*第九章 备忘录—— 文件操作*

第一节：文件的打开与关闭

第二节：文件的读取与写入

第三节：文件的定位

第四节：文件的格式

一个软件由若干程序和文档组合而成，而绝大多数的软件都有输入与输出的功能，以此来实现与用户的交互。我们常用的软件中，可能有一些C语言编写的，也可能没有。但是有一点是可以确定的，这些软件采用的输入输出方式都不是我们已经习惯的纯键盘操作，它们更多地使用图形用户界面（GUI，**Graphical User Interface**），并且通常都需要使用鼠标。我们对标准输入输出的介绍相对很少，即使是printf这么“常用”的函数，我们都只展示了它很少一部分的功能。毕竟，printf与scanf这样的标准输入输出函数不能给我们带来Visual Studio那样的有图形界面的软件，而像一些根本不需要界面的后台程序（比如mciSendString，在无形中就播放了音乐），也用不上这些函数。printf和scanf本身是非常复杂的，所以我们干脆省下力气，只学习最有用的那部分，学习难度较高又很少用到的就被我们忽略了。

除了输入输出，软件还应该有的就是文件操作的能力，C语言也提供了操作文件的函数，与标准输入输出不同，不管是什么样的C语言程序，它们在操作文件的时候都用的是同样的一些函数。所以我们对文件操作会有更详细的介绍，读者会发现，本章的内容适用于图形界面、控制台程序以及后台程序等C语言程序。

9.1文件的打开与关闭

前面的章节中，我们编写过一些游戏，也编写过音乐播放器。遗憾的是，游戏不能存档，歌单也不能保存。想要实现存档的功能，需要利用文件操作，在关闭程序的时候把存档信息保存到文件中，下次启动程序再读取出来，然后就可以接着玩游戏了。这就好像我们睡一觉起来会忘记很多计划好的事，而如果我们在睡觉前把容易忘记的事写到备忘录里，第二天起来看看备忘录就能想起来。

那么到底怎样操作文件呢？我们首先来看文件的打开和关闭，这是最简单的文件操作。

1. #include <stdio.h>
2. int main()
3. {
4. FILE \*p;
5. p = fopen("c:/a.txt", "r");
6. if (p == NULL)
7. {
8. printf("文件打开失败\n");
9. }
10. else
11. {
12. printf("文件打开成功\n");
13. fclose(p);
14. }
15. getchar();
16. return 0;
17. }

程序9-1-1

程序的某次运行结果如下：

文件打开失败

我们先不管为什么会打开失败，先分析分析程序本身。

FILE是一个结构体，用来表示文件的结构体，该结构体在stdio.h中被声明，需要注意的是，我们并没有实例化FILE结构体，而是声明了它的指针p，之所以这样做是因为文件操作函数大都以FILE \*而不是FILE为参数（或返回值）。

接下来我们调用了fopen函数，试图打开一个文件，可以看到，fopen函数有两个参数，两个参数的类型都是char \*。第一个参数表示要打开的文件的路径，这个可以根据实际情况来指定，我们指定的是e:/a.txt。第二个参数则表示打开方式，r表示以只读的方式打开文件，文件的打开方式有非常多种，我们稍后会做详细介绍。如果fopen函数成功打开了一个文件，那么它会返回一个FILE型的指针（FILE \*），我们可以通过这个指针来操作成功打开的文件；而如果fopen未能成功打开一个文件，它就会返回NULL。

我们能通过判断p是否为NULL来判断文件是否打开成功，不管是否打开成功，我们都输出提示信息，而如果打开成功了，我们还需要在程序结束前用fclose来关闭已经打开的文件。在程序运行的时候，c盘目录下并没有一个名为a.txt的文件，所以文件打开失败也就是理所当然的了，如果我们在c盘新建一个a.txt的文件，然后再运行程序，程序就会提示打开成功。不过，文件存在并不一定就能打开成功，因为操作系统的有些重要文件是不允许用户访问的，试图打开这样的文件同样会失败。

可以看到，文件的打开与关闭操作的流程类似于内存的申请与释放操作，fopen对应malloc，它们的返回值都是一个指针，同样地，操作失败会返回NULL，而fclose则对应free，当p为非法的值的时候，操作同样会失败。

**小提示**

打开一个文件与运行一个文件的概念是不一样的。假设有一个文件c:/a.exe，当我们打开它的时候，只需要把它加载到内存中，然后创建对应的FILE指针即可，我们不必关心它的具体内容。而如果我们要运行这个文件，不仅要把它加载到内存中，还要找到它的主函数作为入口，然后按照特定的规则来执行它包含的内容。C语言标准库并不提供运行文件的函数（因为每种格式的文件运行方式不同，我们甚至可以自己创建一种格式，显然我们不能要求C语言标准支持我们创建的格式）。Windows提供的ShellExcute可以用来运行exe文件，我们在前面的章节中也有所介绍。

前面我们说了，文件的打开方式分为很多种，我们已经见过的r表示以只读的方式打开，与只读对应的打开方式还有只写和读写。只写对应w，读写则对应r+或者w+。我们知道，当打开方式为r的时候，fopen会首先检测文件是否存在，存在则尝试打开，不存在则直接返回NULL，当打开方式为r+时也是类似的。而w和w+则不同于r和r+，通过这两种打开方式，fopen会直接新建一个文件，若文件已经存在则会覆盖并清空之前已经存在的文件。

1. #include <stdio.h>
2. int main()
3. {
4. FILE \*p;
5. p = 0;
6. p = fopen("c:/a.txt", "w");
7. if (p == NULL)
8. {
9. printf("文件打开失败，请检查权限\n");
10. }
11. else
12. {
13. printf("文件打开成功\n");
14. fclose(p);
15. }
16. getchar();
17. return 0;
18. }

程序9-1-2

在某台计算机上的运行结果如下：

文件打开成功

有些系统会禁止用户的程序在c盘目录下新建文件，在这种情况下程序会提示文件打开失败，如果读者发现了这种情况，可以尝试将c:/a.txt修改成d:/a.txt（如果读者的计算机上有d盘的话）。

我们会在下一节介绍从文件中读数据与往文件中写数据的函数，当我们用只读的方式打开文件时，只能从文件里读取数据，而不能向其中写入数据（只能调用读取数据的函数来操作该文件），以只写的方式打开文件也是一样的道理。而用读写的方式打开会让我们既可以读，又可以写，只是用读写的方式打开文件会比用只读或只写的方式花费更多的资源。

除了r和w，还有一种打开方式：a，a的作用是往文件尾追加数据，事实上也相当于是只写，当文件不存在时，它会新建一个文件，而当文件已经存在，它会打开已有的文件，并定位到文件末尾处，这样就能在文件末尾处接着写数据了，关于文件定位的概念会在第3节有所介绍。此外，a也可以和+相组合，如果用a+的方式打开文件，我们就能读取文件中的数据。

现在，我们可以清点一下文件的打开方式。r、w、a是最基本的3种方法，而它们又可以分别和+相组合，这样就又多了r+、w+、a+这3种。

我们将r、w、a理解为基本方式，将+理解为附加值。除了+以外，还有另外两个附加值，即b和t。如果r和b组合，就表示用二进制的方式打开文件，如果r和t组合，则表示用文本的方式打开文件。而默认的打开方式是t，也就是说，t可以被省略，我们也建议读者将之省去。需要注意的是，b和+、t和+都可以同时存在，而b和t则不可以同时存在，这很好理解，就像r和w不能同时存在一样。

那么，用二进制的方式打开和用文本的方式打开有什么不同呢？在大多数情况下，两者都是相同的，但是在某些特定的情况，它们却有巨大的不同。比如二进制文件中可以出现-1，而文本文件中不会出现-1（虽然我们知道中文是负数，但是可以保证的是不会有-1），此外，在对回车符的处理上，也有细微的不同，在介绍文件的读写时我们会深入探讨这个问题。我们可以用打开文本文件的方式来打开二进制文件，也可以用打开二进制文件的方式来打开文本文件，只不过这样会导致潜在的不安全因素。所以在打开一个文件之前，通常需要搞清楚它到底是文本文件还是二进制文件。在常见的文件格式中，exe属于二进制文件，txt则属于文本文件。

为了方便读者记忆和理解，我们列出各种打开方式的全称：

r read 只读

w write 只写

a append 追加

t text 文本文件

b binary 二进制文件

+ 读+写

现在我们再来数一数文件打开方式总共有多少种，r、w、a是3种基本方式，它们可以分别和+组合，也可以分别和b组合，甚至还可以同时和+和b组合，因此打开方式一共有3+3+3+3=12种。如下表所示：

|  |  |  |
| --- | --- | --- |
| 打开方式 | 权限 | 具体操作 |
| r | 只读 | 打开文本文件，文件不存在则打开失败，文件只可读 |
| w | 只写 | 新建文本文件，文件已存在则覆盖之前的文件，文件只可写。 |
| a | 只写 | 打开文本文件并定位到文件尾，文件不存在则新建，文件只可写。 |
| r+ | 读写 | 打开文本文件，文件不存在则打开失败，文件可读写。 |
| w+ | 读写 | 新建文本文件，文件已存在则覆盖之前的文件，文件可读写。 |
| a+ | 读写 | 打开文本文件并定位到文件尾，文件不存在则新建，文件可读写。 |
| rb | 只读 | 打开二进制文件，文件不存在则打开失败，文件只可读。 |
| wb | 只写 | 新建二进制文件，文件已存在则覆盖之前的文件，文件只可写。 |
| ab | 读写 | 打开二进制文件并定位到文件尾，文件不存在则新建，文件只可写。 |
| rb+ | 读写 | 打开二进制文件，文件不存在则打开失败，文件可读写。 |
| wb+ | 读写 | 新建二进制文件，文件已存在则覆盖之前的文件，文件可读写。 |
| ab+ | 读写 | 打开二进制文件并定位到文件尾，文件不存在则新建，文件可读写。 |

表9-1-1

这个表格看起来非常复杂，初学者想要完整地背下来有相当难度，所以我们不建议读者强行记忆，而是把它当成一个“字典”，遇到不会的内容再来查阅。当读者有了足够多的练习之后，自然就能轻易写出这个表格了。

我们可以看出，各种打开方式之间的区别无非在这几个方面：读写权限、二进制或文本文件、新建或打开已有的文件。读者在以后的练习中，可以根据自己的需求，通过上表迅速找到这3方面都满足的打开方式。

9.2文件的读取与写入

光知道怎么打开和关闭文件是不够的，学会文件的读取与写入操作才能真正地让文件操作在程序中派上用场。在这一节中，我们将介绍8个不同的文件读取与写入的函数，这些函数每两个分为一组，一共4组，分别是：fputc和fgetc、fputs和fgets、fprintf和fscanf、fwrite和fread，它们提供了4种不同类型的读写操作。

首先来看fputc和fgetc，前者的作用是往文件中写入一个字符（事实上是写入一个字节的数据，而一个char型数据正好占用一个字节），后者则用来从文件中读取一个字节的数据。

我们来看看程序演示：

1. #include <stdio.h>
2. int main()
3. {
4. FILE \*file1=fopen("d:/a.txt","w");
5. char \*str = "Hello World\n";
6. if (file1 == NULL)
7. {
8. printf("文件打开失败\n");
9. getchar();
10. return 0;
11. }
12. for (char \*p = str; \*p != 0; p++)
13. {
14. fputc(\*p, file1);
15. }
16. fclose(file1);
17. file1 = fopen("d:/a.txt","r");
18. if (file1 == NULL)
19. {
20. printf("文件打开失败\n");
21. getchar();
22. return 0;
23. }
24. for (char temp; ;)
25. {
26. temp=fgetc(file1);
27. if (temp == EOF)
28. {
29. break;
30. }
31. putchar(temp);
32. }
33. fclose(file1);
34. getchar();
35. return 0;
36. }

程序9-2-1

程序的运行结果如下：

Hello World

在输出了Hello World字符串之后，还输出了一个回车符，所以我们能看见有一行空行。此时读者打开d盘会发现多了一个a.txt的文件（如果之前已经有a.txt的话，读者务必做好备份，因为程序9-2-1会将之覆盖），打开该文件看到的内容和运行结果一样，同样是字符串加回车。

由于c盘可能会限制用户创建文件，所以我们在以后的程序中，一律使用d盘，之所以这样做是因为没有d盘的计算机非常少，限制用户在c盘创建文件的系统却非常多。

我们首先用file1来新建了一个只写的文件，然后用fputc不断往里面写入数据，需要注意的是，str指向的数组里有结束符0，而0是一个不可显示的字符，文本文件中不应该出现它，所以我们在遇到结束符之后立即停止写入。可以看到，fputc有两个参数，第一个参数是需要写入的数据，第二个参数是目标文件的文件指针。

完成写入的操作之后，我们关闭文件，然后重新用只读的方式打开文件。接下来便是用fgetc读取数据的循环。在循环中，我们不断用temp接受fgetc的返回值，然后对temp的值进行判断，如果temp的值是EOF，则立即跳出循环并关闭文件，否则用putchar输出temp的值。可以看到，fgetc和getchar很类似，只不过它有一个参数，这个参数就是要操作的文件。而EOF是一个宏，原型是：#define EOF -1，意思是End Of File，也就是文件结束的标记。前面我们说过，文本文件中不会出现-1，所以C语言中把-1作为文件结束符，当文件中的数据已经被全部读取后，我们再次调用fgetc函数，该函数就会返回-1，所以fgetc的返回值是否为-1可以作为文件是否结束的判断依据。

当我们从文件中读取出一个字符后，再次读取的时候，fgetc会返回下一个字符，这样才能保证我们不断读取文件，最终能将文件中的数据全部读取。对此，我们可以理解FILE结构体中有一个“游标”，它会记录当前应该读取的数据的位置，我们当然也可以调节这个游标，这是下一节的内容。

我们知道，txt是文本文件格式，但我们同样可以用打开二进制文件的方式来打开它。接下来我们将在程序9-2-1的基础上稍微做一点修改，以此来演示二进制文件和文本文件的区别所在。

1. #include <stdio.h>
2. int main()
3. {
4. FILE \*file1=fopen("d:/a.txt","w");
5. char \*str = "Hello World\n";
6. if (file1 == NULL)
7. {
8. printf("文件打开失败\n");
9. getchar();
10. return 0;
11. }
12. for (char \*p = str; \*p != 0; p++)
13. {
14. fputc(\*p, file1);
15. }
16. fclose(file1);
17. file1 = fopen("d:/a.txt","rb");//打开方式变成了rb
18. if (file1 == NULL)
19. {
20. printf("文件打开失败\n");
21. getchar();
22. return 0;
23. }
24. for (char temp; ;)
25. {
26. temp=fgetc(file1);
27. if (feof(file1))//文件结束的判断方式也有变化
28. {
29. break;
30. }
31. printf("%d ",temp);//有些字符不方便显示，转换为数字更为直观
32. }
33. fclose(file1);
34. getchar();
35. return 0;
36. }

程序9-2-2

程序运行结果如下：

72 101 108 108 111 32 87 111 114 108 100 13 10

我们用文本文件的方式新建了a.txt，然后向里面写入了数据，之后再用二进制的方式打开文件并读取数据，可以看出，当文件的打开方式为二进制的时候，我们判断文件是否结束不再使用EOF，而是使用feof函数，feof函数的作用是判断一个文件是否已经结束，没有结束则返回0，否则返回非0（可以理解为没有结束返回假，结束则返回真），feof适用于文本文件和二进制文件。

可以看到，这些数字分别对应Hello World的ASCII码，奇怪的是，除了有回车符’\n’所对应的13，还多了一个换行符’\r’，即ASCII的10。明明我们往a.txt里面写入数据的时候，只写了’\n’，为什么用二进制的方式来读取就会多一个\r呢？这和操作系统有一定关系，不同的操作系统对于“回车”的定义各不相同，我们知道，回车的愿意是让光标重新回到行首，而当我们输出一个回车符’\n’的时候，光标不仅回到了行首，还来到了新的一行。而我们输出\r’的时候，却发现光标回到了行首，并没有换行。

事实上，Windows对于这两个字符的处理非常特别。理论上讲，ASCII码的13对应于’\n’，也就是回车，而10对应’\r’，也就是换行。但是Windows下，’\r’被认为是13，也就是回车符，而’\n’则被认为是换行符。关于这一点理解起来有点困难，可以只做了解。

再回到程序9-2-2，当我们往文本文件中写入’\n’时，实际上写入的是13和10，而当我们从文本文件中读取到13和10的时候，会被自动转换为10（换行符，也就是Windows下的’\n’）。不过，虽然以文本文件的方式只能读取出换行符，但以二进制的方式却能全部读取不会做转化，这也就是为什么程序9-2-2中会多读取出一个字符。

可以看出，如果我们把文本文件当成二进制文件打开，或者反过来，都可能因为回车符的特殊性而出现问题，所以文件打开方式一定要和它本身匹配。

Windows下对’\n’和’\r’的处理非常特殊，但我们需要知道的是，13永远对应回车符，10永远对应换行符，只是\n’和’\r’相互颠倒了。

1. #include <stdio.h>
2. int main()
3. {
4. printf("%d %d",'\n','\r');
5. printf("%c",13);
6. getchar();
7. return 0;
8. }

程序9-2-3

程序运行结果如下：

10 13

同时，我们可以看到光标来到了行首。虽然ASCII码表上’\n’对应13，但是在Windows下却被视为10。不过，13对应的依然是回车而不是换行。

我们再来看看fputs和fgets，这两个函数类似于puts和gets（gets\_s）。fputs用来往文件中写入字符串，fgets则从文件里读取字符串，显然，fputs和fgets更适合用来操作文本文件。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.txt", "w");
7. char buffer[25];
8. fputs("这是往文件中写入的字符串", file1);
9. fclose(file1);
10. file1=openFile("d:/a.txt","r");
11. fgets(buffer, 25, file1);
12. printf("%s", buffer);
13. fclose(file1);
14. getchar();
15. return 0;
16. }
17. FILE \* openFile(char \*route, char \*mode)
18. {
19. FILE \*temp=fopen(route,mode);
20. if (temp == NULL)
21. {
22. printf("文件打开失败\n");
23. getchar();
24. exit(0);
25. }
26. return temp;
27. }

程序9-2-4

程序运行如下：

这是往文件中写入的字符串

程序9-2-4中，我们自己编写了一个openFile函数来打开文件，它的两个参数对应fopen的两个参数，打开文件的操作实际上也是调用fopen完成的，只是openFile函数有检查文件打开状态的功能，文件打开失败的情况下它会直接结束程序，以免发生更大的错误。

fputs的原型是：int fputs(char \*str,FILE \*file)，str是要写入的字符串，file是目标文件。和puts一样，当fputs遇到结束符0的时候停止，即fputs不会往文件中写入结束符，若写入成功则返回0，否则返回EOF（-1）。

fgets的原型是：char \*fgets(char \*buffer,int sizeOfBuffer,FILE \*file)，buffer是用来接受数据的缓冲区，sizeOfBuffer则是缓冲区的大小，需要注意的是，fgets最多只会读取sizeOfBuffer-1个字节的数据，最后的一个字节会用来容纳结束符0。此外，如果fgets遇到结束符0、回车符或者文件结束，都会停止读取，然后往缓冲区里填充一个0。若读取成功，则返回buffer，否则返回NULL。

在操作文本文件的时候，fputs和fgets非常好用，但是在操作二进制文件的时候它们的表现就不是那么好了。

fprintf和fscanf是格式化文件操作函数（printf和scanf则是格式化输入输出函数），我们知道，printf和scanf的功能很强大，有很强的格式控制能力。同样地，fprintf和fscanf也能很好地控制文件的读取与写入的格式。

fprintf和fscanf看起来很像我们之前见过的sprintf，除了有格式控制字符和参数以外，还需要一个额外的参数，在sprintf中，这个额外的参数是接收数据的缓冲区，在fprintf和fscanf中自然就是文件指针了。

先来看看fprintf，它比fscanf更容易使用，需要注意的地方更少。我们只需要记住，通过fprintf写到文件中的数据都是一个个的字符。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.txt","w");
7. int num = 10;
8. fprintf(file1, "%d", num);
9. fclose(file1);
10. file1 = fopen("d:/a.txt", "r");
11. for (char temp;;)
12. {
13. temp = fgetc(file1);
14. if (feof(file1))
15. {
16. break;
17. }
18. printf("%d ",temp);
19. }
20. fclose(file1);
21. getchar();
22. return 0;
23. }
24. FILE \* openFile(char \*route, char \*mode)
25. {
26. FILE \*temp=fopen(route,mode);
27. if (temp == NULL)
28. {
29. printf("文件打开失败\n");
30. getchar();
31. exit(0);
32. }
33. return temp;
34. }

程序9-2-5

程序运行结果如下：

49 48

可以看到，文件中一共有2个字节的数据，分别是’1’和’0’，我们知道，int型数据的大小是4，而文件的大小只有2，这说明fprintf把整数的10转换成了字符的’1’和’0’，这和printf的本质是一样的，虽然printf显示在屏幕上的看起来是一个数字，但是实际上也是一个个的字符，就像我们打开a.txt看到的结果一样。

我们可以得出这样一个结论，通过fprintf写到文件中的数据等价于通过printf显示到屏幕上的数据，这样一来就非常好理解了。

fscanf相对特殊一点，如果把fscanf看成scanf，那么文件就相当于是键盘。我们知道，对于scanf来说，’\t’、’\n’和空格都算是分隔符，当遇到分隔符的时候scanf就会停止正在读取的变量，接着读取下一个，此外，当读取到一个类型不匹配的数据时，也会停止读取，开始读取下一个，比如有这样的语句:

scanf(“%d%s”,&num,str);

而我们在输入**123.** 后按回车，那么num的实际值是123，因为.不是数据，所以不被接受。同样地，fscanf也有这样的机制，特别地，当遇到文件结束，fscanf也会停止读取，也就是说文件结束也相当于一个分隔符。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.txt","w");
7. char buffer[10];
8. int num1, num2;
9. fprintf(file1, "10\n str 9.9");
10. fclose(file1);
11. file1 = fopen("d:/a.txt", "r");
12. fscanf(file1,"%d%s%d",&num1,buffer,&num2);
13. printf("%d %s %d",num1,buffer,num2);
14. fclose(file1);
15. getchar();
16. return 0;
17. }

1. FILE \* openFile(char \*route, char \*mode)
2. {
3. FILE \*temp=fopen(route,mode);
4. if (temp == NULL)
5. {
6. printf("文件打开失败\n");
7. getchar();
8. exit(0);
9. }
10. return temp;
11. }

程序9-2-6

程序的运行结果如下：

10 str 9

我们往文件中写入的数据是："10\n str 9.9"，接下来，我们试图从文件中读取两个整数和一个字符串，顺序是整数-字符串-整数，可以看到，由于’\n’和空格是分隔符，所以当读取到’\n’的时候，立即终止对num1的读取，然后开始读取buffer，除了’\n’，空格也是分隔符，所以实际上从’s’开始才对buffer中的元素进行赋值，同样在读取到空格之后停止对buffer的读取，最后开始读取num2，遇到’.’之后终止。

使用fprintf和fscanf来读写文件能让我们以熟悉的方式进行程序设计，但是它们身上有和printf、scanf类似的缺陷，那就是数据需要经常在二进制和字符之间转换，当我们用fprintf往文件中写入一个整数的时候，整数会先被转换成字符，然后写入，fscanf也是类似的道理。并且，使用fscanf需要对文件的内容相当熟悉，知道什么时候会出现“整数”，什么时候会出现分隔符和字符串，不然随时可能出现读取错误。

最后再来说说fwrite和fread，这是一组很常用的函数，在特定的情况下，它们不如前面3组函数方便，但通用性却很强。

fwrite的作用是往文件中写入整块的数据，它的原型是：int fwrite(void \*buffer,int size,int count,FILE \*file)，其中buffer是数据的来源，fwrite会从buffer指向的地址处得到数据并写入文件中，size是写入文件中的数据块的大小，count则是数据块的数量，file自然就是文件指针了。所以我们可以看出，buffer中应该有至少size\*count个字节的数据。fwrite的返回值是写入成功的数据块的数量。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.txt","w");
7. char buffer[20] = "测试fwrite的字符串";
8. fwrite(buffer, 1, 18, file1);
9. fclose(file1);
10. file1 = fopen("d:/a.txt", "r");
11. for (char temp;;)
12. {
13. temp = fgetc(file1);
14. if (feof(file1))
15. {
16. break;
17. }
18. putchar(temp);
19. }
20. fclose(file1);
21. getchar();
22. return 0;
23. }
24. FILE \* openFile(char \*route, char \*mode)
25. {
26. FILE \*temp=fopen(route,mode);
27. if (temp == NULL)
28. {
29. printf("文件打开失败\n");
30. getchar();
31. exit(0);
32. }
33. return temp;
34. }

程序9-2-7

程序运行结果如下：

测试fwrite的字符串

可以看出，字符串的长度是19（包含结束符），而我们在往文件中写入数据的时候没有写入结束符，因此实际上写入的是18个字节。事实上，第8行的代码中，我们也可以改成这一段：fwrite(buffer, 18, 1, file1)，这样会影响fwrite的返回值和每次往文件中写的数据的大小，但是结果是一样的。

再来看看fread，它的原型是：int fread(char \*buffer,int size,int count,FILE \*file)，用法和fwrite几乎一样，只不过buffer从提供数据变成了接受数据。对于fwrite来讲，交换第2和第3个参数的影响不是很大，或许会产生效率问题，但是结果不会有变。但是对于fread来说，交换这两个参数不仅有可能影响效率，还可能让我们无从判断读取数据的情况，我们先演示fread的用法，再探讨为什么两个参数的顺序很重要。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <memory.h>
4. FILE \* openFile(char \*, char \*);
5. int main()
6. {
7. FILE \*file1 = openFile("d:/a.txt","w");
8. char buffer[20] = "测试fwrite的字符串";
9. fwrite(buffer, 1, 18, file1);
10. fclose(file1);
11. file1 = fopen("d:/a.txt", "r");
12. memset(buffer,0,20);
13. fread(buffer, 1, 18, file1);
14. printf("%s",buffer);
15. fclose(file1);
16. getchar();
17. return 0;
18. }
19. FILE \* openFile(char \*route, char \*mode)
20. {
21. FILE \*temp=fopen(route,mode);
22. if (temp == NULL)
23. {
24. printf("文件打开失败\n");
25. getchar();
26. exit(0);
27. }
28. return temp;
29. }

程序9-2-8

程序运行结果如下：

测试fwrite的字符串

第12行我们用memset重新设置了buffer中的数据，然后用fread来往里面读取数据。需要注意的是，fread读取出数据后，不会往里面填充结束符，程序9-2-8中我们之所以没有把buffer[18]设为0，是因为memset已经让它为0了。

现在我们了解了fread的基本用法，现在我们要讨论，为什么最好不要交换第2和第3个参数。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <memory.h>
4. #include <math.h>
5. FILE \* openFile(char \*, char \*);
6. int main()
7. {
8. FILE \*file1 = openFile("d:/a.txt","w");
9. char buffer[10];
10. int num;
11. fwrite("string", 1, 6,file1);
12. fclose(file1);
13. file1 = openFile("d:/a.txt","r");
14. memset(buffer, 0, 10);
15. num=fread(buffer, 1, 20, file1);
16. printf("读取成功了%d个数据\n",1\*num);
17. fclose(file1);
18. file1 = openFile("d:/a.txt", "r");
19. memset(buffer, 0, 10);
20. num = fread(buffer, 20, 1, file1);
21. printf("读取成功了%d个数据\n", 20 \* num);
22. fclose(file1);
24. getchar();
25. return 0;
26. }
27. FILE \* openFile(char \*route, char \*mode)
28. {
29. FILE \*temp=fopen(route,mode);
30. if (temp == NULL)
31. {
32. printf("文件打开失败\n");
33. getchar();
34. exit(0);
35. }
36. return temp;
37. }

程序9-2-9

程序运行结果如下：

读取成功了6个数据

读取成功了0个数据

可以看出，a.txt中一共有6个字节的数据，如果我们每次读取1个，一共读取20次，那么只有6个能成功匹配，所以函数的返回值是6。而如果我们每次读取20个，一共读取1次，那么这唯一的一次读取不会匹配成功，所以函数的返回值是0（虽然不管用哪种方式，buffer中都会被填充6个字节的数据）。从运行结果不难看出，用第一种方式，即使20次读取不能全部匹配，我们也能算出读取成功的数据数，而用第二种方式就做不到了。

fwrite和fread适用于文本文件和二进制文件，尤其是在往文件中写入整个结构体的时候。

9.3文件的定位

在前面的程序实例中，我们经常会在同一个程序里先后读和写一个文件，我们的做法是在写完文件之后，关闭文件，然后重新用只读的方式打开，这样显然特别麻烦。我们虽然可以用读写的方式打开一个文件，但是即使这样做，我们在写完文件之后，文件也被定位到了末尾，此时我们读不出任何数据，那么我们有没有办法主动调整文件的定位呢？答案当然是可以的，这就是本节要探讨的内容。

首先来看怎样做到在只打开一次文件的情况下，往文件中写入数据然后读取出来。这里要用的函数就是rewind，rewind是我们非常熟悉的一个函数，我们曾用它来“清空输入缓冲区”，事实上，它的作用是“重置目标文件指针”，让文件重新定位到文件开头处。在Windows下，stdin也被视为一种文件。所以rewind(stdin)能让stdin指向缓冲区的开头处，虽然没能清空缓冲区，但之后输入的数据会覆盖掉已经存在的，间接地达到了清空的目的。

rewind的原型是：void rewind(FILE \*file)，我们也很熟悉它的用法了。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.txt", "w+");
7. char buffer[25] = "测试rewind函数的字符串";
8. fwrite(buffer,1,22,file1);
9. rewind(file1);
10. for (char temp;;)
11. {
12. temp = fgetc(file1);
13. if (feof(file1))
14. {
15. break;
16. }
17. putchar(temp);
18. }
19. fclose(file1);
20. getchar();
21. return 0;
22. }
23. FILE \* openFile(char \*route, char \*mode)
24. {
25. FILE \*temp = fopen(route, mode);
26. if (temp == NULL)
27. {
28. printf("文件打开失败\n");
29. getchar();
30. exit(0);
31. }
32. return temp;
33. }

程序9-3-1

程序的运行结果如下：

测试rewind函数的字符串

我们选用的打开文件的方式是w+，因为我们想要让文件指针有读写的功能，同时又要新建一个文件。接下来我们用fwrite往文件中写入了22个字符（不加上结束符的话，“测试rewind函数的字符串”的长度正好是22）。写完数据之后，文件被定位在文件尾，这时我们用rewind来重置了file1，然后文件就被定位到了开头处，这时我们就能读取出之前写入的数据了。

rewind的功能仅限于重置文件指针，达不到更多定位的功能。C语言标准库还提供了另外一个功能更强的文件定位函数，即fseek，它的原型是：void fseek(FILE \*file,int offset,int origin)，origin是指定位搜索的起始点，而offset则是相对于起始点的偏移量。起始点只有3个固定的值：SEEK\_SET、SEEK\_CUR、SEEK\_END，分别表示文件开头、当前位置、文件末尾，它们事实上是3个宏，分别对应于0、1和2。如fseek(file,0,SEEK\_SET)表示定位到文件开头处，fseek(file,10,SEEK\_CUR)表示定位到当前位置之后的10个字节的地方，fseek(file,-10,SEEK\_END)表示定位到文件末尾的前10个字节处。

有些文件非常特殊，它们在文件开头处有一些描述文件的信息，而这些信息通常都不定长，所以这些文件都会在把描述信息的长度写入文件，这样当有程序读取文件的时候，就知道该从什么地方开始读取数据。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.dat", "wb+");
7. char buffer[15] = "文件描述信息";
8. char data[20] = "存在文件中的数据";
9. int len=12;
10. fwrite(&len,4,1,file1);
11. fwrite(buffer, 1, 12, file1);
12. fwrite(data, 1, 16, file1);
13. fseek(file1,0,SEEK\_SET);
14. fread(&len, 4,1,file1);
15. fseek(file1, len, SEEK\_CUR);
16. for (char temp;;)
17. {
18. temp = fgetc(file1);
19. if (feof(file1))
20. {
21. break;
22. }
23. putchar(temp);
24. }
25. fclose(file1);
26. getchar();
27. return 0;
28. }
29. FILE \* openFile(char \*route, char \*mode)
30. {
31. FILE \*temp = fopen(route, mode);
32. if (temp == NULL)
33. {
34. printf("文件打开失败\n");
35. getchar();
36. exit(0);
37. }
38. return temp;
39. }

程序9-3-2

程序运行结果如下：

存在文件中的数据

我们这次打开文件的方式不再是文本文件，而是二进制文件，文件后缀也从.txt变成了.dat。.dat算是我们自定义的一种格式，我们可以随意指定它的标准。buffer中储存的是文件的描述信息，我们可以在这里面写入文件的作者，文件的完成日期等等，很明显buffer中的数据的长度是12（当然也可以用strlen来计算这个长度）。我们首先往文件中写入了描述信息的长度，然后写入了描述信息，最后再写入所谓的“有用的数据”。

接着我们重置了file1，模拟读取文件的过程，我们首先读取文件的前4个字节，我们知道这里面储存着描述信息的长度，接着我们用fseek来调整了文件的位置，fseek(file1, len, SEEK\_CUR)表示以当前位置为基准，向后移动len个字节，这时候文件就被定位到了“有用的数据”开始的地方，所以我们才能成功读取到所有的数据。

最后要介绍一个函数，它不能定位文件，但是却能得到文件的当前位置，通常可以用它来计算文件的大小。

1. #include <stdio.h>
2. #include <stdlib.h>
3. FILE \* openFile(char \*, char \*);
4. int main()
5. {
6. FILE \*file1 = openFile("d:/a.dat", "rb");
7. int len;
8. fseek(file1, 0, SEEK\_END);
9. len = ftell(file1);
10. printf("d:/a.dat的大小是：%d个字节",len);
11. fclose(file1);
12. getchar();
13. return 0;
14. }
15. FILE \* openFile(char \*route, char \*mode)
16. {
17. FILE \*temp = fopen(route, mode);
18. if (temp == NULL)
19. {
20. printf("文件打开失败\n");
21. getchar();
22. exit(0);
23. }
24. return temp;
25. }

程序9-3-3

程序的某次运行结果如下：

d:/a.dat的大小是：32个字节

这里的a.dat就是程序9-2-2中新建的文件，所以它的大小是32个字节。

ftell的原型是：int ftell(FILE \*file)，它的返回值即使文件的当前定位相对于文件头的偏移量，而如果我们把文件定位到文件尾，求出来的偏移量自然就是文件的大小了。

9.4文件的格式

文件的格式多种多样，纯文本文件txt、音乐文件mp3、视频文件mp4、可执行文件exe等，这些文件的格式各不相同。其中最简单的就是txt，想要新建一个txt文件非常简单，只需要写这样的代码就够了：fopen(“a.txt”,”w”)。那么，这是否意味着，新建mp3文件只需要把.txt改成.mp3就够了？答案是否定的，因为mp3文件和txt的区别不仅仅在于后缀，它们内部的信息也是不同的，更重要的是，txt是文本文件，mp3是二进制文件，想要新建一个mp3文件，必须了解它的编码格式。

**小提示**

初学者常常会问这样的问题： C语言支持mp3格式吗？答案当然是肯定的，只是C语言的标准库不支持mp3而已。我们前面自定义了一种名叫dat的文件格式，除了我们，没人知道它到底是怎么组织的，所以我们肯定不能要求C语言标准库支持我们的格式。mp3也是一样的，虽然它名气很大，但它也只是一种用户自定义的格式，像这样的格式数不胜数，所以我们不能要求C语言标准库为我们提供解析mp3的库函数，所以说C语言标准库不支持mp3。但是一旦我们从格式作者那里了解到了mp3的规范，我们就能用C语言编写对应的程序来对mp3文件进行编码或解码。

mp3文件的编码格式相对复杂，并且目前流行的有两种版本的mp3文件格式，所以我们不对mp3文件的格式做分析。png文件格式的编码格式相对容易理解，我们以png为例，来说明文件格式的多种多样。

png格式在文件的开头处有8个字节的标志字符，用于表示该文件是png文件。这8个字节对应的整数分别是：137 、80 、78 、71、13、10、26、10，我们知道，137不是一个可显示的字符，之所以把第一个字符设置成这样，是为了防止png文件被误认为是文本文件。不难看出，80、78、71对应的字符正好是PNG。

需要注意的一点是，这8个字节里面出现了13和10，我们说过，这两个字符会因文件的打开方式而有不同，所以在分析png文件的时候，一定要用打开二进制文件的方式来打开。我们可以写一个程序来判断某个文件是否为png格式的文件，png是一种非常常见的图片格式，用截图软件截取到的图片是png格式的，网络上的图片也有很多是png格式，用Windows自带的画图可以做出png图片。这里我们准备了一个a.png的文件，并把它放在了d盘。

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5. char flag[8] = { 137 ,80 ,78 ,71,13,10,26,10 };
6. char buffer[8];
7. FILE \*img = fopen("d:/a.png", "rb");
8. fread(buffer, 1, 8, img);
9. int result = memcmp(buffer, flag, 8);
10. if (result == 0)
11. {
12. printf("该文件是png文件");
13. }
14. else
15. {
16. printf("该文件不是png文件");
17. }
18. fclose(img);
19. getchar();
20. return 0;
21. }
22. FILE \* openFile(char \*route, char \*mode)
23. {
24. FILE \*temp = fopen(route, mode);
25. if (temp == NULL)
26. {
27. printf("文件打开失败\n");
28. getchar();
29. exit(1);
30. }
31. return temp;
32. }

程序9-4-1

程序的某次运行结果如下;

该文件是png文件

程序9-4-1中，我们声明了一个数组flag用来储存标志字符，这些信息当然是预先知道的，在百度百科、维基百科等很多地方都能获取到标志字符的具体组成。接下来我们读取了文件的前8个字节，并把结果存放到了buffer中。然后用memcmp来比较了buffer和flag数组，我们知道，当buffer和flag中的数据完全一样的时候，memcmp会返回0，所以我们可以通过判断memcmp的返回值是否为0来判断目标文件是否为png文件。

如果我们把目标文件改成a.dat或者a.txt，那么得到的结果就一定是不是png文件了（如果我们没有刻意地把png文件的后缀改成dat或者txt的话）。

**试一试**

把程序9-4-1的第7行修改成：FILE \*img = fopen("d:/a.png", "rb");，看看结果会有什么变化。

要想完全解析png文件并把它显示出来需要做相当多的工作，并且涉及到图形界面的操作，所以我们不再具体介绍png的详细编码，但是我们可以编写程序从png文件中提取出各种附加信息，比如它的分辨率。

通常情况下，不管是什么格式的图片，都会在文件中储存图片的分辨率信息，而分辨率储存的位置又会因为图片格式的不同而有不同，有些格式把它们隐藏地非常隐秘，需要花费很大一番力气才能找出来（比如jpg格式），而png则非常简单，它的分辨率信息储存在离文件头偏移16个字节的地方。我们直接定位到该处，然后读取即可。

1. #include <stdio.h>
2. #include <string.h>
3. int tran(const char \*, int);
4. int main()
5. {
6. char flag[8] = { 137 ,80 ,78 ,71,13,10,26,10 };
7. char buffer[8];
8. int w, h;
9. FILE \*img = fopen("d:/a.png", "rb");
10. fread(buffer, 1, 8, img);
11. int result = memcmp(buffer, flag, 8);
12. if (result != 0)
13. {
14. printf("该文件不是png文件");
15. fclose(img);
16. getchar();
17. return 0;
18. }
19. fseek(img, 8, SEEK\_CUR);
20. fread(buffer, 1, 4, img);
21. w = tran(buffer, 4);
22. fread(buffer, 1, 4, img);
23. h = tran(buffer, 4);
24. printf("图片信息：%d×%d", w, h);
25. fclose(img);
26. getchar();
27. return 0;
28. }
29. FILE \* openFile(char \*route, char \*mode)
30. {
31. FILE \*temp = fopen(route, mode);
32. if (temp == NULL)
33. {
34. printf("文件打开失败\n");
35. getchar();
36. exit(1);
37. }
38. return temp;
39. }
40. int tran(const char \*buffer, int Bytes)
41. {
42. if (Bytes < 1 || Bytes>4)
43. {
44. return 0;
45. }
46. int temp = 0;
47. for (int i = Bytes - 1; i >= 0; i--)
48. {
49. \*((char \*)(&temp) + i) = buffer[Bytes - i - 1];
50. }
51. return temp;
52. }

程序9-4-2

程序在某台计算机上的某次运行结果如下：

图片信息：432×234

程序9-4-2稍微有点复杂，但难点仅在tran函数上。首先我们打开文件并判断其是否为png文件，如果不是则直接退出程序，如果是则继续分析。

程序第19行，我们在img的当前位置的基础上向后移动了8个字节，这是因为之前已经读取了8个字节，再移动8个字节就相当于移动到了离文件开头16个字节的地方。

png文件用8个字节来储存图片的宽度和高度（以像素为单位，所以储存的实际上就是分辨率了）。前4个字节是宽度，后4个字节是高度。读者可能会有疑问，既然如此，那么我们直接把前4个字节读取到w中，后4个字节读取到h中，不就可以了吗？为什么还要用一个tran函数来操作一下呢？

原来，这和我们之前的章节中提到过的“小端模式”有关，我们知道，大多数的操作系统都采用的小端模式，所以对于数字1来说，将它储存在int型数据中的时候，实际上是这样的：00000001 00000000 00000000 00000000，而当这个int型数据被写入文件的时候，会被拆分成4个字节来分别写入，写入的4个字节分别是：1、0、0、0，这样，当我们重新从文件中读取出这4个字节的数据，并把它们装入int型数据的时候，该int型数据的值依然是1。而png格式则有点不同，它采用的储存模式是典型的“大端模式”，也就是说，如果要表示宽度为1，则文件中的实际数据会是：0、0、0、1，这样，当我们直接把这4个字节的数据放入int型数据的时候，该数据就会变成：00000000 00000000 00000000 00000001，显然对于小端模式的操作系统来讲，这个数就不再是1了。

为了解决这个问题，我们需要对原始数据加以转换。即把第1个字节的数据放到int型数据的第4个字节里，第2个字节的数据放到int型数据的第3个字节里，以此类推。这就有了tran函数。

tran函数中，有两个参数，一个是const char \*，一个是int，事实上，我们完全可以只设置第1个参数的，但是处于通用性的考虑，我们加入了第2个参数，用来表示需要转换的字节数，这样不仅能转换int型数据，还能转换short。注意到第1个参数的类型是const char \*而不是char \*，这样设置有两个考虑，一方面是，当别人需要调用tran函数的时候，看到const就知道tran不会对buffer中的数据做任何修改，另一方面，这让tran能够同时操作char \*类型的数据和const char \*类型的数据（如果参数是char \*，那么就不能用const char \*做实参了）。

接下来便是转换的核心过程，首先是需要转换的字节数，程序9-4-2中当然是4个字节了。我们判断字节数是否合法，不合法则直接返回0。如果合法，则执行转换操作。我们声明了一个int型变量temp来储存转换后的结果，把它初始化为0是非常有必要的。循环变量i从Bytes-1开始依次递减，最小取值为0，这样的设置让我们联想到数组，事实上我们操作的也确实是数组。

&temp可以取到temp的地址，这个地址的类型是int \*，所以我们需要把它转换成char \*，这样我们就能依次操作temp的4个字节了。我们不妨认为(char \*)(&temp)的结果是一个指向char型数据的指针p，那么\*((char \*)(&temp) + i)就相当于是\*(p+i)了，这样相当于把temp拆成了4个char型数据。不难看出，当i为3的时候，Bytes-i-1的值为0，此时，我们相当于把buffer中第1个字节的数据赋值给了temp的最后1个字节，当i取其它值的时候也是类似的道理，这样，在经过循环之后，temp中就储存着转换后的值了。

以我们现在的知识储备，用C语言写一个复制文件的程序是相当简单的，但是千万要注意，在新建文件的时候，需要对文件格式有相当的了解。比如我们想要新建一个空的Excel文件（后缀是.xls），那么用这样的语句是不合理的：fopen(“a.xls”,”w”)，通过这种方法新建的.xls文件无法被Excel打开（事实上它根本不算是.xls文件）。所以，如果我们编写的某个程序需要新建.xls文件，那么可行的方法有两种，一种是研究.xls文件的具体格式，接着用fopen(“a.xls”,”w”)的方式新建一个”.xls”文件，最后把.xls文件的格式信息写入文件中。另一种方式则更简单，即提前准备好一个空的.xls文件，每次新建的时候就把这个文件复制一份，显然这样会让我们省事不少，至少节省了学习成本。

练习题

1.编写程序，该程序在启动的时候检查count.dat是否存在（.dat文件是二进制文件，其格式为：文件最前面3个字符分别为;’D’、’A’、’T’，以此作为文件标志，紧接着有4个字节的数据用来储存一个int型数据，最后有16个字节用来储存一个SYSTEMTIME结构体，int型数据中储存着程序已经运行的总次数，SYSTEMTIME则储存上次运行的时间），若count.dat存在则检查格式是否正确，若正确则读取int型数据和SYSTEMTIME结构体。若count.dat不存在或格式错误，则新建一个count.dat文件。最后无论结果如何，都显示相关数据，并将文件中的数据更新（包括int型数据和SYSTEMTIME结构体）。

提示：SYSTEMTIME结构体包含于windows.h中，它有8个成员，分别是wYear、wMonth、

wDayOfWeek、wDay、wHour、wMinute、wSecond和wMilliseconds，分别代表：年、月、一周中的第几天、日、时、分、秒和毫秒。通过GetLocalTime函数可以获得当前时间，这两行代码就能读取出日期的相关信息：SYSTEMTIME time; GetLocalTime(&time);（在显示相关信息的时候，只显示年、月、日、时、分）。

2.编写程序，要求用户输入两个文件路径，source和dest，若source存在，则将source复制到dest。

3.编写程序，对文本文件进行加密和解密，加密的方式为把文件中的每个字符的ASCII码+1，解密方式则为把已加密的文件中的每个字符的ASCII码-1。