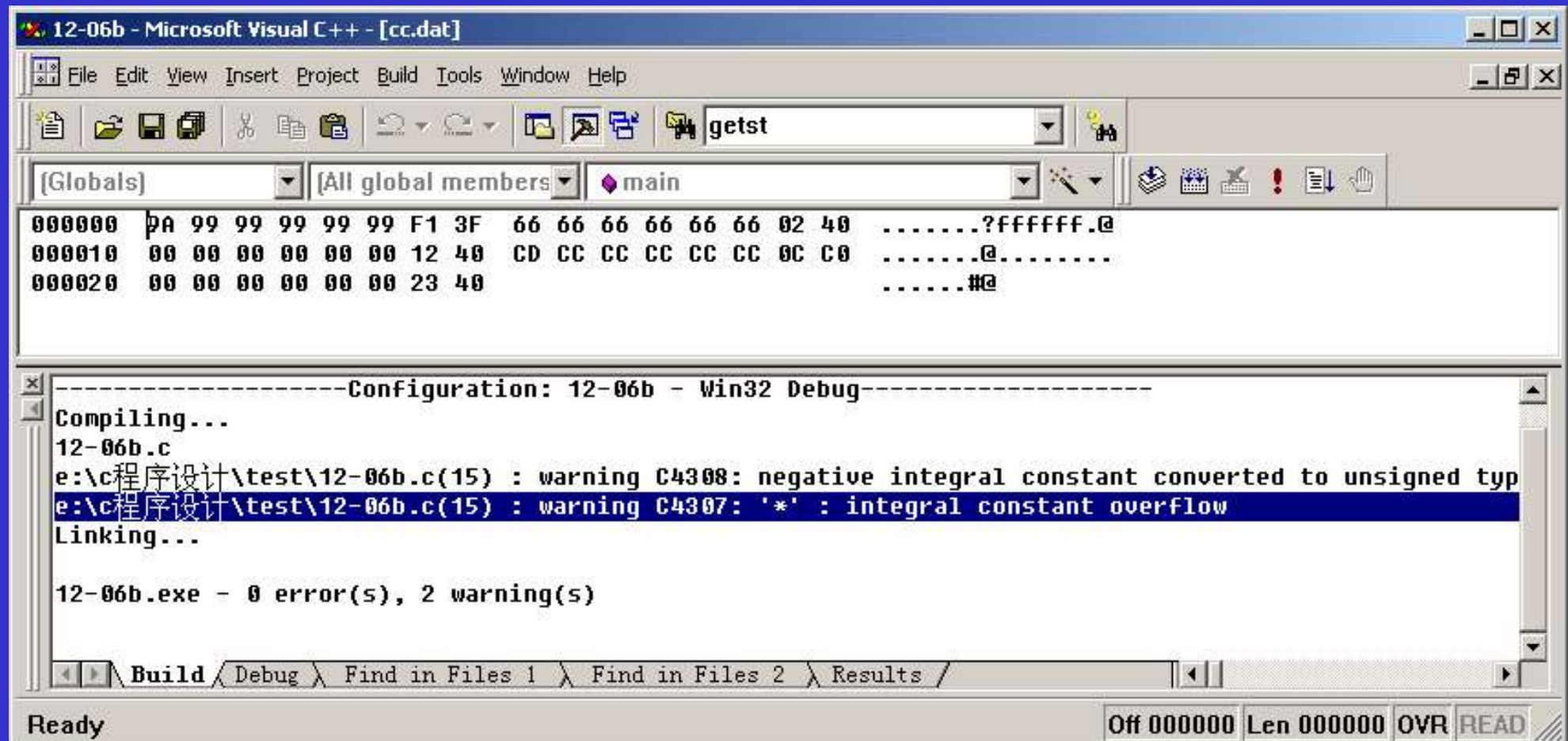
```
for (k=0; k<6; k++) printf("%4.1f ", y[k]);
```

```
}
```

运行结果:

1.1 2.3 2.3 4.5 4.5 -3.6 请按任意键继续...

cc.dat文件中数据为二进制的: 1.1, 2.3, 4.5, -3.6, 9.5



```
12-06b - Microsoft Visual C++ - [cc.dat]
File Edit View Insert Project Build Tools Window Help
[Globals] [All global members] main
00000000 0A 99 99 99 99 99 F1 3F 66 66 66 66 66 02 40 .....?ffffff.@
00000100 00 00 00 00 00 00 12 40 CD CC CC CC CC CC 0C C0 .....@.....
00000200 00 00 00 00 00 00 23 40 .....#@

-----Configuration: 12-06b - Win32 Debug-----
Compiling...
12-06b.c
e:\c程序设计\test\12-06b.c(15) : warning C4308: negative integral constant converted to unsigned type
e:\c程序设计\test\12-06b.c(15) : warning C4307: '*' : integral constant overflow
Linking...

12-06b.exe - 0 error(s), 2 warning(s)

Build Debug Find in Files 1 Find in Files 2 Results /
Ready Off 000000 Len 000000 OVR READ
```

## 注意:

(1) 关于fseek的返回值: 如果成功, fseek返回0, 否则, 返回一个非零的值。对于那些不能重定位的设备, fseek的返回值是不确定的。

(2) 可以在一个文件中用fseek把指针重定位在任何地方, 甚至文件指针可以定位到文件结束符之外, fseek将清除文件结束符, 并忽略先前ungetc调用对输入输出流的作用。

(3) 若文件是以追加方式被打开的, 那么当前文件指针的位置是由上一次I/O 操作决定的, 而不是由下一次写操作在那里决定的。若一个文件是以追加方式打开的, 至今还没有进行I/O操作, 那么文件指针是在文件的开始之处。

(4) 对于以文本方式打开的文件，fseek的用途是有限的。因为carriage return–linefeed 的转换，会使得fseek产生预想不到的结果。在文本方式下，fseek操作结果确定的是：

1) 相对于任何起始位置，重定位的位移值是0；

2) 从文件起始位置(SEEK\_SET)进行重定位，而位移的值是用ftell函数返回的值。

● carriage return–linefeed 的转换：Windows中对于以文本方式打开的文件，若向文件中写入回车符\n, 其ASCII值是0x0D，Windows系统会自动加一个换行符0x0A。但二进制方式下如果向文件写回车符\n(0x0D), Windows系统不会自动加换行符0x0A。但打开这个文件时，很多文本编辑器会自动加上换行符0x0A，因此给人的感觉就是文本文件中，0D 0A总是成对出现。

(5) 同样在文本方式下，CTRL+Z 在输入时被当作文件结束符。对于打开进行读写的文件，fopen和所有相关的函数都在文件尾部检测CTRL+Z字符，如果可能就删除。这么做的原因是，用fseek和ftell在某个文件中移动文件指针时，若这个文件是用CTRL+Z 标记结束的，可能会使得fseek在靠近文件尾部时，行为失常（定位变得不确切）。

● 特别提示：向文件中写数据就像向磁带中录歌曲一样，同一个位置上，后写入的数据将覆盖并抹去以前的数据，不可能在原来的数据前插入数据，新老数据也不可能在同一个位置上同时存在。



有程序:

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fp; int k;
  double x[5]={1.1, 2.3, 4.5, -3.6, 9.5}, y[5]={0};
  if ((fp=fopen("cc.dat","w+b"))==NULL)
  { printf("cannot open this file !\n"); exit(0); }
  fwrite(x, sizeof(double), 5, fp);
  fseek(fp, 0L, SEEK_SET);
  fwrite(&x[2], sizeof(double), 3, fp);
  fseek(fp, 0L, SEEK_SET);
  fread(y, sizeof(double), 5, fp);
  fclose(fp);
  for (k=0; k<5; k++)
    printf("%4.1f ", y[k]);
}
```

运行结果: 4.5 -3.6 9.5 -3.6 9.5 请按任意键继续...

### ③ ftell()函数

**功能** 返回文件的当前读写位置（出错返回-1）。

其形式为: **long ftell( FILE \*fp)**

使用ftell函数的程序示例:

```
#include <stdio.h>
main()
{ FILE *fp;
  char *p="Hello!";
  fp = fopen("11.txt", "a");
  printf("ftell = %d\n", ftell(fp));
  fprintf(fp, "%s\n", p);
  printf("ftell = %d\n", ftell(fp));
  fclose(fp);
}
```

第1次运行结果是:

```
ftell = 0
```

```
ftell = 8
```

第2次运行结果是:

```
ftell = 0
```

```
ftell = 16
```

第3次运行结果是:

```
ftell = 0
```

```
ftell = 24
```

● 每次运行结果不尽相同的原因是，每次运行都用追加方式向文件11.txt中写入字符串“Hello!”以及回车符和换行符，文件11.txt会变得越来越长，每次追加写完后文件指针所在的位置都是文件尾，但数值各不相同。

有程序:

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

```
#include <stdlib.h>
```

```
main( )
```

```
{ FILE *fp;
```

```
double x[5]={1,2,3,4,5}, y[5]={0}; int k;
```

```
if((fp=fopen("a.dat","a+b"))==NULL)
```

```
{ printf("cannot open a.dat!\n"); exit(0); }
```

```
printf("ftell=%d\n", ftell(fp));
```

```
fwrite(x,sizeof(double),5,fp);          printf("ftell=%d\n", ftell(fp));
```

```
fseek(fp,0L,SEEK_SET);
```

```
fread(&y[0],sizeof(double),2,fp);  printf("ftell=%d\n", ftell(fp));
```

```
fwrite(x,sizeof(double),5,fp);      printf("ftell=%d\n", ftell(fp));
```

```
fseek(fp,-4L*(long)sizeof(double),SEEK_END);
```

```
fread(&y[2],sizeof(double),2,fp);  printf("ftell=%d\n", ftell(fp));
```

```
fclose(fp);
```

```
for (k=0; k<5; k++)  printf("%f ", y[k]);
```

```
}
```

第1次运行结果是：

`ftell=0`

`ftell=40`

`ftell=16`

`ftell=40`

`ftell=24`

1. 000000 2. 000000 2. 000000 3. 000000 0. 000000 请按任意键继续. . .

此时文件长度为40个字节。

第2次运行结果是：

`ftell=0`

`ftell=80`

`ftell=16`

`ftell=56`

`ftell=64`

1. 000000 2. 000000 2. 000000 3. 000000 0. 000000 请按任意键继续. . .

此时文件长度为80个字节。

● 每次运行时，第二个 `fwrite(x,sizeof(double),5,fp);` 看起来没起任何作用。原因是：

● 因为文件打开方式是"a+b", 这种方式可在文件尾部追加写也可读入。但当写入数据后执行:

```
fseek(fp,0L,SEEK_SET);  
fread(&y[0],sizeof(double),2,fp);  
printf("ftell=%d\n", ftell(fp));
```

在读入操作后, 此时立即执行写操作:

```
fwrite(x,sizeof(double),5,fp);  
printf("ftell=%d\n", ftell(fp));
```

由于没有刷新使得输入输出缓冲区同步, 因此这5个double数没有写入到文件中, 或者说: 文件写入不成功, 文件长度仍然是40个字节。ftell告诉你的是此时文件指针在文件尾部的位置, 所以是40。

● 因此, 若文件打开方式是"a+b", 在执行读操作后, 如果想成功写入, 需要先用fseek或rewind移动文件指针 (位置不限, 可以原地不动) ! 但要注意, 此时用fflush并不起作用。

把程序的第二个fwrite前加入语句:

```
fseek(fp,0L,SEEK_CUR); 或者  rewind(fp);
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fp;
  double x[5]={1,2,3,4,5},y[5]={0}; int k;
  if((fp=fopen("a.dat","a+b"))==NULL)
  { printf("cannot open a.dat!\n");  exit(0);  }
  printf("ftell=%d\n", ftell(fp));
  fwrite(x,sizeof(double),5,fp);      printf("ftell=%d\n", ftell(fp));
  fseek(fp,0L,SEEK_SET);
  fread(&y[0],sizeof(double),2,fp);  printf("ftell=%d\n", ftell(fp));
  fseek(fp, 0L, SEEK_CUR); /* 文件指针保持原地不动 */
  fwrite(x,sizeof(double),5,fp);      printf("ftell=%d\n", ftell(fp));
  fseek(fp,-4L*sizeof(double),SEEK_END);
  fread(&y[2],sizeof(double),2,fp);  printf("ftell=%d\n", ftell(fp));
  fclose(fp);
  for (k=0; k<5; k++)  printf("%f ", y[k]);
}
```

删掉a.dat文件，重新开始执行，第**1**次运行结果是：

ftell=0

ftell=40

ftell=16

ftell=80

ftell=64

1.000000 2.000000 2.000000 3.000000 0.000000 请按任意键继续...

此时文件长度为**80**个字节，说明第二个fwrite写入成功了！

第**2**次运行结果是：

ftell=0

ftell=120

ftell=16

ftell=160

ftell=144

1.000000 2.000000 2.000000 3.000000 0.000000 请按任意键继续...

此时文件长度为**160**个字节。



把程序修改一下：先写入5个double数，然后移动文件指针到文件开始处，再读入这5个double数，此时文件指针应该就停留在文件尾部，再写入5个double数。

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ FILE *fp;
  double x[5]={1,2,3,4,5},y[5]={0}; int k;
  if((fp=fopen("a.dat","a+b"))==NULL)
  { printf("cannot open a.dat!\n"); exit(0); }
  printf("ftell=%d\n", ftell(fp));
  fwrite(x,sizeof(double),5,fp);          printf("ftell=%d\n", ftell(fp));
  fseek(fp,0L,SEEK_SET);
  fread(&y[0],sizeof(double),5,fp);        printf("ftell=%d\n", ftell(fp));
  fwrite(x,sizeof(double),5,fp);           printf("ftell=%d\n", ftell(fp));
  fseek(fp,-4L*sizeof(double),SEEK_END);
  fread(&y[2],sizeof(double),2,fp);        printf("ftell=%d\n", ftell(fp));
  fclose(fp);
  for (k=0; k<5; k++) printf("%f ", y[k]);
}
```

删掉a.dat文件，重新开始执行，运行结果是：

ftell=0

ftell=40

ftell=40

ftell=40

ftell=24

1.000000 2.000000 2.000000 3.000000 5.000000 请按任意键继续...

此时文件长度为40个字节，说明第二个fwrite写入不成功。

- 这说明在读操作之后，即使文件指针已经移到了文件尾也不能写入，需要在读操作之后，通过移动文件指针，让输入输出缓冲区同步后，才能进行写操作。

- 结论：读操作之后即使文件指针已经在文件尾部，也需要执行fseek或rewind后，才能正确追加写操作。因为输入和输出不是同一个缓冲区。

## 12.2.4 文件缓冲区的清除

### ● **fflush()**函数

其形式为:

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

### 功能

函数**fflush**的功能是：清空文件的输入输出缓冲区，使文件的输入输出缓冲区中的内容立刻输出到实际的文件中。若成功执行，则函数返回值为0，若出错，则函数返回值为-1。

下面通过一个例子，说明**fflush**函数的用途。

有程序：

```
#include <stdio.h>
#include <stdlib.h>
void main( )
{ FILE *fp;
  char c[21];
  int i;
  if((fp=fopen("r.txt","w+"))==NULL)
  { printf("cannot open this file!");
    exit(0);
  }
  for(i=0; i<2; i++)
    fputs("1234567890", fp);
  fseek(fp, -18L, SEEK_CUR );
  fputs(" ", fp);
  fgetc(c,20, fp);
  c[20] = '\0';
  puts(c);
  fclose(fp);
}
```

- 程序的运行结果是:

**2345678901234567890**

- 如果此时打开文件**r.txt**，看到的内容是:

**12 2345678901234567890**

- 按照程序的操作，应该向文件**r.txt**输出了2次字符串**"1234567890"**，**r.txt**中应该有**20**个字符。

- **fseek(fp,-18L, SEEK\_CUR);**应该使得文件指针指向第3个字符'3'的开始处。

- 随后**fputs (" ", fp);**写出一个空格字符，空格字符应该覆盖掉第3个字符 '3'，此时文件指针指向第4个字符'4'。

- **fgets(c,20, fp);**应该读入字符串**"45678901234567890"**

- 但输出结果不是这样，同样文件**r.txt**中的内容也不是我们期待的：**12 45678901234567890**

- 出现这个问题的原因是， `fputs(" ", fp);`向文件`r.txt`中写数据，并不是立刻写到磁盘上的文件中，而是停留在文件输出缓冲区中，文件输出缓冲区中的数据只有当输出缓冲区满，或收到强制写出指令时，才会真正写到磁盘上的文件中。

- 如果此时立刻用`fgets`从文件输入缓冲区读入，会造成输入输出缓冲区与磁盘文件的不一致，导致混乱。

- 因此需要在使用`fgets`前，用`fflush(fp)`清空缓冲区，使得输出缓冲区的内容写到磁盘上，这样结果才会保证正确。

- 上面的程序修改为：

```
#include <stdio.h>
#include <stdlib.h>
void main( )
{ FILE *fp;
  char c[21];
  int i;
  if((fp=fopen("r.txt","w+"))==NULL)
  { printf("cannot open this file!");
    exit(0);
  }
  for(i=0; i<2; i++)
    fputs("1234567890", fp);
  fseek(fp, -18L, SEEK_CUR );
  fputs(" ", fp);
  fflush(fp); /* 关键是需要fflush清空缓冲区，让数据同步 */
  fgets(c,20, fp);
  c[20] = '\0';
  puts(c);
  fclose(fp);
}
```

- 程序的运行结果是:

**45678901234567890**

- 再打开文件**r.txt**，看到的内容将是:

**12 45678901234567890**

- 现在结果完全正确了。除了**fflush**函数外，通常**fseek()**、**rewind()**、**fclose()**也具备清空缓冲区的能力。例如，上例中的**fflush(fp);**语句如果用**fseek(fp, 0L, SEEK\_CUR);**代替，结果也是正确的。而**fseek(fp, 0L, SEEK\_CUR);**并没有真正移动指针位置，但此操作把缓冲区的内容清空并让输入输出缓冲区同步了。

- 注意：用**scanf()**、**getchar()**等从键盘输入时，也可以用**fflush(stdin)**清空标准输入缓冲区，防止读入数据后遗留回车等控制字符对下一次读数据产生副作用。



## 12.2.5 文件指针错误状态的清除

### ● **clearerr()**函数

其形式为: `void clearerr(FILE *fp);`

#### 功能

函数**clearerr**的功能是，清除由于读写等操作失败引起文件输入输出缓冲区处于的错误状态，以保证其后其他文件操作函数的正确使用。

下面通过一个例子，说明**clearerr**函数的用途。

【例12-7】编写程序，从键盘上输入年月，告诉这个月有多少天。

```
#include<stdio.h>
main( )
{   int year,month,flag=0;
    printf("Please input : year-month = ?\n");
    flag = scanf("%d-%d",&year,&month);
    while (!flag || year<=0 || month<=0 || month>12)
    {   printf("ERROR! Please input again! year-month = ?\n");
        flag= scanf("%d-%d",&year,&month);
    }
    switch(month)
    { case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        printf("There are 31 days in this month.\n"); break;
      case 4: case 6: case 9: case 11:
        printf("There are 30 days in this month.\n"); break;
      case 2:
        flag = 0;
        if ((year%4==0 && year%100!=0) || year%400==0)
            flag=1;
        printf("There are %d days in this month.\n", 28+flag);
    }
}
```

## 此程序运行时，提示：

Please input : year-month = ?

- 如果此时输入：2022-12

不会有任伺问题。

- ## ● 但如果不小心输入了：20.22-12

## 结果会是:

## 进入死循环！

- ## ● 原因是：

**%d格式遇到小数点时产生错误，无法正常读入，scanf的返回值为0，而且20.22-12一直残留在输入缓冲区中，下次再读还会遇到，继续出错！**

- ## ● 加上fflush(stdin)清除输入缓冲区:

[illegible]

```
#include<stdio.h>
main( )
{   int year,month,flag=0;
    printf("Please input : year-month = ?\n");
    flag = scanf("%d-%d",&year,&month);
    while (!flag || year<=0 || month<=0 || month>12)
    {   fflush(stdin);
        printf("ERROR! Please input again! year-month = ?\n");
        flag= scanf("%d-%d",&year,&month);
    }
    switch(month)
    { case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        printf("There are 31 days in this month.\n"); break;
      case 4: case 6: case 9: case 11:
        printf("There are 30 days in this month.\n"); break;
      case 2:
        flag = 0;
        if ((year%4==0 && year%100!=0) || year%400==0)
            flag=1;
        printf("There are %d days in this month.\n", 28+flag);
    }
}
```

- 再次编译运行，此程序运行时，提示：

Please input : year-month = ?

- 如果此时输入：20.22-12

在Win7操作系统的VS2008上结果仍然会是死循环！

(但在Win11+VS2012或WinXP+VS2008上运行不再死循环)

说明fflush(stdin)不一定起作用。

- 原因是：由于scanf没有被正确执行，使得标准输入流stdin此时处于错误状态，任何试图对stdin的操作都失效了。

- 此时必须先清除标准输入流stdin的错误状态，才能再去清除标准输入缓冲区的内容。

- 加上：clearerr(stdin);

运行结果：

Please input : year-month = ?

20.22-12

ERROR! Please input again! year-month = ?

```

#include <stdio.h>
main( )
{
    int year,month,flag=0;
    printf("Please input : year-month = ?\n");
    flag = scanf("%d-%d",&year,&month);
    while (!flag || year<=0 || month<=0 || month>12)
    {
        clearerr(stdin);
        fflush(stdin);
        printf("ERROR! Please input again! year-month = ?\n");
        flag= scanf("%d-%d",&year,&month);
    }
    switch(month)
    { case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        printf("There are 31 days in this month.\n"); break;
      case 4: case 6: case 9: case 11:
        printf("There are 30 days in this month.\n"); break;
      case 2:
        flag = 0;
        if ((year%4==0 && year%100!=0) || year%400==0)
            flag=1;
        printf("There are %d days in this month.\n", 28+flag);
    }
}

```

结论:

● 为了防止在某些操作系统上 `fflush(stdin);` 不起作用，但在另外的操作系统上又可能起作用，造成结果不确定，建议：

`clearerr(stdin);` 应该和 `fflush(stdin);` 配对使用。

● 注意：必须先清除错误状态，再刷新！

```
clearerr(stdin); /* 对标准输入 */  
fflush(stdin);
```

或：

```
clearerr(fp); /* 对从文件输入 */  
fflush(fp);
```

## 12.3 程序举例

【例12-8】 统计文件letter.txt中的字符个数。

C程序如下：

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ long count=0; char c;
  FILE *fp;
  if ((fp=fopen("letter.txt","r+"))==NULL)
  { printf("cannot open this file!\n");
    exit(0);
  }
  c = fgetc(fp);
  while(c != EOF)
  { count++;
    c = fgetc(fp);
  }
  printf("count=%ld\n", count);
  fclose(fp);
}
```

若文件letter.txt内容为：

kssjsk

12344

4444

(注意：最后一行没有回车符)

则运行结果为：

count=17

请按任意键继续...



**【例12-9】** 下列C程序的功能是，用追加的形式打开文件gg.txt，查看文件读写指针的位置，然后向文件写入"data"，再查看文件读写指针的位置。

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ long p;
  FILE *fp;
  if ((fp=fopen("gg.txt", "a"))==NULL)
  { printf("cannot open this file!\n");
    exit(0);
  }
  p=ftell(fp);
  printf("p=%ld\n", p);
  fprintf(fp, "data");
  p=ftell(fp);
  printf("p=%ld\n", p);
  fclose(fp);
}
```

● 若文本文件gg.txt内容为:

ajjaj

skadkkd

ldldlld

● 运行结果:

p=0

p=29

请按任意键继续...

● 此时文本文件gg.txt内容为:

ajjaj

skadkkd

ldldlld

data

【例12-10】 下面的C程序将程序中的10个学生的信息写到文件student.dat中。

```
#include <stdio.h>
#include <stdlib.h>
typedef struct student
{int num; char name[8]; char gender; int age; double score;} STU;
STU s[10]={ {101,"Zhang", 'M', 19, 95.6},
            {102,"Wang", 'F', 18, 92.4}, {103,"Zhao", 'M', 19, 85.7},
            {104,"Li", 'M', 20, 96.3}, {105,"Guo", 'M', 19, 90.2},
            {106,"Lin", 'M', 18, 91.5}, {107,"Ma", 'F', 17, 98.7},
            {108,"Zhen", 'M', 21, 90.1}, {109,"Xu", 'F', 19, 89.8},
            {110,"Mao", 'M', 18, 94.9}};

main( )
{ FILE *fp;
  if ((fp=fopen("student.dat", "wb"))==NULL) // 或 "w+b"
  { printf("cannot open this file !\n"); exit(0); }
  fwrite(stu, sizeof(struct student), 10, fp);
    /* 将10个人的信息一次性写到文件student.dat中 */
  fclose(fp);
}
```

● 此时如果你去查看一下文件student.dat的属性，会发现结果文件长度是320字节。而不是 $25 \times 10 = 250$ 字节。原因是自动强制对齐问题。

● 对于结构体：

```
struct student
{
    int num;
    char name[8];
    char gender;
    int age;
    double score;
}
```

num	name
name	gender <空3个字节>
age	<空4个字节>
double	

● 因为其中单个最长的成员double是8字节，因此结构体自动以8字节对齐。int num占第一个8字节的前4字节。char name[8];占第一个8字节的后4字节，第二个8字节的前4字节。char gender;占第二个8字节的第5字节，后面空闲3个字节。int age;占第三个8字节的前4字节，后面空闲4个字节。double score;占第4个8字节，因此每个student结构体占32个字节。

打印每一个成员的起始地址验证存放方式:

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

```
struct student
```

```
{ int num;  
  char name[8];  
  char gender;  
  int age;  
  double score;
```

```
};
```

```
main( )
```

```
{ struct student stu;  
  printf("sizeof(struct student)=%d\n",sizeof(struct student));  
  printf(" num=%p\n", &stu.num);  
  printf(" name=%p\n", stu.name);  
  printf("gender=%p\n", &stu.gender);  
  printf("  sno=%p\n", &stu.age);  
  printf("score=%p\n", &stu.score);  
}
```

运行结果是:

sizeof(struct student)=32

num=3CC5F76C

name=3CC5F770

gender=3CC5F778

sno=3CC5F77C

score=3CC5F784

请按任意键继续...

注意: 不同计算机不同时刻  
的运行结果都可能不相同。

- 若文件开头加上

`#pragma pack(4)`

文件student.dat大小将是280字节。因为按4字节对齐，只有char gender;占一个4字节（其中后面空闲3个字节），每个student结构体占28个字节。

- 若文件开头加上

`#pragma pack(2)`

文件student.dat大小将是260字节。因为按2字节对齐，只有char gender;占一个2字节（其中后面空闲1个字节），每个student结构体占26个字节。

- 若文件开头加上

`#pragma pack(1)`

文件student.dat大小将是250字节，因为按1字节对齐，每个student结构体占25个字节。

**【例12-11】** 下列C程序将文件student.dat中10个学生的信息显示输出。

```
#include <stdio.h>
#include <stdlib.h>
typedef struct student
{ int num;
  char name[8];  char gender;
  int age;       double score;
} STU;
main( )
{ int i;
  STU stu[10];
  FILE *fp;
  if ((fp=fopen("student.dat", "rb"))==NULL)
  { printf("cannot open student.dat!\n"); exit(0); }
  fread(stu, sizeof(STU), 10, fp);
  fclose(fp);
  printf("No.   Name   Gender Age Score\n");
  for (i=0; i<=9; i++)
    printf("%-8d%-9s%-8c%-8d%-5.2f\n",
           stu[i].num, stu[i].name, stu[i].gender, stu[i].age, stu[i].score);
}
```

## ● 运行结果:

No.	Name	Gender	Age	Score
101	Zhang	M	19	95.60
102	Wang	F	18	92.40
103	Zhao	M	19	85.70
104	Li	M	20	96.30
105	Guo	M	19	90.20
106	Lin	M	18	91.50
107	Ma	F	17	98.70
108	Zhen	M	21	90.10
109	Xu	F	19	89.80
110	Mao	M	18	94.90

请按任意键继续...

● 注意: 读文件时, 结构体的对齐方式必须与写出student.dat时一致! #pragma pack中的大小必须一样, 否则无法正确读入数据。



## ● 关于文件操作最后强调几点：

- 1、 用二进制方式(打开方式中带b)打开的文件，只能用fread和fwrite二进制读写函数进行读写，而不能用文本操作的函数进行读写。
- 2、 用文本方式打开的文件，不要用fread和fwrite二进制读写函数进行读写，而应该用专门用于文本操作的函数 fputc, fgetc, fputs, fgets, fscanf, fprintf 读写。
- 3、 如果打开方式与用于读写的函数的方式不匹配，结果是难以预料的，甚至出现结果莫名其妙无法解释。

# 第11次作业(第12章)

p.343 1, 2, 6