第23讲：文件操作

目录

正文开始

1. 为什么使用文件?

如果没有文件，我们写的程序的数据是存储在电脑的内存中，如果程序退出，内存回收，数据就丢失了，等再次运行程序，是看不到上次程序的数据的，如果要将数据进行**持久化的保存**，我们可以使用**文件**。

2. 什么是文件?

磁盘（硬盘）上的文件是文件。

但是在程序设计中，我们一般谈的文件有两种：程序文件、数据文件（从文件功能的角度来分类的）。

2.1 程序文件

程序文件包括源程序文件（后缀为.c）,目标文件（windows环境后缀为.obj）,可执行程序（windows环境后缀为.exe）。

2.2 数据文件

文件的内容不一定是程序，而是程序运行时读写的数据，比如程序运行需要从中读取数据的文件，或者输出内容的文件。

本章讨论的是数据文件。

在以前各章所处理数据的输入输出都是以终端为对象的，即从终端的键盘输入数据，运行结果显示到显示器上。

其实有时候我们会把信息输出到磁盘上，当需要的时候再从磁盘上把数据读取到内存中使用，这里处理的就是磁盘上文件。

2.3 文件名

一个文件要有一个唯一的文件标识，以便用户识别和引用。

文件名包含3部分：文件路径+文件名主干+文件后缀

例如：`c:\code\test.txt`

为了方便起见，文件标识常被称为**文件名**。

3. 二进制文件和文本文件

根据数据的组织形式，数据文件被分为**文本文件**和**二进制文件**。

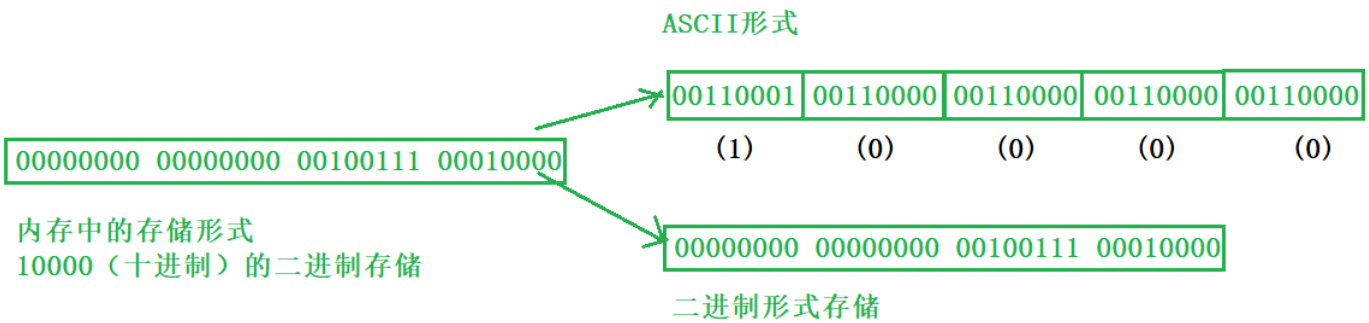
数据在内存中以二进制的形式存储，如果不加转换的输出到外存的文件中，就是**二进制文件**。

如果要求在外存上以ASCII码的形式存储，则需要在存储前转换。以ASCII字符的形式存储的文件就是**文本文件**。

一个数据在文件中是怎么存储的呢?

字符一律以ASCII形式存储，数值型数据既可以用ASCII形式存储，也可以使用二进制形式存储。

如有整数10000，如果以ASCII码的形式输出到磁盘，则磁盘中占用5个字节（每个字符一个字节），而二进制形式输出，则在磁盘上只占4个字节。



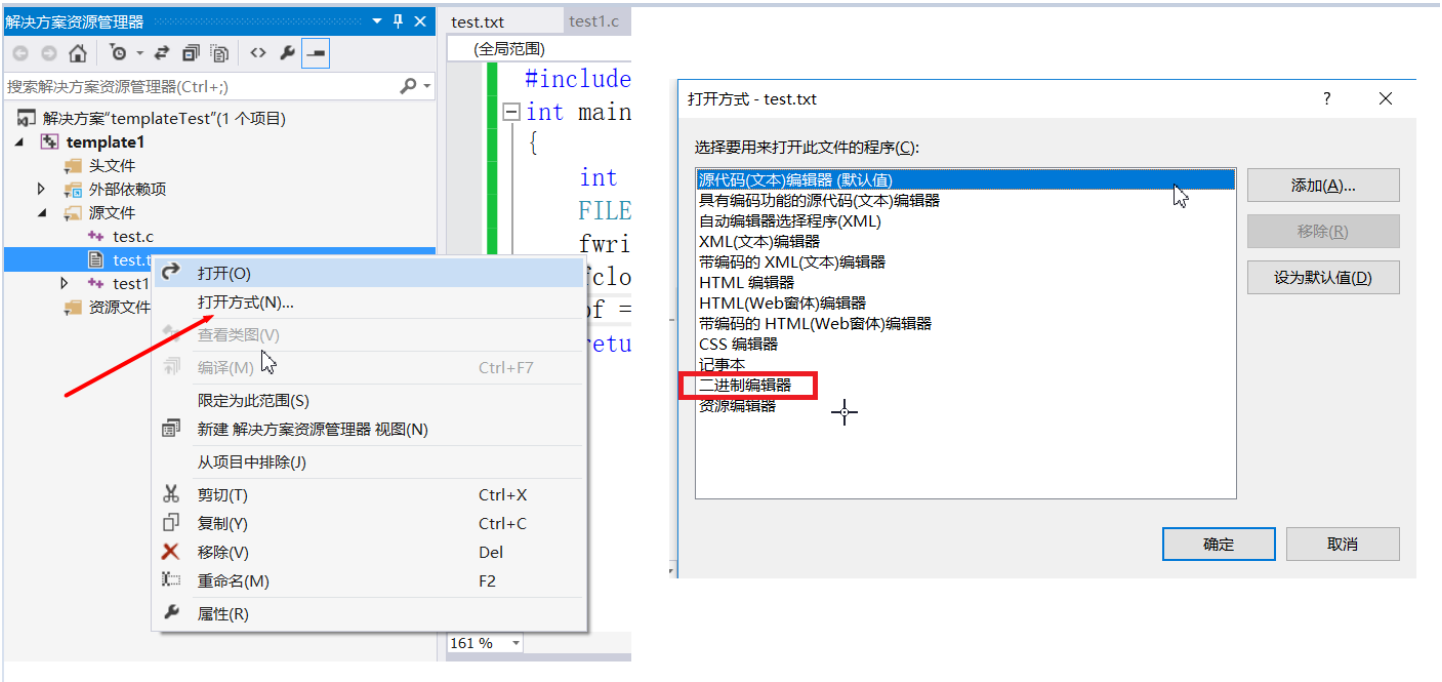
演示代码：

代码块

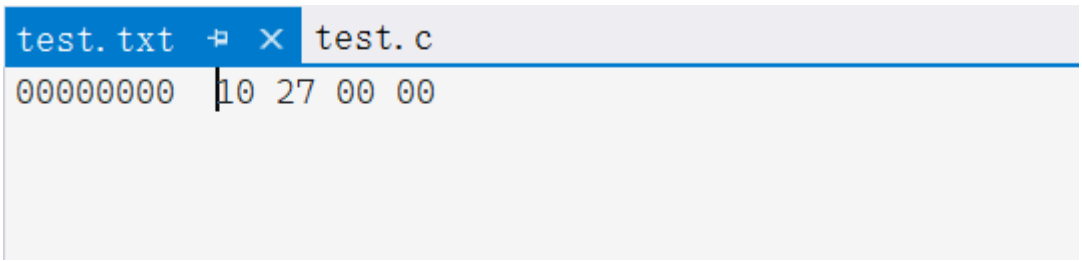
```
1   #include <stdio.h>
```

```c
int main()
{
    int a = 10000;
    FILE* pf = fopen("test.txt", "wb");
    fwrite(&a, 4, 1, pf);//二进制的形式写到文件中
    fclose(pf);
    pf = NULL;
    return 0;
}
```

在VS上打开二进制文件：

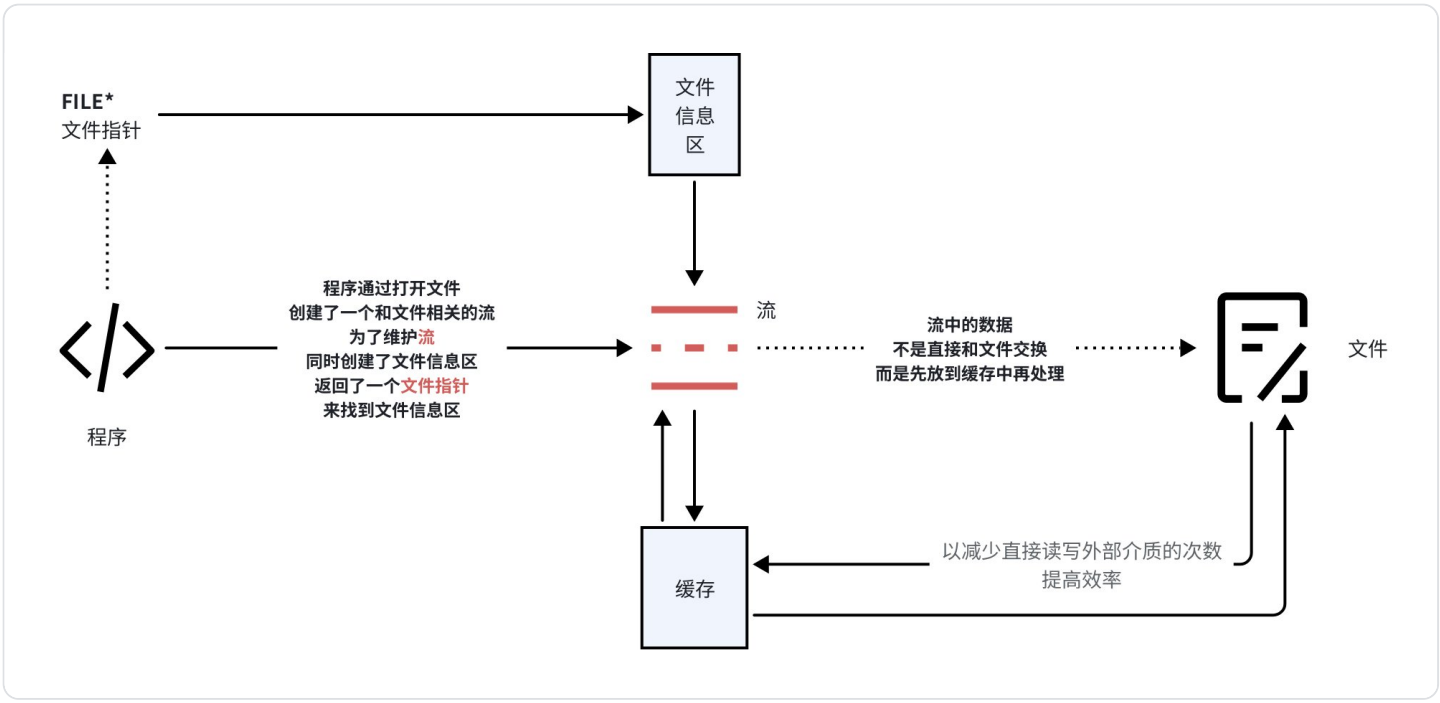


VS上打开二进制文件的方法



10000在二进制文件中

4. 文件的打开和关闭



4.1 流和标准流

4.1.1 流

我们程序的数据需要输出到各种外部设备，也需要从外部设备获取数据，不同的外部设备的输入输出操作各不相同，为了方便程序员对各种设备进行方便的操作，我们抽象出了流的概念，我们可以把流想象成流淌着字符的河。

C程序针对文件、画面、键盘等的数据输入输出操作都是通过流操作的。

一般情况下，我们要想向流里写数据，或者从流中读取数据，都是要打开流，然后操作。

4.1.2 标准流

那为什么我们从键盘输入数据，向屏幕上输出数据，并没有打开流呢?

那是因为C语言程序在启动的时候，默认打开了3个流：

- **stdin** - 标准输入流，在大多数的环境中从键盘输入，scanf函数就是从标准输入流中读取数据。
- **stdout** - 标准输出流，大多数的环境中输出至显示器界面，printf函数就是将信息输出到标准输出流中。
- **stderr** - 标准错误流，大多数环境中输出到显示器界面。

这是默认打开了这三个流，我们使用scanf、printf等函数就可以直接进行输入输出操作的。

`stdin`、`stdout`、`stderr` 三个流的类型是：`FILE *`，通常称为**文件指针**。

C语言中，就是通过`FILE*`的文件指针来维护流的各种操作的。

4.2 文件指针

缓冲文件系统中，关键的概念是“**文件类型指针**”，简称“**文件指针**”。

每个被使用的文件都在内存中开辟了一个相应的**文件信息区**，用来存放文件的相关信息（如文件的名字，文件状态及文件当前的位置等）。这些信息是保存在一个结构体变量中的。该结构体类型是由系统声明的，取名 **FILE**.

例如，VS2013 编译环境提供的 `stdio.h` 头文件中有以下的文件类型申明：

代码块

```
struct _iobuf
{
    char *_ptr;
    int   _cnt;
    char *_base;
    int   _flag;
    int   _file;
    int   _charbuf;
    int   _bufsiz;
    char *_tmpfname;
};

typedef struct _iobuf FILE;
```

不同的C编译器的FILE类型包含的内容不完全相同，但是大同小异。

每当打开一个文件的时候，系统会根据文件的情况自动创建一个FILE结构的变量，并填充其中的信息，使用者不必关心细节。

一般都是通过一个FILE的指针来维护这个FILE结构的变量，这样使用起来更加方便。

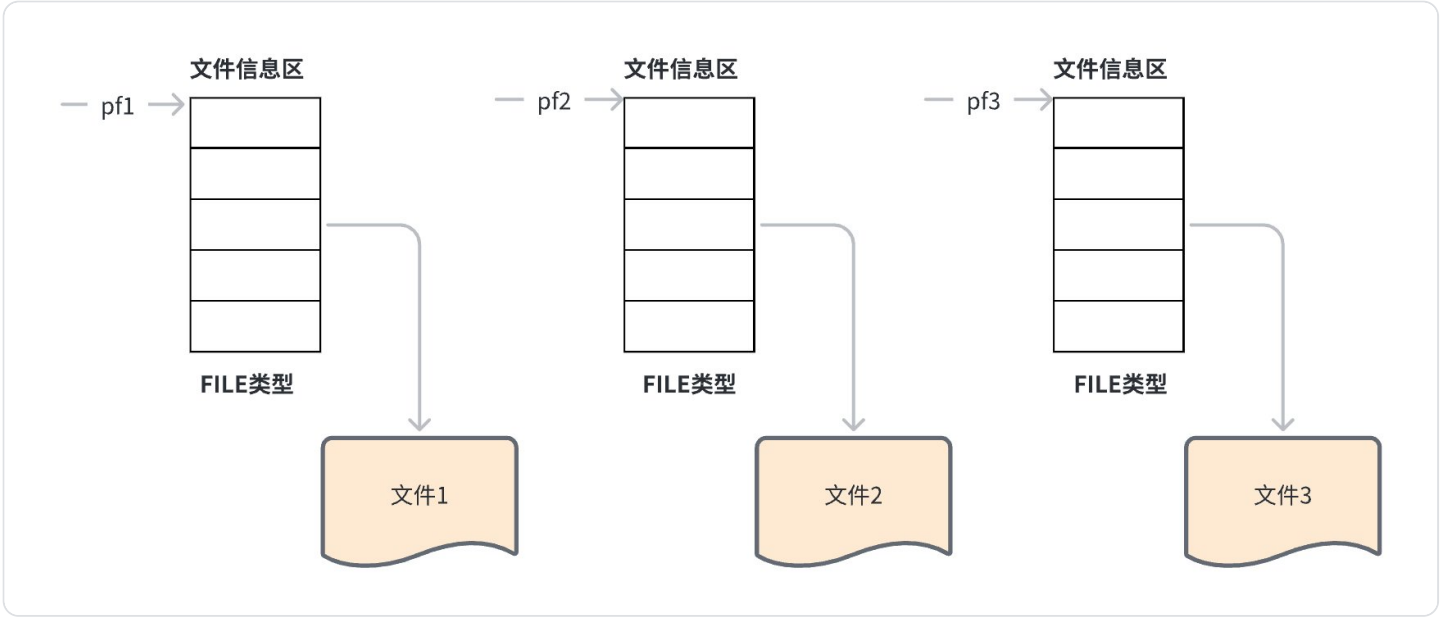
下面我们可以创建一个FILE*的指针变量:

代码块

```
FILE* pf;//文件指针变量
```

定义pf是一个指向FILE类型数据的指针变量。可以使pf指向某个文件的文件信息区（是一个结构体变量）。通过该文件信息区中的信息就能够访问该文件。也就是说，**通过文件指针变量能够间接找到与它关联的文件**。

比如：



4.3 文件的打开和关闭

文件在读写之前应该先**打开文件**，在使用结束之后应该**关闭文件**。

在编写程序的时候，在打开文件的同时，都会返回一个FILE*的指针变量指向该文件，也相当于建立了指针和文件的关系。

ANSI C 规定使用 `fopen` 函数来打开文件， `fclose` 来关闭文件。

4.3.1 fopen

代码块

```
1   //打开文件
2   FILE * fopen ( const char * filename, const char * mode );
```

功能： `fopen` 函数是用来打开参数 `filename` 指定的文件，同时将打开的文件和一个流进行关联，后续对流的操作是通过 `fopen` 函数返回的指针来维护。具体对流（关联的文件）的操作是通过参数 `mode` 来指定的。

参数：

`filename` ：表示被打开的文件的名字，这个名字可以绝对路径，也可以是相对路径。

`mode` ：表示对打开的文件的操作方式，具体见下面的表格。

返回值：

- 若文件成功打开，该函数将返回一个指向 `FILE` 对象的指针，该指针可用于后续操作中标识对应的流。
- 若打开失败，则返回 `NULL` 指针，所以一定要对 `fopen` 的返回值做判断，来验证文件是否打开成功。

代码演示：

fopen函数的演示

```c
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "r"); //以“r”的形式打开文件，如果文件不存在，则打开失败
    if(pf == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    printf("打开文件成功，可以对文件进行操作\n");
    //不再使用文件时，需要关闭文件

    return 0;
}
```

mode表示对打开的文件的操作方式：

文件使用方式	含义	如果指定文件不存在
“r”（只读）	为了输入数据，打开一个已经存在的文本文件	出错
“w”（只写）	为了输出数据，打开一个文本文件	建立一个新的文件
“a”（追加）	向文本文件尾添加数据	建立一个新的文件
“rb”（只读）	为了输入数据，打开一个二进制文件	出错
“wb”（只写）	为了输出数据，打开一个二进制文件	建立一个新的文件
“ab”（追加）	向一个二进制文件尾添加数据	建立一个新的文件
“r+”（读写）	为了读和写，打开一个文本文件	出错
“w+”（读写）	为了读和写，建议一个新的文件	建立一个新的文件
“a+”（读写）	打开一个文件，在文件尾进行读写	建立一个新的文件
“rb+”（读写）	为了读和写打开一个二进制文件	出错
“wb+”（读写）	为了读和写，新建一个新的二进制文件	建立一个新的文件
	打开一个二进制文件，在文件尾进行读和写	建立一个新的文件

“ab+”（读写）		

4.3.2 fclose

fclose函数原型

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

功能：关闭参数 `stream` 关联的文件，并取消其关联关系。与该流关联的所有内部缓冲区均会解除关联并刷新：任何未写入的输出缓冲区内容将被写入，任何未读取的输入缓冲区内容将被丢弃。

参数：

`stream`：指向要关闭的流的 `FILE` 对象的指针

返回值：成功关闭 `stream` 指向的流会返回0，否则会返回 `EOF` 。

代码演示：

代码块

```
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "r"); //以“r”的形式打开文件，如果文件不存在，则打开失败
    if(fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    printf("打开文件成功，可以对文件进行操作\n");
    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}
```

5. 文件的顺序读写

在进行文件读写的时候，会涉及下面这些函数，我们一一给大家介绍：

函数名	功能	适用于
fgetc	从输入流中读取一个字符	所有输入流
fputc	向输出流中写入一个字符	所有输出流
fgets	从输入流中读取一个字符串	所有输入流
fputs	向输出流中写入一个字符串	所有输出流
fscanf	从输入流中读取带有格式的数据	所有输入流
fprintf	向输出流中写入带有格式的数据	所有输出流
fread	从输入流中读取一块数据	文件输入流
fwrite	向输出流中写入一块数据	文件输出流

上面说的适用于所有输入流一般指适用于标准输入流和其他输入流（如文件输入流）；所有输出流一般指适用于标准输出流和其他输出流（如文件输出流）。

5.1 fputc

fputc函数原型

```
1   int fputc ( int character, FILE * stream );
```

功能：将参数`character`指定的字符写入到`stream`指向的输出流中，通常用于向文件或标准输出流写入字符。在写入字符之后，还会调整指示器。字符会被写入流内部位置指示器当前指向的位置，随后该指示器自动向前移动一个位置。

参数：

`character`：被写入的字符

`stream`：是一个FILE*类型的指针，指向了输出流（通常是文件流或`stdout`）。

返回值：

- 成功时返回写入的字符（以`int`形式）。
- 失败时返回`EOF`（通常是 -1），错误指示器会被设置，可通过`ferror()`检查具体错误。

代码演示：

代码块

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

```c

int main()
{
    FILE* fp = fopen("test.txt", "w"); //以w的形式打开文件，才能正确的写文件
    if(fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    fputc('a', fp);
    fputc('b', fp);
    fputc('c', fp);

    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}


//也可以循环写入多个字符的
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "w"); //以w的形式打开文件，才能正确的写文件
    if(fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    int ch = 0;
    for(ch = 'a'; ch <= 'z'; ch++)
    {
        fputc(ch, fp);
    }

    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}
```

5.2 fgetc

fgetc函数的原型

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

功能：从参数`stream`指向的流中读取一个字符。函数返回的是文件指示器当前指向的字符，读取这个字符之后，文件指示器自动前进到下一个字符。

参数：

`stream`：FILE*类型的文件指针，可以是`stdin`，也可以是其他输入流的指针。如果是`stdin`就从标准输入流读取数据。如果是文件流指针，就从文件读取数据。

返回值：

- 成功时返回读取的字符（以`int`形式）。
- 若调用时流已处于文件末尾，函数返回`EOF`并设置流的文件结束指示器（`feof`）。
  若发生读取错误，函数返回`EOF`并设置流的错误指示器（`ferror`）。

代码演示：

代码块

```c
//我们就写代码，来读取前面test.txt文件中的前10个字符
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "r");
    if(fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    int i = 0;
    for(i = 0; i < 10; i++)
    {
        int c = fgetc(fp);
        fputc(c, stdout); //使用fputc在标准输出流上打印字符
    }

    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}
```

5.3 feof和ferror

代码块

```
int feof ( FILE * stream );
//检测stream指针指向的流是否遇到文件末尾

int ferror ( FILE * stream );
//检测stream指针指向的流是否发生读/写错误
```

如果在读取文件的过程中，遇到了文件末尾，文件读取就会结束。这时读取函数会在对应的流上设置一个文件结束的指示符，这个文件结束指示符可以通过 `feof` 函数检测到。如果 `feof` 函数检测到文件结束指示符已经被设置，则返回非0的值，如果没有设置则返回0。

如果在读/写文件的过程中，发生了读/写错误，文件读取就会结束。这时读/写函数会在对应的流上设置一个错误指示符，这个错误指示符可以通过 `ferror` 函数检测到。如果 `ferror` 函数检测到错误指示符已经被设置，则返回非0的值，如果没有设置则返回0。

测试feof函数

```
#include <stdio.h>
//假设test.txt文件中存放abcdef
int main()
{
    FILE* fp = fopen("test.txt", "r");
    if (fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        int c = fgetc(fp);
        if (c == EOF)
        {
            if (feof(fp))
                printf("遇到文件末尾了\n");
            else if (ferror(fp))
                printf("读取发生了错误\n");
        }
        else
        {
            fputc(c, stdout); //使用fputc在标准输出流上打印字符
        }
    }
```

```
27
28      //不再使用文件时，需要关闭文件
29      fclose(fp);
30      fp = NULL; //将指针置为NULL,避免成为野指针。
31      return 0;
32  }
```

测试ferror函数

```
1   #include <stdio.h>
2   //以写的形式打开文件后，再去读文件，就会发生错误
3   int main()
4   {
5       FILE* fp = fopen("test.txt", "w");
6       if (fp == NULL)
7       {
8           perror("fopen\n");
9           return 1;
10      }
11      int c = fgetc(fp);//读文件
12      if (c == EOF)
13      {
14          if (feof(fp))
15              printf("遇到文件末尾了\n");
16          else if (ferror(fp))
17          {
18              printf("读文件发生了错误\n");
19          }
20      }
21      else
22      {
23          fputc(c, stdout); //使用fputc在标准输出流上打印字符
24      }
25
26      //不再使用文件时，需要关闭文件
27      fclose(fp);
28      fp = NULL; //将指针置为NULL,避免成为野指针。
29      return 0;
30  }
```

5.4 fputs

fputs 函数原型

```
1   int fputs ( const char * str, FILE * stream );
```

功能：将参数`str`指向的字符串写入到参数`stream`指定的流中（不包含结尾的`\0`），适用于文件流或标准输出（stdout）。

参数：

`str`：str是指针，指向了要写入的字符串（必须以`\0`结尾）

`stream`：是一个`FILE*`的指针，指向了要写入字符串的流。

返回值：

- **成功**时返回非负整数。
- **失败**时返回`EOF`（-1），同时会设置流的错误指示器，可以使用`ferror()`检查错误原因。

代码演示：

代码块

```c
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "w");
    if(fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    int i = 0;
    fputs("abc\0def", fp);
    fputs("hehe", fp);
    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}
```

运行结果，文件中存放的信息如下:

代码块

```
abchehe
```

5.5 fgets

fgets函数原型

```
char * fgets ( char * str, int num, FILE * stream );
```

功能：从 `stream` 指定输入流中读取字符串，至读取到换行符、文件末尾（EOF）或达到指定字符数（包含结尾的空字符 `\0`），然后将读取到的字符串存储到str指向的空间中。

参数：

- `str`：是指向字符数组的指针，str指向的空间用于存储读取到的字符串。
- `num`：最大读取字符数（包含结尾的 `\0`，实际最多读取 `num-1` 个字符）。
- `stream`：输入流的文件指针（如文件流或 `stdin`）。

返回值：

- **成功**时返回 `str` 指针。
- 若在尝试读取字符时遇到文件末尾，则设置文件结束指示器，并返回 `NULL`，需通过 `feof()` 检测。
- 若发生读取错误，则设置流错误指示器，并返回 NULL，通过 `ferror()` 检测。

代码演示：

代码块

```c
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "r");
    if (fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    char arr1[] = "**************";
    fgets(arr1, sizeof(arr1), fp);

    char arr2[] = "**************";
    fgets(arr2, sizeof(arr1), fp);

    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}
```

使用细节：

- 若读取到换行符（`\n`），会将其包含在字符串中（除非超出 `num-1` 限制），然后以 `\0` 结尾。
- 文件末尾无换行符时，字符串以 `\0` 结尾，不包含 `\n`。

5.6 fprintf

代码块

```
int fprintf ( FILE * stream, const char * format, ... );
//类比printf函数学习
```

功能： `fprintf` 是将格式化数据写入指定文件流的函数。它与 `printf` 类似，但可以输出到任意输出流（如磁盘文件、标准输出、标准错误等），而不仅限于控制台。

参数：

- `stream`：指向 `FILE` 对象的指针，表示要写入的文件流（如 `stdout`、文件指针等）。
- `format`：格式化字符串，包含要写入的文本和格式说明符（如 `%d`、`%s` 等）。
- `...`：可变参数列表，提供与格式字符串中说明符对应的数据。

返回值：

- 成功时，返回写入的字符总数（非负值）。
- 失败时，先设置对应流的错误指示器，再返回负值，可以通过 `ferror()` 来检测。

代码演示：

代码块

```
#include <stdio.h>
struct Stu
{
    char name[30];
    int age;
    float score;
};

int main()
{
    struct Stu b = {"zhangsan", 20, 85.5f};
    FILE* fp = fopen("test.txt", "w");
    if(fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
```

```
18      fprintf(fp, "%s %d %.1f", b.name, b.age, b.score);
19      //不再使用文件时，需要关闭文件
20      fclose(fp);
21      fp = NULL; //将指针置为NULL,避免成为野指针。
22      return 0;
23  }
```

5.7 fscanf

代码块

```
1   int fscanf ( FILE * stream, const char * format, ... );
2   //类比scanf函数来学习
```

功能： `fscanf` 是从指定文件流中读取格式化数据的函数。它类似于 `scanf`，但可以指定输入源（如文件、标准输入等），而非仅限于控制台输入。适用于从文件解析结构化数据（如整数、浮点数、字符串等）。

参数：

- `stream`：指向 `FILE` 对象的指针，表示要读取的文件流（如 `stdin`、文件指针等）。
- `format`：格式化字符串，定义如何解析输入数据（如 `%d`、`%f`、`%s` 等）。
- `...`：可变参数列表，提供存储数据的变量地址（需与格式字符串中的说明符匹配）。

返回值：

- **成功时**，函数返回成功填充到参数列表中的项数。该值可能与预期项数一致，也可能因以下原因少于预期（甚至为零）：
  - 格式和数据匹配失败；
  - 读取发生错误；
  - 到达文件末尾（EOF）。
- 如果在成功读取任何数据之前发生：
  - 发生读取错误，会在对应流上设置错误指示符，则返回 `EOF`。
  - 到达文件末尾，会在对应流上设置文件结束指示符，则返回 `EOF`。

代码演示：

假设在fprintf函数学习的时候，已经向文件中写入了数据：`zhangsan 20 85.5`，我们现在想在文件中读取这些数据出来。

代码块

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

```c
struct Stu
{
    char name[30];
    int age;
    float score;
};

int main()
{
    struct Stu s = { 0 };
    FILE* fp = fopen("test.txt", "r");
    if (fp == NULL)
    {
        perror("fopen\n");
        return 1;
    }
    fscanf(fp, "%s %d %f", s.name, &(s.age), &(s.score));//按照格式读取数据
    fprintf(stdout, "%s %d %.1f\n", s.name, s.age, s.score);//打印数据在stdout上

    //不再使用文件时，需要关闭文件
    fclose(fp);
    fp = NULL; //将指针置为NULL,避免成为野指针。
    return 0;
}
```

5.8 fwrite

代码块

```c
size_t fwrite ( const void * ptr, size_t size, size_t count, FILE * stream );
```

功能：函数用于将数据块写入 `stream` 指向的文件流中，是以2进制的形式写入的。

参数：

- `ptr`：指向要写入的数据块的指针。
- `size`：要写入的每个数据项的大小（以字节为单位）。
- `count`：要写入的数据项的数量。
- `stream`：指向 `FILE` 类型结构体的指针，指定了要写入数据的文件流。

返回值：返回实际写入的数据项数量。如果发生错误，则返回值可能小于 `count`。

使用注意事项：

- 需要包含 `<stdio.h>` 头文件。
- 在使用 `fwrite()` 之前，需要确保文件已经以二进制可写方式打开。
- `fwrite()` 通常用于二进制数据的写入，如果写入文本数据，请谨慎处理换行符和编码等问题。

代码演示：

假设要将一组整数写入到文件 "data.bin" 中：

代码块

```c
#include <stdio.h>

int main()
{
    int data[] = {1, 2, 3, 4, 5};
    // 打开文件
    FILE *fp = fopen("data.bin", "wb");

    // 检查文件是否成功打开
    if (fp == NULL) {
        perror("fopen");
        return -1;
    }

    // 将数据写入文件
    if (fwrite(data, sizeof(int), 5, fp) != 5) {
        perror("fwrite");
        return -1;
    }

    // 关闭文件
    fclose(fp);
    fp = NULL;
    return 0;
}
```

5.9 fread

代码块

```c
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
```

功能：函数用于从 `stream` 指向的文件流中读取数据块，并将其存储到 `ptr` 指向的内存缓冲区中。

参数：

- `ptr`：指向内存区域的指针，用于存储从文件中读取的数据。
- `size`：要读取的每个数据块的大小（以字节为单位）。
- `count`：要读取的数据块的数量。
- `stream`：指向 FILE 类型结构体的指针，指定了要从中读取数据的文件流。

返回值：返回实际读取的数据块数量。

使用注意事项：

- 需要包含 `<stdio.h>` 头文件。
- 在使用 `fread()` 之前，需要确保文件已经以二进制可读方式打开。
- `ptr` 指向的内存区域必须足够大，以便存储指定数量和大小的数据块。
- 如果 `fread()` 成功读取了指定数量的数据块，则返回值等于 `count`；如果读取数量少于 `count`，则可能已经到达文件结尾或者发生了错误。
- 在二进制文件读取时，`fread()` 是常用的函数，但对于文本文件读取，通常使用 `fgets()` 或 `fscanf()`。

代码演示：

假设有一个二进制文件 "data.bin"，包含一些整数数据，我们将使用 `fread()` 函数读取这些数据：

代码块

```c
#include <stdio.h>

int main()
{
    int data[5] = {0};  // 假设文件中包含 5 个整数数据

    // 打开文件
    FILE *fp = fopen("data.bin", "rb");

    // 检查文件是否成功打开
    if (fp == NULL) {
        perror("fopen");
        return -1;
    }

    // 从文件中读取整数数据块
```

```c
    size_t num_read = fread(data, sizeof(int), 5, fp);

    // 检查读取是否成功
    if (num_read != 5)
    {
        if (feof(fp))
            printf("Reached end of file\n");
        else if (ferror(fp))
            printf("Error reading file\n");
    }
    else
    {
        // 输出读取的数据
        for (int i = 0; i < 5; i++) {
            printf("Data[%d]: %d\n", i, data[i]);
        }
    }

    // 关闭文件
    fclose(fp);

    return 0;
}
```

5.10 对比一组函数：

> scanf/fscanf/sscanf
>
> printf/fprintf/sprintf

5.10.1 sprintf

代码块

```c
int sprintf ( char * str, const char * format, ... );
```

功能：将格式化数据写入字符数组（字符串）。它类似于 `printf`，但输出目标不是控制台或文件，而是用户指定的内存缓冲区。常用于动态生成字符串、拼接数据或转换数据格式。简而言之就是将格式化的数据转换成一个字符串。

参数：

- `str`：指向字符数组的指针，用于存储生成的字符串（需确保足够大以防止溢出）。
- `format`：格式化字符串，定义输出格式（如 `%d`、`%f`、`%s` 等）。

- `...`：可变参数列表，提供与格式字符串中说明符对应的数据。

返回值：

- **成功时**：返回写入 `buffer` 的字符数（不包括结尾的空字符 `\0`）。
- **失败时**：返回负值。

代码演示：

代码块

```
#include <stdio.h>

struct Stu
{
    char name[30];
    int age;
    float score;
};

int main()
{
    struct Stu s = {"zhangsan", 100, 85.5f};
    //将s中的结构数据转换成一个字符串
    char arr[100] = {0};
    sprintf(arr, "%s %d %.1f", s.name, s.age, s.score);
    printf("%s\n", arr);
    return 0;
}
```

5.10.2 sscanf

代码块

```
int sscanf ( const char * str, const char * format, ...);
```

功能：从字符串中读取格式化数据。它与 `scanf` 类似，但输入源是内存中的字符串而非控制台或文件。常用于解析字符串中的结构化数据（如提取数字、分割文本等）。

参数：

- `str`：要解析的源字符串（输入数据来源）。
- `format`：格式化字符串，定义如何解析数据（如 `%d`、`%f`、`%s` 等）。
- `...`：可变参数列表，提供存储数据的变量地址（需与格式字符串中的说明符匹配）

返回值：

- **成功时**：返回成功解析并赋值的参数数量（非负值）。
- **失败或未匹配任何数据**：若输入结束或解析失败，返回 `EOF`（通常是 `-1`）。

代码演示：

代码块

```c
#include <stdio.h>

struct Stu
{
    char name[30];
    int age;
    float score;
};

int main()
{
    char str[] = "zhangsan 100 85.5";
    struct Stu s = {0};
    //从str中的字符串内中解析出格式化的数据，存放在结构体中
    sscanf(str, "%s %d %f", s.name, &(s.age), &(s.score));
    //打印结构体s中的数据
    printf("%s %d %.1f\n", s.name, s.age, s.score);
    return 0;
}
```

scanf	针对标准输入(stdin)的格式化输入函数
printf	针对标准输出(stdout)的格式化输出函数
fscanf	针对所有输入流（可以是文件流，也可以是stdin）的格式化输入函数
fprintf	针对所有输出流（可以是文件流，也可以是stdout）的格式化输出函数
sprintf	将格式化的数据转换成字符串
sscanf	从字符串中提取格式化的数据

6. 文件的随机读写

6.1 fseek

代码块

```c
int fseek ( FILE * stream, long int offset, int origin );
```

功能：根据文件指针的位置和偏移量来定位文件指针（文件内容的光标）。

参数：

- `stream`：指向 FILE 类型结构体的指针，指定了要定位文件指针的文件流。
- `offset`：偏移量，这个偏移量要根据第三个参数来确定取值，可以是正数（向文件末尾方向移动）、负数（向文件开头方向移动）或 0（不移动）。
- `origin`：偏移的起始点（基准位置），这个参数有三个取值：
  - `SEEK_SET`：文件开始位置
  - `SEEK_CUR`：文件指针当前的位置
  - `SEEK_END`：文件末尾位置

返回值：

- 成功：如果文件位置指针被成功移动，`fseek` 返回 0。
- 失败：如果发生错误（例如试图移动到一个不存在的负偏移位置），则返回 非零值（通常是 -1）。

使用注意事项：

- 二进制文件 vs 文本文件：
  - 对于二进制文件，`fseek` 的行为非常直观，可以精确地移动到任何字节位置。
  - 对于文本文件，`fseek` 的行为在某些环境下可能不确定（因为换行符的转换等问题）。唯一被严格保证的用法是：
    - 将 `offset` 设置为 0，`origin` 设置为 `SEEK_SET`（即移动到文件开头）。
    - 将 `offset` 设置为 `ftell()` 的返回值（`ftell` 返回当前指针位置）。

例子：

代码块

```c
/* fseek example */
#include <stdio.h>

int main ()
{
    FILE * pFile;
    pFile = fopen ( "example.txt" , "wb" );
    fputs ( "This is an apple." , pFile );
    fseek ( pFile , 9 , SEEK_SET );
    fputs ( " sam" , pFile );
    fclose ( pFile );
    return 0;
```

```
}
```

6.2 ftell

代码块

```
long int ftell ( FILE * stream );
```

功能：返回文件指针相对于起始位置的偏移量

参数：

- `stream`:指向一个已打开文件的文件指针。

返回值：

- 成功：返回当前文件位置指针相对于文件开头的偏移量（字节数），类型为 `long int`。
- 失败：如果发生错误（例如文件流无效或文件不支持定位），则返回 -1L。

使用注意事项：

- 二进制文件 vs 文本文件：
  - 对于二进制文件，`ftell` 的返回值就是相对于文件开头的精确字节数，非常可靠。
  - 对于文本文件，由于不同系统对换行符的处理不同（如Windows是"\r\n"，Linux是"\n"），`ftell` 的返回值可能不直接等于从文件开头计数的字符数。但在大多数实现中，它仍然可以安全地与 `fseek` 配合使用来记录和恢复位置。

例子：

代码块

```
/* ftell example : getting size of a file */
#include <stdio.h>

int main ()
{
    FILE * pFile;
    long size;
    pFile = fopen ("myfile.txt","rb");
    if (pFile == NULL)
        perror ("Error opening file");
    else
    {
        fseek (pFile, 0, SEEK_END);   // non-portable
        size = ftell (pFile);
        fclose (pFile);
```

```
16          printf ("Size of myfile.txt: %ld bytes.\n",size);
17      }
18      return 0;
19  }
```

6.3 rewind

代码块

```
1   void rewind ( FILE * stream );
```

功能：让文件指针的位置回到文件的起始位置

参数： `stream` :指向一个已打开文件的文件指针。

使用注意事项：

- 清除错误标志：
  - `rewind` 不仅移动文件指针，还会清除文件的错误标志和文件结束标志。
  - 这意味着即使在读取文件时遇到了错误或到达了文件末尾，调用 `rewind` 后，文件状态会被重置，可以重新开始操作。
- 与 `fseek` 的区别：
  - `fseek(fp, 0, SEEK_SET)` 只移动文件指针，不清除错误标志。
  - `rewind(fp)` 移动文件指针并清除错误标志。

例子：

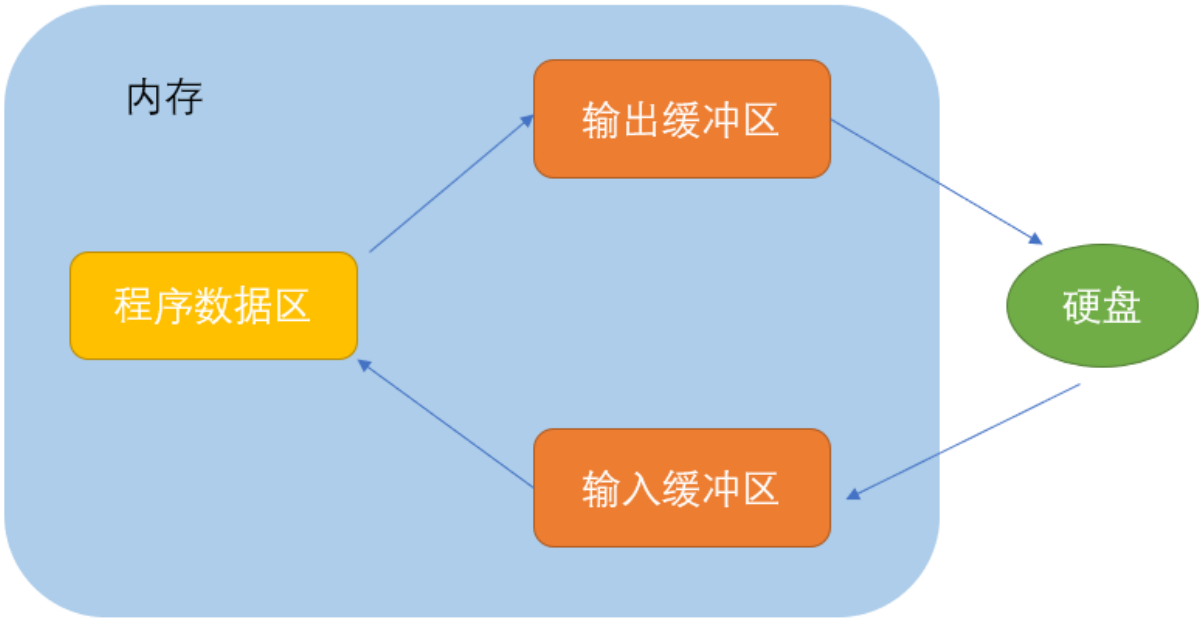
代码块

```
1   /* rewind example */
2   #include <stdio.h>
3
4   int main ()
5   {
6       int n;
7       FILE * pFile;
8       char buffer [27];
9
10      pFile = fopen ("myfile.txt","w+");
11      for ( n='A' ; n<='Z' ; n++)
12          fputc ( n, pFile);
13      rewind (pFile);
14
15      fread (buffer,1,26,pFile);
```

```
16      fclose (pFile);
17
18      buffer[26]='\0';
19      printf(buffer);
20      return 0;
21  }
```

7. 文件缓冲区

ANSI C 标准采用“**缓冲文件系统**” 处理数据文件的，所谓缓冲文件系统是指系统自动地在内存中为程序中每一个正在使用的文件开辟一块“**文件缓冲区**”。从内存向磁盘输出数据会先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘上。如果从磁盘向计算机读入数据，则从磁盘文件中读取数据输入到内存缓冲区（充满缓冲区），然后再从缓冲区逐个地将数据送到程序数据区（程序变量等）。缓冲区的大小根据C编译系统决定的。



7.1 fflush

代码块

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

功能：强制刷新参数 `stream` 指定流的缓冲区，确保数据写入底层设备。

- 对**输出流**：将缓冲区中未写入的数据立即写入文件。
- 对**输入流**：行为由具体实现决定，非C语言标准行为（可能清空输入缓冲区）

- 参数为 `NULL` 时：刷新所有打开的输出流

参数：

`stream`：指向文件流的指针（如 `stdout`、文件指针等）

返回值：成功返回 `0`，失败返回 `EOF`

注意事项：

1. 仅对**输出流**或**更新流**（最后一次操作为输出）有明确刷新行为
2. 输入流的刷新行为不可移植（如清空输入缓冲区是非标准特性）
3. 程序正常终止（`exit`）或调用 `fclose` 时会自动刷新，但程序崩溃时缓冲区数据可能丢失

代码演示：

代码块

```c
#include <stdio.h>
#include <windows.h>
//VS2022 WIN11环境测试
int main()
{
    FILE*pf = fopen("test.txt", "w");
    fputs("abcdef", pf);//先将代码放在输出缓冲区
    printf("睡眠10秒-已经写数据了，打开test.txt文件，发现文件没有内容\n");
    Sleep(10000);
    printf("刷新缓冲区\n");
    fflush(pf);//刷新缓冲区时，才将输出缓冲区的数据写到文件（磁盘）
    //注：fflush 在高版本的VS上不能使用了
    printf("再睡眠10秒-此时，再次打开test.txt文件，文件有内容了\n");
    Sleep(10000);
    fclose(pf);
    //注：fclose在关闭文件的时候，也会刷新缓冲区
    pf = NULL;

    return 0;
}
```

这里可以得出一个**结论**：

因为有缓冲区的存在，C语言在操作文件的时候，需要做刷新缓冲区或者在文件操作结束的时候关闭文件。

如果不做，可能导致读写文件的问题。

8. 更新文件

在文件的打开模式中有三种方式值得注意，分别是: "r+", "w+", "a+"，分别是什么意思?

行为	"r+"	"w+"	"a+"
解释	可读/可写	可读/可写	可读/可写
文件不存在时	打开失败	自动创建新文件	自动创建新文件
文件存在时	保留内容	清空内容	保留内容
初始文件指针位置	文件开头	文件开头	文件末尾
写入是否覆盖原有数据	是（可定位覆盖）	是（内容已清空，从头写入）	否(默认是在文件末尾写数据)
典型用途	修改文件部分内容	创建新文件或完全重写旧文件	在文件末尾追加数据，比如记录日志

关键要点：

1. 在写完文件后，要继续读文件的时候，在读取之前一定要使用 `fflush()` 刷新文件缓冲区，再使用 `fseek()` ， `rewind()` 重新定位文件指示器的位置。
2. 在读完文件后，需要继续写文件之前，在写文件之前，使用 `fseek()` ， `rewind()` 重新定位文件指示器的位置。

代码块

```c
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "w+");
    if (fp == NULL)
    {
        perror("fopen for w+");
        return 1;
    }
    //写abcdefghi到文件中
    fputs("abcdefghi", fp);
    //刷新缓冲区，保证数据写入文件
    fflush(fp);
    //要读取数据b字符，先定位文件指针
    fseek(fp, 1, SEEK_SET);
    int ch = fgetc(fp);//读取字符
    printf("%c\n", ch);
    //在b的位置开始写入hello
```

```
    fseek(fp, -1, SEEK_CUR);
    //解释：因为前面读取一个字符后,文件指示器现在指向了c,需要从当前位置退回一个字符

    fputs("hello", fp);

    //关闭文件
    fclose(fp);
    fp = NULL;

    return 0;
}
```

完