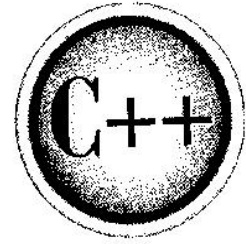The
Complete
Reference

C++

Chapter 9

File I/O

This chapter describes the C file system. As explained in Chapter 8, C++ supports two complete I/O systems: the one inherited from C and the object-oriented system defined by C++. This chapter covers the C file system. (The C++ file system is discussed in Part Two.) While most new code will use the C++ file system, knowledge of the C file system is still important for the reasons given in the preceding chapter.

C Versus C++ File I/O

There is sometimes confusion over how C's file system relates to C++. First, C++ supports the entire Standard C file system. Thus, if you will be porting C code to C++, you will not have to change all of your I/O routines right away. Second, C++ defines its own, object-oriented I/O system, which includes both I/O functions and I/O operators. The C++ I/O system completely duplicates the functionality of the C I/O system and renders the C file system redundant. While you will usually want to use the C++ I/O system, you are free to use the C file system if you like. Of course, most C++ programmers elect to use the C++ I/O system for reasons that are made clear in Part Two of this book.

Streams and Files

Before beginning our discussion of the C file system, it is necessary to know the difference between the terms *streams* and *files*. The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed. That is, the C I/O system provides a level of abstraction between the programmer and the device. This abstraction is called a *stream* and the actual device is called a *file*. It is important to understand how streams and files interact.

> **Note** *The concept of streams and files is also important to the C++ I/O system discussed in Part Two.*

Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. Even though each device is very different, the file system transforms each into a logical device called a stream. All streams behave similarly. Because streams are largely device independent, the same function that can write to a disk file can also be used to write to another type of device, such as the console. There are two types of streams: text and binary.

Text Streams

A *text stream* is a sequence of characters. Standard C allows (but does not require) a text stream to be organized into lines terminated by a newline character. However, the newline character is optional on the last line. (Actually, most C/C++ compilers do not terminate text streams with newline characters.) In a text stream, certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

Binary Streams

A *binary stream* is a sequence of bytes that have a one-to-one correspondence to those in the external device—that is, no character translations occur. Also, the number of bytes written (or read) is the same as the number on the external device. However, an implementation-defined number of null bytes may be appended to a binary stream. These null bytes might be used to pad the information so that it fills a sector on a disk, for example.

Files

In C/C++, a *file* may be anything from a disk file to a terminal or printer. You associate a stream with a specific file by performing an open operation. Once a file is open, information may be exchanged between it and your program.

Not all files have the same capabilities. For example, a disk file can support random access while some printers cannot. This brings up an important point about the C I/O system: All streams are the same but all files are not.

If the file can support *position requests*, opening that file also initializes the *file position indicator* to the start of the file. As each character is read from or written to the file, the position indicator is incremented, ensuring progression through the file.

You disassociate a file from a specific stream with a close operation. If you close a file opened for output, the contents, if any, of its associated stream are written to the external device. This process is generally referred to as *flushing* the stream, and guarantees that no information is accidentally left in the disk buffer. All files are closed automatically when your program terminates normally, either by main( ) returning to the operating system or by a call to **exit( )**. Files are not closed when a program terminates abnormally, such as when it crashes or when it calls **abort( )**.

Each stream that is associated with a file has a file control structure of type **FILE**. Never modify this file control block.

If you are new to programming, the separation of streams and files may seem unnecessary or contrived. Just remember that its main purpose is to provide

a consistent interface. You need only think in terms of streams and use only one file system to accomplish all I/O operations. The I/O system automatically converts the raw input or output from each device into an easily managed stream.

File System Basics

The C file system is composed of several interrelated functions. The most common of these are shown in Table 9-1. They require the header **stdio.h**. C++ programs may also use the C++-style header **<cstdio>**.

Name	Function
fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as **putc( )**.
getc( )	Reads a character from a file.
fgetc( )	Same as **getc( )**.
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file what **printf( )** is to the console.
fscanf( )	Is to a file what **scanf( )** is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

Table 9-1. *Commonly Used C File-System Functions*

The header file **stdio.h** and **<cstdio>** header provide the prototypes for the I/O functions and define these three types: **size_t**, **fpos_t**, and **FILE**. The **size_t** type is some variety of unsigned integer, as is **fpos_t**. The **FILE** type is discussed in the next section.

Also defined in **stdio.h** and **<cstdio>** are several macros. The ones relevant to this chapter are **NULL, EOF, FOPEN_MAX, SEEK_SET, SEEK_CUR,** and **SEEK_END**. The **NULL** macro defines a null pointer. The **EOF** macro is generally defined as −1 and is the value returned when an input function tries to read past the end of the file. **FOPEN_MAX** defines an integer value that determines the number of files that may be open at any one time. The other macros are used with **fseek( )**, which is the function that performs random access on a file.

The File Pointer

The file pointer is the common thread that unites the C I/O system. A *file pointer* is a pointer to a structure of type **FILE**. It points to information that defines various things about the file, including its name, status, and the current position of the file. In essence, the file pointer identifies a specific file and is used by the associated stream to direct the operation of the I/O functions. In order to read or write files, your program needs to use file pointers. To obtain a file pointer variable, use a statement like this:

```
FILE *fp;
```

Opening a File

The **fopen( )** function opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file. Most often (and for the rest of this discussion), the file is a disk file. The **fopen( )** function has this prototype:

FILE *fopen(const char *filename*, const char *mode*);

where *filename* is a pointer to a string of characters that make up a valid filename and may include a path specification. The string pointed to by *mode* determines how the file will be opened. Table 9-2 shows the legal values for *mode*. Strings like "r+b" may also be represented as "rb+."

Mode	Meaning
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.

Table 9-2. *The Legal Values for Mode*

Mode	Meaning
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

Table 9-2. *The Legal Values for Mode* (continued)

As stated, the **fopen( )** function returns a file pointer. Your program should never alter the value of this pointer. If an error occurs when it is trying to open the file, **fopen( )** returns a null pointer.

The following code uses **fopen( )** to open a file named TEST for output.

```
FILE *fp;
fp = fopen("test", "w");
```

While technically correct, you will usually see the preceding code written like this:

```
FILE *fp;

if ((fp = fopen("test","w"))==NULL) {
  printf("Cannot open file.\n");
  exit(1);
}
```

This method will detect any error in opening a file, such as a write-protected or a full disk, before your program attempts to write to it. In general, you will always want to confirm that **fopen( )** succeeded before attempting any other operations on the file.

Although most of the file modes are self-explanatory, a few comments are in order. If, when opening a file for read-only operations, the file does not exist, **fopen( )** will fail. When opening a file using append mode, if the file does not exist, it will be created. Further, when a file is opened for append, all new data written to the file will be written to the end of the file. The original contents will remain unchanged. If, when a file is opened for writing, the file does not exist, it will be created. If it does exist, the contents of the original file will be destroyed and a new file created. The difference between modes r+ and w+ is that r+ will not create a file if it does not exist; however, w+ will. Further, if the file already exists, opening it with w+ destroys its contents; opening it with r+ does not.

As Table 9-2 shows, a file may be opened in either text or binary mode. In most implementations, in text mode, carriage return/linefeed sequences are translated to newline characters on input. On output, the reverse occurs: newlines are translated to carriage return/linefeeds. No such translations occur on binary files.

The number of files that may be open at any one time is specified by **FOPEN_MAX**. This value will usually be at least 8, but you must check your compiler's documentation for its exact value.

Closing a File

The **fclose( )** function closes a stream that was opened by a call to **fopen( )**. It writes any data still remaining in the disk buffer to the file and does a formal operating-system-level close on the file. Failure to close a stream invites all kinds of trouble, including lost data, destroyed files, and possible intermittent errors in your program. **fclose( )** also frees the file control block associated with the stream, making it available for reuse. There is an operating-system limit to the number of open files you may have at any one time, so you may have to close one file before opening another.

The **fclose( )** function has this prototype:

int fclose(FILE *fp);

where *fp* is the file pointer returned by the call to **fopen( )**. A return value of zero signifies a successful close operation. The function returns EOF if an error occurs. You can use the standard function **ferror( )** (discussed shortly) to determine and report any problems. Generally, **fclose( )** will fail only when a disk has been prematurely removed from the drive or there is no more space on the disk.

Writing a Character

The C I/O system defines two equivalent functions that output a character: **putc( )** and **fputc( )**. (Actually, **putc( )** is usually implemented as a macro.) There are two identical functions simply to preserve compatibility with older versions of C. This book uses **putc( )**, but you can use **fputc( )** if you like.

The putc( ) function writes characters to a file that was previously opened for writing using the fopen( ) function. The prototype of this function is

int putc(int *ch*, FILE **fp*);

where *fp* is the file pointer returned by fopen( ) and *ch* is the character to be output. The file pointer tells putc( ) which file to write to. Although *ch* is defined as an int, only the low-order byte is written.

If a putc( ) operation is successful, it returns the character written. Otherwise, it returns EOF.

Reading a Character

There are also two equivalent functions that input a character: getc( ) and fgetc( ). Both are defined to preserve compatibility with older versions of C. This book uses getc( ) (which is usually implemented as a macro), but you can use fgetc( ) if you like.

The getc( ) function reads characters from a file opened in read mode by fopen( ). The prototype of getc( ) is

int getc(FILE **fp*);

where *fp* is a file pointer of type FILE returned by fopen( ). getc( ) returns an integer, but the character is contained in the low-order byte. Unless an error occurs, the high-order byte is zero.

The getc( ) function returns an EOF when the end of the file has been reached. Therefore, to read to the end of a text file, you could use the following code:

```
do {
  ch = getc(fp);
} while(ch!=EOF);
```

However, getc( ) also returns EOF if an error occurs. You can use ferror( ) to determine precisely what has occurred.

Using fopen( ), getc( ), putc( ), and fclose( )

The functions fopen( ), getc( ), putc( ), and fclose( ) constitute the minimal set of file routines. The following program, KTOD, is a simple example of using putc( ), fopen( ), and fclose( ). It reads characters from the keyboard and writes them to a disk file until the user types a dollar sign. The filename is specified from the command line. For example, if you call this program KTOD, typing **KTOD TEST** allows you to enter lines of text into the file called TEST.

```
/* KTOD: A key to disk program. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("You forgot to enter the filename.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  do {
    ch = getchar();
    putc(ch, fp);
  } while (ch != '$');

  fclose(fp);

  return 0;
}
```

The complementary program DTOS reads any text file and displays the contents on the screen. It demonstrates getc().

```
/* DTOS: A program that reads files and displays them
         on the screen. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;
```

```
    if(argc!=2) {
      printf("You forgot to enter the filename.\n");
      exit(1);
    }

    if((fp=fopen(argv[1], "r"))==NULL) {
      printf("Cannot open file.\n");
      exit(1);
    }

    ch = getc(fp);    /* read one character */

    while (ch!=EOF) {
      putchar(ch);   /* print on screen */
      ch = getc(fp);
    }

    fclose(fp);

    return 0;
  }
```

To try these two programs, first use KTOD to create a text file. Then read its contents using DTOS.

Using feof( )

As just described, getc( ) returns EOF when the end of the file has been encountered. However, testing the value returned by getc( ) may not be the best way to determine when you have arrived at the end of a file. First, the file system can operate on both text and binary files. When a file is opened for binary input, an integer value that will test equal to EOF may be read. This would cause the input routine to indicate an end-of-file condition even though the physical end of the file had not been reached. Second, getc( ) returns EOF when it fails and when it reaches the end of the file. Using only the return value of getc( ), it is impossible to know which occurred. To solve these problems, the C file system includes the function feof( ), which determines when the end of the file has been encountered. The feof( ) function has this prototype:

int feof(FILE *fp);

feof( ) returns true if the end of the file has been reached; otherwise, it returns 0. Therefore, the following routine reads a binary file until the end of the file is encountered:

```
while(!feof(fp)) ch = getc(fp);
```

Of course, you can apply this method to text files as well as binary files.

The following program, which copies text or binary files, contains an example of **feof( )**. The files are opened in binary mode and **feof( )** checks for the end of the file.

```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("You forgot to enter a filename.\n");
    exit(1);
  }

  if((in=fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }
  if((out=fopen(argv[2], "wb")) == NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* This code actually copies the file. */
  while(!feof(in)) {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
  }

  fclose(in);
  fclose(out);

  return 0;
}
```

Working with Strings: fputs( ) and fgets( )

In addition to getc( ) and putc( ), the C file system supports the related functions **fgets( )** and **fputs( )**, which read and write character strings from and to a disk file. These functions work just like **putc( )** and **getc( )**, but instead of reading or writing a single character, they read or write strings. They have the following prototypes:

int fputs(const char *str, FILE *fp);
char *fgets(char *str, int length, FILE *fp);

The **fputs( )** function writes the string pointed to by *str* to the specified stream. It returns EOF if an error occurs.

The **fgets( )** function reads a string from the specified stream until either a newline character is read or *length* −1 characters have been read. If a newline is read, it will be part of the string (unlike the **gets( )** function). The resultant string will be null terminated. The function returns *str* if successful and a null pointer if an error occurs.

The following program demonstrates **fputs( )**. It reads strings from the keyboard and writes them to the file called TEST. To terminate the program, enter a blank line. Since **gets( )** does not store the newline character, one is added before each string is written to the file so that the file can be read more easily.

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

int main(void)
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w"))==NULL) {
    printf('Cannot open file.\n");
    exit(1);
  }

  do {
    printf("Enter a string (CR to quit):\n");
    gets(str);
    strcat(str, "\n");   /* add a newline */
    fputs(str, fp);
  } while(*str!='\n');

  return 0;
}
```

rewind( )

The **rewind( )** function resets the file position indicator to the beginning of the file specified as its argument. That is, it "rewinds" the file. Its prototype is

void rewind(FILE *fp*);

where *fp* is a valid file pointer.

To see an example of **rewind( )**, you can modify the program from the previous section so that it displays the contents of the file just created. To accomplish this, the program rewinds the file after input is complete and then uses **fgets( )** to read back the file. Notice that the file must now be opened in read/write mode using "w+" for the mode parameter.

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

int main(void)
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w+"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  do {
    printf("Enter a string (CR to quit):\n");
    gets(str);
    strcat(str, "\n");  /* add a newline */
    fputs(str, fp);
  } while(*str!='\n');

  /* now, read and display the file */
  rewind(fp);  /* reset file position indicator to
                  start of the file. */
  while(!feof(fp)) {
    fgets(str, 79, fp);
    printf(str);
  }
```

```
        return 0;
    }
```

ferror( )

The **ferror( )** function determines whether a file operation has produced an error. The ferror( ) function has this prototype:

    int ferror(FILE *fp);

where *fp* is a valid file pointer. It returns true if an error has occurred during the last file operation; otherwise, it returns false. Because each file operation sets the error condition, **ferror( )** should be called immediately after each file operation; otherwise, an error may be lost.

The following program illustrates **ferror( )** by removing tabs from a file and substituting the appropriate number of spaces. The tab size is defined by TAB_SIZE. Notice how **ferror( )** is called after each file operation. To use the program, specify the names of the input and output files on the command line.

```
/* The program substitutes spaces for tabs
   in a text file and supplies error checking. */

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

#define TAB_SIZE 8
#define IN 0
#define OUT 1

void err(int e);

int main(int argc, char *argv[])
{
  FILE *in, *out;
  int tab, i;
  char ch;

  if(argc!=3) {
    printf("usage: detab <in> <out>\n");
    exit(1);
  }

  if((in = fopen(argv[1], "rb"))==NULL) {
```

```
    printf("Cannot open %s.\n", argv[1]);
    exit(1);
}

if((out = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open %s.\n", argv[1]);
    exit(1);
}

tab = 0;
do {
    ch = getc(in);
    if(ferror(in)) err(IN);

    /* if tab found, output appropriate number of spaces */
    if(ch=='\t') {
        for(i=tab; i<8; i++) {
            putc(' ', out);
            if(ferror(out)) err(OUT);
        }
        tab = 0;
    }
    else {
        putc(ch, out);
        if(ferror(out)) err(OUT);
        tab++;
        if(tab==TAB_SIZE) tab = 0;
        if(ch=='\n' || ch=='\r') tab = 0;
    }
} while(!feof(in));
fclose(in);
fclose(out);

return 0;
}

void err(int e)
{
    if(e==IN) printf("Error on input.\n");
    else printf("Error on output.\n");
    exit(1);
}
```

Erasing Files

The **remove( )** function erases the specified file. Its prototype is

int remove(const char *filename);

It returns zero if successful; otherwise, it returns a nonzero value.

The following program erases the file specified on the command line. However, it first gives you a chance to change your mind. A utility like this might be useful to new computer users.

```
/* Double check before erasing. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  char str[80];

  if(argc!=2) {
    printf("usage: xerase <filename>\r");
    exit(1);
  }

  printf("Erase %s? (Y/N): ", argv[1]);
  gets(str);

  if(toupper(*str)=='Y')
    if(remove(argv[1])) {
      printf("Cannot erase file.\n");
      exit(1);
    }
  return 0;
}
```

Flushing a Stream

If you wish to flush the contents of an output stream, use the **fflush( )** function, whose prototype is shown here:

int fflush(FILE *fp);

This function writes the contents of any buffered data to the file associated with *fp*. If you call **fflush( )** with *fp* being null, all files opened for output are flushed. The **fflush( )** function returns 0 if successful; otherwise, it returns EOF.

fread( ) and fwrite( )

To read and write data types that are longer than one byte, the C file system provides two functions: **fread( )** and **fwrite( )**. These functions allow the reading and writing of blocks of any type of data. Their prototypes are

size_t fread(void *buffer*, size_t *num_bytes*, size_t *count*, FILE *fp*);
size_t fwrite(const void *buffer*, size_t *num_bytes*, size_t *count*, FILE *fp*);

For **fread( )**, *buffer* is a pointer to a region of memory that will receive the data from the file. For **fwrite( )**, *buffer* is a pointer to the information that will be written to the file. The value of *count* determines how many items are read or written, with each item being *num_bytes* bytes in length. (Remember, the type **size_t** is defined as some type of unsigned integer.) Finally, *fp* is a file pointer to a previously opened stream.

The **fread( )** function returns the number of items read. This value may be less than *count* if the end of the file is reached or an error occurs. The **fwrite( )** function returns the number of items written. This value will equal *count* unless an error occurs.

Using fread( ) and fwrite( )

As long as the file has been opened for binary data, **fread( )** and **fwrite( )** can read and write any type of information. For example, the following program writes and then reads back a **double**, an **int**, and a **long** to and from a disk file. Notice how it uses sizeof to determine the length of each data type.

```c
/* Write some non-character data to a disk file
   and read it back.  */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fp;
   double d = 12.23;
   int i = 101;
   long l = 123023L;
```

```
   if((fp=fopen("test", "wb+"))==NULL) {
     printf("Cannot open file.\n");
     exit(1);
   }

   fwrite(&d, sizeof(double), 1, fp);
   fwrite(&i, sizeof(int), 1, fp);
   fwrite(&l, sizeof(long), 1, fp);

   rewind(fp);

   fread(&d, sizeof(double), 1, fp);
   fread(&i, sizeof(int), 1, fp);
   fread(&l, sizeof(long), 1, fp);

   printf("%f %d %ld", d, i, l);

   fclose(fp);

   return 0;
}
```

As this program illustrates, the buffer can be (and often is) merely the memory used to hold a variable. In this simple program, the return values of **fread( )** and **fwrite( )** are ignored. In the real world, however, you should check their return values for errors.

One of the most useful applications of **fread( )** and **fwrite( )** involves reading and writing user-defined data types, especially structures. For example, given this structure:

```
struct struct_type {
  float balance;
  char name[80];
} cust;
```

the following statement writes the contents of **cust** to the file pointed to by **fp**.

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

fseek( ) and Random-Access I/O

You can perform random-access read and write operations using the C I/O system with the help of fseek( ), which sets the file position indicator. Its prototype is shown here:

int fseek(FILE *fp, long int *numbytes*, int *origin*);

Here, *fp* is a file pointer returned by a call to fopen( ). *numbytes* is the number of bytes from *origin* that will become the new current position, and *origin* is one of the following macros:

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

Therefore, to seek *numbytes* from the start of the file, *origin* should be **SEEK_SET**. To seek from the current position, use **SEEK_CUR**; and to seek from the end of the file, use **SEEK_END**. The **fseek( )** function returns 0 when successful and a nonzero value if an error occurs.

The following program illustrates **fseek( )**. It seeks to and displays the specified byte in the specified file. Specify the filename and then the byte to seek to on the command line.

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

int main(int argc, char *argv[])
{
  FILE *fp;

  if(argc!=3) {
    printf("Usage: SEEK filename byte\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open file.\n");
```

```
    exit(1);
  }

  if(!seek(fp, atol(argv[2]), SEEK_SET)) {
    printf("Seek error.\n");
    exit(1);
  }

  printf("Byte at %ld is %c.\n", atol(argv[2]), getc(fp));
  fclose(fp);

  return 0;
}
```

You can use **fseek( )** to seek in multiples of any type of data by simply multiplying the size of the data by the number of the item you want to reach. For example, assume that you have a mailing list that consists of structures of type **list_type**. To seek to the tenth address in the file that holds the addresses, use this statement:

```
fseek(fp, 9*sizeof(struct list_type), SEEK_SET);
```

You can determine the current location of a file using **ftell( )**. Its prototype is

long int ftell(FILE *fp);

It returns the location of the current position of the file associated with *fp*. If a failure occurs, it returns −1.

In general, you will want to use random access only on binary files. The reason for this is simple. Because text files may have character translations performed on them, there may not be a direct correspondence between what is in the file and the byte to which it would appear that you want to seek. The only time you should use **fseek( )** with a text file is when seeking to a position previously determined by **ftell( )**, using **SEEK_SET** as the origin.

Remember one important point: Even a file that contains only text can be opened as a binary file, if you like. There is no inherent restriction about random access on files containing text. The restriction applies only to files opened as text files.

fprintf( ) and fscanf( )

In addition to the basic I/O functions already discussed, the C I/O system includes **fprintf( )** and **fscanf( )**. These functions behave exactly like **printf( )** and **scanf( )** except that they operate with files. The prototypes of **fprintf( )** and **fscanf( )** are

```
int fprintf(FILE *fp, const char *control_string,...);
int fscanf(FILE *fp, const char *control_string,...);
```

where *fp* is a file pointer returned by a call to **fopen( )**. **fprintf( )** and **fscanf( )** direct their I/O operations to the file pointed to by *fp*.

As an example, the following program reads a string and an integer from the keyboard and writes them to a disk file called TEST. The program then reads the file and displays the information on the screen. After running this program, examine the TEST file. As you will see, it contains human-readable text.

```c
/* fscanf() - fprintf() example */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  char s[80];
  int t;

  if((fp=fopen("test", "w")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  printf("Enter a string and a number: ");
  fscanf(stdin, "%s%d", s, &t); /* read from keyboard */

  fprintf(fp, "%s %d", s, t); /* write to file */
  fclose(fp);

  if((fp=fopen("test","r")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  fscanf(fp, "%s%d", s, &t); /* read from file */
  fprintf(stdout, "%s %d", s, t); /* print on screen */

  return 0;
}
```

A word of warning: Although **fprintf( )** and **fscanf( )** often are the easiest way to write and read assorted data to disk files, they are not always the most efficient. Because formatted ASCII data is being written as it would appear on the screen (instead of in binary), extra overhead is incurred with each call. So, if speed or file size is a concern, you should probably use **fread( )** and **fwrite( )**.

The Standard Streams

As it relates to the C file system, when a program starts execution, three streams are opened automatically. They are **stdin** (standard input), **stdout** (standard output), and **stderr** (standard error). Normally, these streams refer to the console, but they may be redirected by the operating system to some other device in environments that support redirectable I/O. (Redirectable I/O is supported by Windows, DOS, Unix, and OS/2, for example.)

Because the standard streams are file pointers, they may be used by the C I/O system to perform I/O operations on the console. For example, **putchar( )** could be defined like this:

```
int putchar(char c)
{
    return putc(c, stdout);
}
```

In general, **stdin** is used to read from the console, and **stdout** and **stderr** are used to write to the console.

You may use **stdin**, **stdout**, and **stderr** as file pointers in any function that uses a variable of type **FILE ***. For example, you could use **fgets( )** to input a string from the console using a call like this:

```
char str[255];
fgets(str, 80, stdin);
```

In fact, using **fgets( )** in this manner can be quite useful. As mentioned earlier in this book, when using **gets( )** it is possible to overrun the array that is being used to receive the characters entered by the user because **gets( )** provides no bounds checking. When used with **stdin**, the **fgets( )** function offers a useful alternative because it can limit the number of characters read and thus prevent array overruns. The only trouble is that **fgets( )** does not remove the newline character and **gets( )** does, so you will have to manually remove it, as shown in the following program.

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

int main(void)
{
  char str[80];
  int i;

  printf("Enter a string: ");
  fgets(str, 10, stdin);

  /* remove newline, if present */
  i = strlen(str)-1;
  if(str[i]=='\n') str[i] = '\0';

  printf("This is your string: %s", str);

  return 0;
}
```

Keep in mind that **stdin**, **stdout**, and **stderr** are not variables in the normal sense and may not be assigned a value using **fopen( )**. Also, just as these file pointers are created automatically at the start of your program, they are closed automatically at the end; you should not try to close them.

The Console I/O Connection

There is actually little distinction between console I/O and file I/O. The console I/O functions described in Chapter 8 actually direct their I/O operations to either **stdin** or **stdout**. In essence, the console I/O functions are simply special versions of their parallel file functions. The reason they exist is as a convenience to you, the programmer.

As described in the previous section, you can perform console I/O using any of the file system functions. However, what might surprise you is that you can perform disk file I/O using console I/O functions, such as **printf( )**! This is because all of the console I/O functions operate on **stdin** and **stdout**. In environments that allow redirection of I/O, this means that **stdin** and **stdout** could refer to a device other than the keyboard and screen. For example, consider this program:

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

```
int main(void)
{
  char str[80];

  printf("Enter a string: ");
  gets(str);
  printf(str);

  return 0;
}
```

Assume that this program is called TEST. If you execute TEST normally, it displays its prompt on the screen, reads a string from the keyboard, and displays that string on the display. However, in an environment that supports I/O redirection, either **stdin**, **stdout**, or both could be redirected to a file. For example, in a DOS or Windows environment, executing TEST like this:

```
TEST > OUTPUT
```

causes the output of TEST to be written to a file called OUTPUT. Executing TEST like this:

```
TEST < INPUT > OUTPUT
```

directs **stdin** to the file called INPUT and sends output to the file called OUTPUT. When a program terminates, any redirected streams are reset to their default status.

Using freopen( ) to Redirect the Standard Streams

You can redirect the standard streams by using the **freopen( )** function. This function associates an existing stream with a new file. Thus, you can use it to associate a standard stream with a new file. Its prototype is

FILE *freopen(const char *filename, const char *mode, FILE *stream);

where *filename* is a pointer to the filename you wish associated with the stream pointed to by *stream*. The file is opened using the value of *mode*, which may have the same values as those used with **fopen( )**. **freopen( )** returns *stream* if successful or **NULL** on failure.

The following program uses **freopen( )** to redirect **stdout** to a file called OUTPUT:

```
#include <stdio.h>

int main(void)
{
  char str[80];

  freopen("OUTPUT", "w", stdout);

  printf("Enter a string: ");
  gets(str);
  printf(str);

  return 0;
}
```
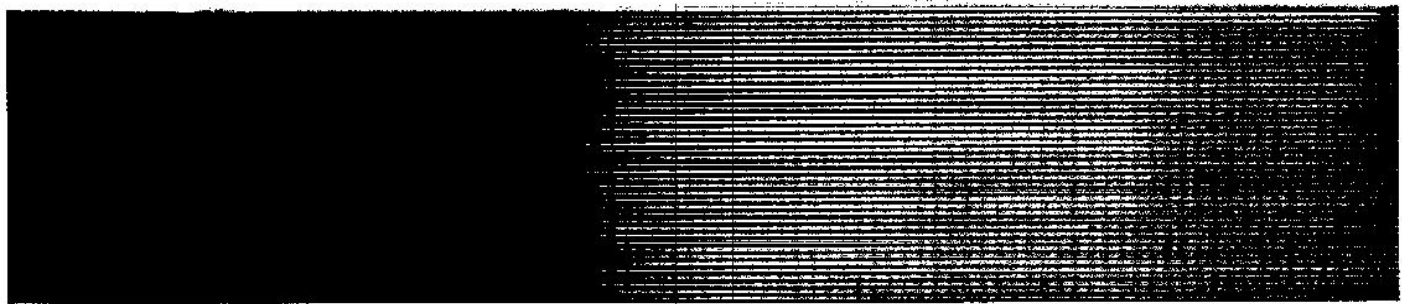
ays its
the
dout,
nt,

In general, redirecting the standard streams by using **freopen( )** is useful in special situations, such as debugging. However, performing disk I/O using redirected **stdin** and **stdout** is not as efficient as using functions like **fread( )** or **fwrite( )**.

atus.

on
lard