第 X 章 C 语音中文件

X.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。每个文件都有一个名字，称为文件名。一般来说，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。本节介绍有关文件的几个基本概念。

1. 文本文件与二进制文件

在 C 语言中，根据文件中数据的存储形式，文件一般分为文本文件和二进制文件两种。

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。例如，一个 short 整数 23145，它由 5 个数字字符组成，在文本文件中为了存储该整数就需要 5 个字节，如图 X.1(a)所示。用一般的文本编辑器能编辑、人能读懂的文件是文本文件，比如 short 整数 23145 存放在一个文件中，用文本编辑器打开，你会看到 23145 这个数字串。这种文件是由 ASCII 字节流组成的，又称为流式文件。

二进制文件中的数据与该数据在计算机内的二进制形式是一致的，其中一个字节并不代表一个字符。例如，同样的一个 short 整数 23145，化成二进制数为 0101101001101001，因此，它在二进制文件中存一个 short 数只需要占 2 个字节就够了，如图 X.1(b)所示。

2	3	1	4	5	
0011 0010	0011 0011	0011 0001	0011 0100	0011 0101	占 5 个字节

(a) 按文本形式存放

0101 1010	0110 1001	占 2 个字节

(b) 按二进制形式存放

图 X.1　整数 2314 在文件中的两种存储形式

例如，将 int a=12345,b=567890; float f=97.6875f; double f2=97.6875; 用文本方式写到文本文件 data1.txt 中，用二进制方式写到二进制文件 data2.txt 中，则用文本编辑器打开 data1.txt (文件长度 34 字节)，结果如图 X.2 所示。可以看到字符型的 ASCII 串：12345 567890 97.687500 97.687500。

而用文本编辑器打开二进制文件 data2.txt (文件长度 20 字节)，结果如图 X.3 所示。可以说基本上是无法直接读懂的。真正的意思是：

12345_{10} = 303916 = 00 00 30 39_{16} 对应　39 30 00 00　　(4 字节颠倒存放)

567890_{10} = $8AA52_{16}$ = 00 08 AA 52_{16}对应　52 AA 08 00　(4 字节颠倒存放)

97.6875_{10}=$(0.1000011011)_2*2^7$　的 float= 42 C3 60 00_{16}对应　00 60 C3 42 (4 字节颠倒存放)

97.6875_{10}的 double = 40 58 6C 00 00 00 00 00 对应 00 00 00 00 00 6C 58 40　(8 字节颠倒存放)

同样是 97.6875，其 float 和 double 的二进制表示结果是完全不同的。有关 float, double 数的二进制存放格式，请参考 2.1.2 小节。

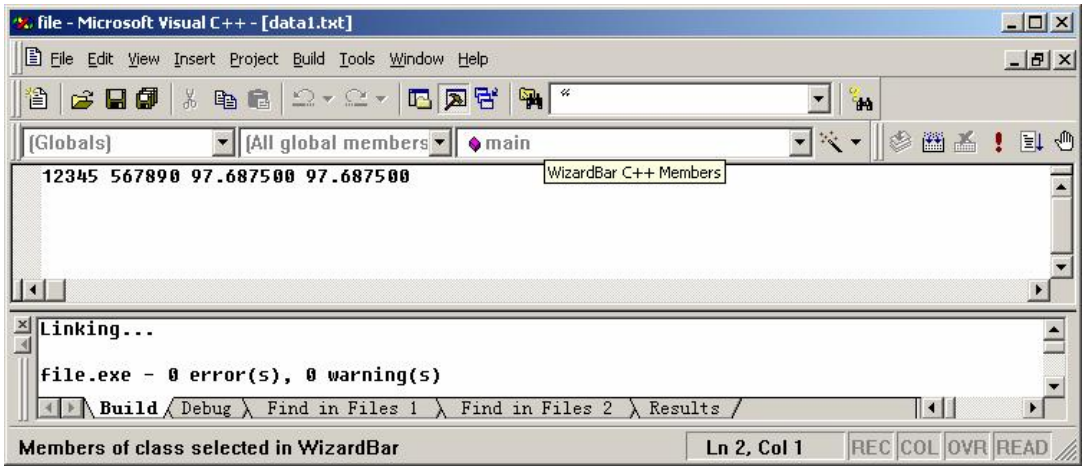

图 X.2　文本文件中的整数和实数

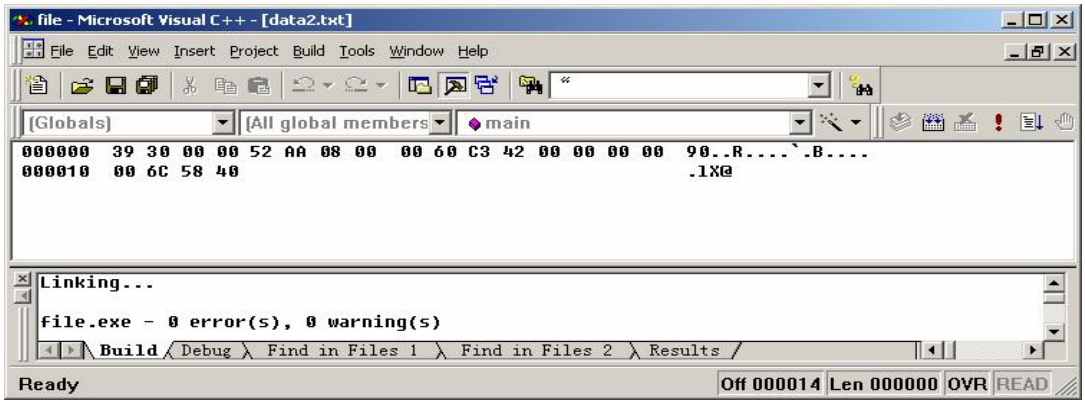

图 X.3　二进制文件中的整数和实数

2. 缓冲文件系统

对文件的处理一般有两种方式，分别称为缓冲文件系统与非缓冲文件系统。

所谓缓冲文件系统是指系统自动地为正在被使用的文件在内存中开辟一个缓冲区。当需要向外存储器中的文件输出数据时，必须先将数据送到为该文件开辟的缓冲区中，当缓冲区满以后才一起送到外存储器中。当需要从外存储器中的文件读入数据进行处理时，也首先一次从外存储器将一批数据读入缓冲区（将缓冲区充满），然后再从缓冲区中将数据逐个读出进行处理。由此可以看出，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缓冲区进行的，对文件中数据的处理也是在该缓冲区中进行的。缓冲文件系统又称为高级文件系统。

设置文件缓冲区的好处是：减少程序直接进行 I/O 操作的次数，将每次读写一个数便进行一次 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 I/O 操作是机械操作，不可能每秒钟操作成千上万次，而 CPU 的执行速度通常都在亿次以上，因而过多的 I/O 操作会影响整个程序的执行效率。

设置文件缓冲区的缺点是：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缓冲区，还没有真正写到磁盘介质上，如果此时程序非正常终止，会出现缓冲区中的数据丢失，没有真正写到文件中去。

所谓非缓冲文件系统，是指系统不自动为文件开辟缓冲区，而是由用户程序自己为文件设定缓冲区。非缓冲文件系统又称为低级文件系统。这种情况，是程序的每次 I/O 读写，都直接访问磁盘介质。实例就是每次计算机开机时，加载操作系统的过程，就是非缓冲文件操作。

在 C 语言中，对文件的操作都是通过库函数来实现的，本章主要介绍缓冲文件系统中的操作。

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文件。

在缓冲文件系统中，每个被使用的文件都要在内存中开辟一个缓冲区，在这个缓冲区中，用于存放文件的有关信息，一般包括文件的名字、文件的状态以及文件所在的位置等信息。在 C 语言中，缓冲区中的这些信息作为一个整体来组织，即每个文件对应一个结构体类型的数据，其中的成员用于存放文件的有关信息。系统将该结构体类型定义为 FILE（注意：是英文大写字母，结构体类型 FILE 在 stdio.h 中定义，因此要进行文件操作必须要 include 系统头文件 stdio.h），简称文件类型。利用结构体 FILE 类型可以定义文件类型的变量，用于存放缓冲区中的文件信息；也可以定义文件类型的指针，用于指向存放文件信息的缓冲区。

定义文件类型指针的一般形式为

```
FILE  *指针变量名;
```

其中指针变量名用于指向一个已经打开的文件，实际上是指向文件缓冲区的首地址。例如，

```
FILE  *fp;
```

定义了一个结构体 FILE 类型的指针变量 fp，用它可以指向某一个被打开文件的结构体数据。

一般来说，对文件操作有以下三个方面：

(1)打开文件

在计算机内存中开辟一个缓冲区，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存(磁盘、U 盘等)中，并释放缓冲区。

X.2　文件的基本操作

X.2.1　文件的打开与关闭

1. 文件的打开

在 C 语言中，打开一个文件的一般形式如下：

```
FILE  *fp(或其它指针变量名);
   …
fp=fopen("文件名", "文件使用方式");
```

由此可以看出，为了打开一个文件，首先要为该文件定义一个文件指针，然后用 C 语言提供的 fopen 函数打开文件。fopen 函数有两个参数："文件名"，"文件使用方式"，它们均是字符串。其中"文件名"指出要打开的文件的名字；"文件使用方式"指出以何种方式打开文件。

在正常情况下，fopen 函数的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。

在 C 语言中，"文件使用方式"可以是以下几种对文件的访问形式：

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 NULL。
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。
a	追加写	向文件尾增加数据。若指定的文件不存在，则创建一个新文件。
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 NULL。
w+	读写	为读写打开一个新文件。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。
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。

如果在后面附加"b"，如："rb"，"r+b"，"wb"，"w+b"，"ab"，"a+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。文本文件可以在后面附加"t"，如："rt"，"r+t"，"wt"，"w+t"，"at"，"a+t"，不过可以省略不写。

例如，程序段

```
FILE  *fp;
fp=fopen("ABC.TXT", "r+");
```

与

```
FILE  *fp;
char  *fname="ABC.TXT";
fp=fopen(fname, "r+");
```

是等价的，它们都是以读写方式打开文本文件 ABC.TXT，若文件 ABC.TXT 不存在，则返回空指针 NULL。

最后需要指出的是，在打开一个文件时，有时会出错。例如，在用"r"或"r+"方式打开一个文件时，要求被打开的文件必须存在，如果不存在，则 fopen()函数会返回一个空指针值 NULL，从而完不成“打开”的任务。又如，在打开文件时，如果磁盘出故障，或者磁盘已满无法建立新文件，此时也完不成“打开”任务，此时 fopen()函数也返回一个空指针值 NULL。在这种情况下，后面的程序也就无法对文件进行处理。

由于上述原因，在一般的 C 程序中，常采用以下方式来打开文件：

```
#include  <stdio.h>
#include  <stdlib.h>
FILE  *fp;
    …
if ((fp=fopen("文件名", "文件使用方式"))==NULL)
{    printf("cannot open this file!\n");
     exit(0);    /* 终止调用过程 */
}
```

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

cannot open this file!

并退出当前的调用过程。为了使用 exit( )函数，必须要 include 系统头文件 stdlib.h 。

2. 文件的关闭

对文件操作完成后，必须要关闭文件。

在 C 语言中，关闭文件的一般形式如下：

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存文件中，然后释放该缓冲区。

当文件被关闭后，如果再想对该文件进行操作，则必须再打开它。

虽然 C 语言允许打开多个文件，但由于资源的限制能同时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。因此，建议当一个文件使用完后应立即关闭它。

X.2.2 文件的读写

对文件进行读操作，是指从指定的文件向程序输入数据。

对文件进行写操作，是指将程序中处理好的数据写到指定的文件中。

在 C 语言中，为了实现对文件的读写，提供了字符读写函数、数据块读写函数与格式读写函数。

1. 字符读写函数

字符读写函数主要适用于文本文件的读写。

(1) 读字符函数 fgetc()

读字符函数的一般形式为

fgetc(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。若读到文件尾或读入不成功，返回值为 EOF。例如，

```
char  c;
   …
c=fgetc(fp);     /* 假设该文件已打开，下同 */
```

从 fp 指向的文件中读取一个字符赋给字符型变量 c。

例 X.1 从文本文件 a.txt 中顺序读入字符并在屏幕上显示输出。C 程序如下：

```
#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(c!=EOF)       /* EOF 为文件的结束标志 */
  {    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 */
}
```

程序中的 EOF 是文本文件的结束标志。

(2) 写字符函数 fputc()

写字符函数的一般形式为

fputc(c, fp)

其中 fp 为文件类型的指针，指向已打开的文件；c 可以是字符型变量，也可以是字符型常量与字符型表达式。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 X.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 程序如下：

```
#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 */
}
```

C 语言除了提供读写单个字符的函数外，还提供了读写字符串的函数。

(3) 读字符串函数 fgets()

读字符串函数的一般形式为

fgets(str，n，fp);

其中 fp 为文件类型的指针，指向已打开的文件；str 是一个字符内存区首地址指针；n 是一个整型变量，也可以是整型常量或整型表达式。该函数的功能是，从指定的文件读入 n-1 个字符存放到由 str 指向的内存空间中，读入结束后，将自动在最后加一个字符串结束符'\0'。如果读入不成功，函数的返回值为 NULL。例如，

```
char  s[20];
```

…

fgets(s, 20, fp);　　/* 假设文件 fp 已打开 */

从 fp 指向的文件中读取 19 个字符后再加一个字符串结束符'\0'，存放到由 s 指向的存储空间中。

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。

(4) 写字符串函数 fputs()

写字符串函数的一般形式为

fputs(str, fp);

其中 fp 为文件类型的指针，指向已打开的文件；str 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。该函数的功能是，将指定的字符串写到文件 fp 中。若写成功，则返回函数值 0，否则返回非 0 函数值。例如，语句

fputs("How do you do!", fp); /* 假设文件 fp 已打开 */

的功能是将字符串"How do you do!"写到由 fp 指向的文件中。

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，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，必要时可以人为地加入如"\n"这样的格式控制字符串。

2. 数据块读写函数

数据块读写函数主要适用于二进制文件的读写。

在具体介绍数据块读写函数之前，先介绍一个判文件结束函数。

(1) 判文件结束函数 feof()

在前面例 X.1 的程序中读取文件字符时，曾经用文件结束标志 EOF 来判断是否遇到文件结束符。但在 C 语言中，只有文本文件才是以 EOF 作为文件结束标志的，而二进制文件不是以 EOF 作为文件结束标志的。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。

判文件结束函数的一般形式为

feof(fp)

其中 fp 指向已打开的文件。该函数的功能是在读 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 1；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。例如，例 X.1 中的程序也可以改为

```
#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 */
}
```

程序中的 while 循环的条件 !feof(fin) 是用来判断是否到达文件末尾。

由此可以看出，对于文本文件来说，既可以用文件结束标志 EOF 来判断文件是否结束，也可以用函数 feof()来判断文件是否结束；但对于二进制文件来说，只能用函数 feof()来判断文件是否结束。

使用函数 feof()进行文件操作，经常会导致在初学者看来莫名其妙的问题。例如，有程序：

```
#include   <stdio.h>
#include   <stdlib.h>
main( )
{  FILE  *fin, *fout;
   char  a[80];
   if ((fin=fopen("a.txt","r"))==NULL)
   {  printf("cannot  open  this  file !\n");
      exit(0);
   }
   if ((fout=fopen("b.txt","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*/
}
```

此程序的本意是完全复制 a.txt 文件的内容到 b.txt。结果如何呢？若文件 a.txt 的内容是：

```
abcdefg
12345
998e929292
ddlldldldldldlldldldldl
```

那么所生成的文件 b.txtt 的内容会是什么呢？打开 b.txt 文件，其内容是：

```
abcdefg
12345
998e929292
ddlldldldldldlldldldldl
ddlldldldldldlldldldldl
```

看起来最后一行重复多写了一次。原因是：feof(fin)只有当读不成功才置为 1， 此时 while(!feof(fin))才终止，而 fgets 最后一次读不成功，没有读入任何信息，字符数组 a 仍是上次的内容，因此紧跟 fgets 的 fputs 把上次的字符串又写了一次。因此 feof 函数的作用老是慢一拍。其实无论哪个读写函数，读写后都会有返回值指明读写是否成功。比如，fgets 如果读入不成功，将返回 NULL。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。因此为了避免出现最后一行重复写的问题，上面程序可以修改为：

```
#include   <stdio.h>
#include   <stdlib.h>
main( )
{  FILE  *fin, *fout;
   char  a[80];
   if ((fin=fopen("a.txt","r"))==NULL)
   {  printf("cannot open this file !\n");
      exit(0);
   }
   if ((fout=fopen("b.txt","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)  /* 从文件读取一行字符 */
          break;   /* 只要读不成功, 立刻终止 */
       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()

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，则返回读入的项数，否则返回小于等于的数。其形式为

```
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。

count　　要读入的数据项个数。

fp　　文件类型指针，指向已打开的文件。

例 X.3　下列 C 程序段的功能是,从二进制文件 b.dat 中读入 4 个整数存放到整型数组 x 中。

```
#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);
  fclose(fp);
  printf("%d %d %d %d\n",x[0], x[1], x[2], x[3]);
}
```

(3) 数据块写函数 fwrite()

数据块写函数的功能是，将一组数据以二进制格式写到指定的文件中。若成功写出，则返回写出的项数，否则返回小于等于的数。其形式为

fwrite(buffer, size, count, fp);

其中：buffer　输出数据的首地址。

size　每个数据项的字节数。

count　要写出的数据个数。

fp　文件类型指针，指向已打开的文件。

例 X.4　下列 C 程序段的功能是，将一维数组中的元素存放到二进制文件 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);
  fclose(fp);
}
```

3. 格式读写函数

格式读写函数主要适用于文本文件的读写。

(1) fscanf()函数

该函数的功能是，从指定的文件中格式化读数据。若成功读入，则返回读入的项数，否则返回小于等于的数。其一般形式为

fscanf(文件指针, 格式控制, 地址表);

这个函数与格式输入函数 scanf()很相似，它们的区别就在于，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。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);

表示从键盘输入数据。

例 X.5　下列 C 程序段的功能是从文本文件 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);
}
```

(2) fprintf()函数

该函数的功能是，格式化写数据到指定的文件中。若成功写出，则返回写出的项数，否则返回小于等于的数。其形式为

fprintf(文件指针, 格式控制, 输出表)

这个函数与格式输出函数 printf()很相似，它们的区别就在于，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。

例 X.6　下列 C 程序段的功能是，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;
    double x;
    a=10; 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);
}
```

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。

有关文件读写的函数还有许多，请读者参看附录 2。

X.2.3　文件的定位

为了正确地对文件进行读写操作，在一个文件被打开后，系统就为该文件设置一个读写指针，用于指示当前读写的位置。当进行一次读写操作后，文件的读写指针也就自动地发生改变，通常指向相应所读写数据在文件中所在位置的后面。并且，C 语言也提供了改变文件读写指针的函数，称为文件定位函数。

文件定位函数主要有以下几个。

1. rewind()函数

该函数的功能是，将文件的读写指针移动到文件的开头。其形式为

```
rewind(fp)
```

其中 fp 是已经打开的文件指针。

2. fseek()函数

该函数的功能是，将文件的读写指针移动到指定的位置。其形式为

```
fseek(文件指针, 偏移量, 起始位置)
```

其中各参数的意义如下：

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	表示文件首
SEEK_CUR 或 1	表示当前读写的位置
SEEK_END 或 2	表示文件尾

偏移量是指以“起始位置”为基点，读写指针向文件尾方向移动的字节数。这个参数的类型要求为长整型。

例如，

```
fseek(fp, 100L, SEEK_SET);
```

表示以文件首为起点，将文件读写指针往文件尾方向移动 100 个字节。又如，

```
fseek(fp, 50L, SEEK_CUR);
```

表示将文件读写指针从当前读写位置开始往文件尾方向移动 50 个字节。又如，

fseek(fp, -10L, SEEK_END);

表示以文件尾为基点，将文件读写指针往文件首方向（向前）移动 10 个字节。当偏移量为正数时，表示文件读写指针往文件尾方向（向前）移动，偏移量为负数时，表示文件读写指针往文件首方向（向后）移动。

上述三个语句也可以写成如下：

```
fseek(fp, 100L, 0);
fseek(fp, 50L, 1);
fseek(fp, -10L, 2);
```

下面通过一个例子说明 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)
  {  printf("cannot  open  this  file !\n");
     exit(0);
   }
  fwrite(x, sizeof(double), 5, fp);  /* 将 x 中 5 个 double 数写到文件中 */
                                     /* 此时文件指针指向最后一个数后的位置 */
  fseek(fp, 0L, SEEK_SET);  /*此时文件指针移动到文件开始处，为读做好准备*/
  fread(&y[0], sizeof(double), 2, fp); /* 读入 2 个 double 数到 y[0], y[1]中 */
                                /* 此时文件指针指向文件中第 3 个 double 数的开始处 */
  fseek(fp, -4L*sizeof(double), SEEK_END);
                                /*从文件尾向前移动 4 个 double 数的位置*/
                               /* 此时文件指针指向文件中第 2 个 double 数的开始处 */
  fread(&y[2], sizeof(double), 2, fp); /* 读入 2 个 double 数到 y[2], y[3]中 */
                              /* 此时文件指针指向文件中第 4 个 double 数的开始处 */
  fseek(fp, - (long)sizeof(double), SEEK_CUR);
                               /*从文件的当前位置向前移动 1 个 double 数的位置,*/
                              /* 此时文件指针指向文件中第 3 个 double 数的开始处 */
  fread(&y[4], sizeof(double), 2, fp); /* 读入 2 个 double 数到 y[4], y[5]中 */
  fclose(fp);
 for (k=0; k<6; k++) printf("%4.1f ", y[k]);
}
```

程序的运行结果是

```
1.1  2.3  2.3  4.5  4.5  -3.6
```

关于 fseek 的几点注意事项：

（1）如果成功定位，fseek 返回值为 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）你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(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) 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。

（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）。

3. ftell( )函数

该函数的功能是，返回文件的当前读写位置(出错返回-1L)。其形式为

```
long ftell(FILE *fp);
```

有程序：

```
# 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!"，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。

X.2.4 文件缓冲区的清除

函数 fflush 的功能是，清空文件的输入输出缓冲区流，即使输出到文件中（出错返回-1L）。

其形式为 int fflush(FILE *fp);

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);
    fgets(c,20, fp);
    c[20] = '\0';
    puts(c);
    fclose(fp);
}
```

程序的运行结果是

```
23456789012345678901
```

如果此时打开文件 r.txt，看到的内容将是：

```
12 23456789012345678901
```

按照程序的操作，应该是向文件 r.txt 输出了 2 次字符串"1234567890"，r.txt 中有 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',fputs(" ", fp);写出一个空格字符，空格字符应该覆盖了第 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 的内容也不是我们期待的：

```
12 45678901234567890
```

出现这个问题的原因是，向文件中写数据 fputs(" ", fp);，并不是立刻写到磁盘上的文件中，而是在文件输出缓冲区中，这时候立刻用 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。 程序改为：

```
#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 );  /* 关键是需要这个函数清空缓冲区 */
    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，结果也是完全正确的。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了。

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。

X.3 程序举例

例 X.7 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);
}
```

在上述程序中，函数“fgetc(fp)” 的功能主要是读入文件指针所指当前位置的字符值，让文件的读写指针往后移动一个字符，fgetc 的返回值是，如遇文件结束，则返回 EOF，否则返回读入的字符值。

例 X.8　下列 C 程序的功能是，用“追加”的形式打开文本文件 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);
}
```

例 X.9　下列 C 程序的功能是，将程序中的 10 个学生的信息以二进制方式写到文件 student.dat 中。

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

```
struct student
{ int num;
  char name[8];
  char sex;
  int age;
  double score;
};
main()
{ struct student stu[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, "Gou", 'M', 19, 90.2},
                          {106, "Lin", 'M', 18, 91.5},
                          {107, "Ma", 'F', 17, 98.7},
                          {108, "Zhen", 'M', 21, 90.1},
                          {109, "Xu", 'M', 19, 89.8},
                          {110, "Mao", 'F', 18, 94.9}};
  FILE *fp;
  if ((fp=fopen("student.dat", "wb"))==NULL)
  {   printf("cannot open student.dat!\n");
      exit(0);
  }
  fwrite(stu, sizeof(struct student), 10, fp);
            /* 将 10 个人的信息一次性写到文件中 */
  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*10=250 字节。原因是还是自动对齐。对于结构体

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

因为其中单个最长的成员 double 是 8 字节，因此自动以 8 字节对齐。int num 占 1 个 8 字节（其中后面空闲 4 个字节），char name[8];占 1 个 8 字节，char sex; 和 int age;占 1 个 8 字节（char sex;占 8 字节的第 1 个字节，空闲 3 个字节，int age;占 8 字节从第 5 个字节开始的 4 个字节），double score; 占 1 个 8 字节，因此每个结构体占 32 个字节。

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 sex;占一个 4 字节（其中后面空闲 3 个字节），每个结构体占 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 sex;占一个 2 字节（其中后面空闲 1 个字节），每个结构体占 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，每个结构体占 25 个字节。

例 X.10　下列 C 程序的功能是，将二进制文件 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 sex;
  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);
    /* 将 10 个人的信息一次性从文件中读入到内存结构体数组中 */
  fclose(fp);
  printf("No.      Name      Sex      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].sex, stu[i].age, stu[i].score);
}
```

如果使用例 X.9 中创建的文件，则上述程序的运行结果为

```
No.      Name      Sex      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	Gou	M	19	90.20
106	Lin	M	18	91.50
107	Ma	F	17	98.70
108	Zhen	M	21	90.10
109	Xu	M	19	89.80
110	Mao	F	18	94.90

需要特别说明的是，若例 X.9 中创建文件的程序中使用了#pragma pack 方式，本例中必须使用同样的#pragma pack 方式，才能保证正确读入数据。

练习 12

1. 编写一个 C 程序，首先从键盘输入 20 个双精度实数，并写入文本文件 fd.dat 中。然后将写入文件 fdata.dat 中的 10 个双精度实数显示输出。

2. 编写一个 C 程序，从键盘输入一个字符串（输入的字符串以“#”作为结束），将其中的小写字母全部转换成大写字母，并写入到文件 upper.txt 中。然后再从该文件中的内容读出并显示输出。

3. 编写一个 C 程序，主函数从命令行得到一个文件名，然后调用函数 fgets()从文件中读入一字符串存放到字符数组 str 中（字符个数最多为 80 个）。在主函数中输出字符串与该字符串的长度。fgets 函数的格式为：char *fgets(char *string, int n, FILE *stream);

4. 编写一个 C 程序，将源文件拷贝到目的文件中。两个文件均为文本文件，文件名均由命令行给出，并且源文件名在前，目的文件名在后。

5. 设二进制文件 student.dat 中存放着学生信息，这些信息由以下结构体描述：

```
struct student
{ long int num;
  char name[10];
  int age;
  char sex;
  char sreciality[20];
  char addr[40];
};
```

请编写一个 C 程序，显示输出学号在 970101～970135 之间的学生学号 num、姓名 name、年龄 age 与性别 sex。

6. 设有一学生情况登记表如表 X.1 所示。

表 X.1　学生情况登记表

学号(num)	姓名(name)	性别(sex)	年龄(age)	成绩(grade)
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	Gao	M	19	90.2
106	Lin	M	18	91.5
107	Ma	F	17	98.7
108	Zhen	M	21	90.1
109	Xu	M	19	89.5
110	Mao	F	18	94.5

编写一个 C 程序，依次实现以下操作：

(1) 定义一个结构体类型

```
struct  student
{ char  num[7];
  char  name[8];
  char  sex[3];
  char  age[5];
  char  grade[9];
};
```

(2) 为表 X.1 定义一个结构体类型(struct student)数组，并进行初始化。

(3) 打开一个可读写的新文件 stu.dat。

(4) 用函数 fwrite()将结构体数组内容写入文件 stu.dat 中。

(5) 关闭文件 stu.dat。

(6) 打开可读写文件 stu.dat。

(7) 从文件 stu.dat 中读出各学生情况并输出。输出格式如表 X.1 所示，但不要表格框线。

(8) 关闭文件 stu.dat。

7. 将表 X.1 的内容按结构体类型写入文本文件 st.dat 中；然后对该文件按成绩从低到高进行冒泡排序，并输出排序结果；最后，在排序后的文件中用对分查找法查找并输出成绩在 95.0 到 100 分之间的学生情况。

具体要求：

(1) 在定义的结构体类型中，各成员均为字符数组。即结构体类型的定义如下：

```
struct  student
{ char  num[8];
  char  name[8];
  char  sex[5];
  char  age[5];
  char  grade[10];
};
```

(2) 编写一个对文本文件（其中每一个记录的结构如(1)中定义）按成绩(grade)进行冒泡排序的函数 mudisk(fp，n)。其中：

fp 为文件类型指针，指向待排序的文件；

n 为长整型变量，存放待排序文件中的记录个数（即学生的个数)。

(3) 编写一个对按成绩(grade)有序的文件（其中每一个记录的结构如(1)中定义）进行对分查找的函数 nibsrch(fp，n，a，b，m)。其中：

fp 为文件类型指针，指向给定的有序文件；

n 为长整型变量，存放按成绩有序文件中的记录个数（即学生的个数)；

a 与 b 均为字符串指针，分别指向成绩（grade 作为字符串）值的下限与上限（即查找成绩在 a 到 b 之间的学生)；

m 为整型变量指针，该指针指向的变量返回成绩在 a 到 b 内的第一个记录号（即数组元素下标)。

(4) 在主函数外定义结构体类型，且主函数放在所有函数的前面。

(5) 在主函数中依次完成以下操作：

①为表 X.1 定义一个结构体类型（struct student）数组，并进行初始化。

②打开可读写的新文本文件 st.dat。

③使用函数 fwrite()将结构体数组内容写入文件 st.dat 中。

④关闭文件 st.dat。

⑤打开可读写文件 st.dat。

⑥调用(2)中的函数 mudisk(fp，n)，对文件 st.dat 按成绩（grade）从低到高进行冒泡排序。

⑦调用(3)中的函数 nibsrch(fp，n，a，b，m)，查找成绩在 95.0 到 100 分之间的学生情况。

⑧根据查找的返回结果，使用函数 fread()，从文件中读出成绩在 95.0 到 100 分之间的学生情况并输出。输出格式如表 X.1 所示，但不要表格框线。

⑨关闭文件 st.dat。