13 File Input/Output

You will learn about the following in this chapter:

- Functions:

  `fopen()`, `getc()`, `putc()`, `exit()`, `fclose()`

  `fprintf()`, `fscanf()`, `fgets()`, `fputs()`

  `rewind()`, `fseek()`, `ftell()`, `fflush()`

  `fgetpos()`, `fsetpos()`, `feof()`, `ferror()`

  `ungetc()`, `setvbuf()`, `fread()`, `fwrite()`
- How to process files using C's standard I/O family of functions
- Text modes and binary modes, text and binary formats, and buffered and nonbuffered I/O
- Using functions that can access files both sequentially and randomly

Files are essential to today's computer systems. They are used to store programs, documents, data, correspondence, forms, graphics, photos, music, videos, and myriad other kinds of information. As a programmer, you will have to write programs that create files, write into files, and read from files. In this chapter, we show you how.

Communicating with Files

Often you need programs that can read information from files or can write results into a file. One such form of program-file communication is file redirection, as you saw in Chapter 8, "Character Input/Output and Input Validation." This method is simple but limited. For example, suppose you want to write an interactive program that asks you for book titles and then saves the complete listing in a file. If you use redirection, as in

```
books > bklist
```

your interactive prompts are redirected into `bklist`. Not only does this put unwanted text into `bklist`, it prevents you from seeing the questions you are supposed to answer.

C, as you might expect, offers more powerful methods of communicating with files. It enables you to open a file from within a program and then use special I/O functions to read from or write to that file. Before investigating these methods, however, let's briefly review the nature of a file.

What Is a File?

A *file* is a named section of storage, usually on a disk, or, more recently, on a solid-state device. You think of `stdio.h`, for instance, as the name of a file containing some useful information. To the operating system, however, a file is a bit more complicated. A large file, for example, could wind up stored in several scattered fragments, or it might contain additional data that allows the operating system to determine what kind of file it is. However, these are the operating system's concerns, not yours (unless you are writing operating systems). Your concern is how files appear to a C program.

C views a file as a continuous sequence of bytes, each of which can be read individually. This corresponds to the file structure in the Unix environment, where C grew up. Because other environments may not correspond exactly to this model, C provides two ways to view files: the text view and the binary view.

The Text Mode and the Binary Mode

First, let's distinguish between text and binary content, text and binary file formats, and text and binary modes for files.

All file content is in binary form (zeros and ones). But if a file primarily uses the binary codes for characters (for instance, ASCII or Unicode) to represent text, much as a C string does, then it is a text file; it has text content. If, instead, the binary values in the file represent machine-language code or numeric data (using the same internal representation as, say, used for `long` or `double` values) or image or music encoding, the content is binary.

Unix uses the same file format for both kinds of content. Not surprisingly, given that C was created as tool for developing Unix, both C and Unix use `\n` (the line-feed character) to indicate a line break in text. Unix directories maintain a file-size count that programs can use to determine when end-of-file is reached. However, other systems have had other ways of handling files specifically intended to hold text. That is, they have a format for text files different from the Unix model. For example, pre-OS X Macintosh files used `\r` (the carriage-return character) to indicate a new line. Early MS-DOS files used the combination `\r\n` to indicate a newline and an imbedded Ctrl+Z character to denote end-of-file, even though the actual file would be padded with additional null characters to make the total size a multiple of 256. (In Windows, Notepad still produces MS-DOS format text files, but newer editors may use a more Unix-like format.) Other systems might make every line in a text file of the same length, padding each line with null characters, if necessary, to make the length come out right. Or a system might encode the length of each line at the beginning of each line.

To bring some regularity to the handling of text files, C provides two ways of accessing a file: *binary* mode and *text* mode. In the binary mode, each and every byte of the file is accessible to a program. In the text mode, however, what the program sees can differ from what is in the file. With the text view, the local environment's representation of such things as the end of a line or end-of-file are mapped to the C view when a file is read. Similarly, the C view is mapped to the local representation of output. For example, a C program compiled on an older Macintosh and using text mode would convert `\r` to `\n` when reading a file in text mode and convert `\n` to `\r` when writing to a file. Or a C text-mode program compiled on an MS-DOS platform would convert `\r\n` to `\n` when reading from a file and convert `\n` to `\r\n` when writing to a file. Text-mode programs written for other environments make similar adjustments.

You aren't restricted to using only the text view for a text file. You can also use the binary view of the same file. If you do for an old MS-DOS text file, your program sees both the `\r` and the `\n` characters in the file; no mapping takes place. (Figure 13.1 illustrates this with some nautical text.) If you want to write a text-viewing program that works for, say, old Mac formats, MS-DOS formats, and Unix/Linux formats, you would use binary mode so that the program could determine the actual file contents and act accordingly.

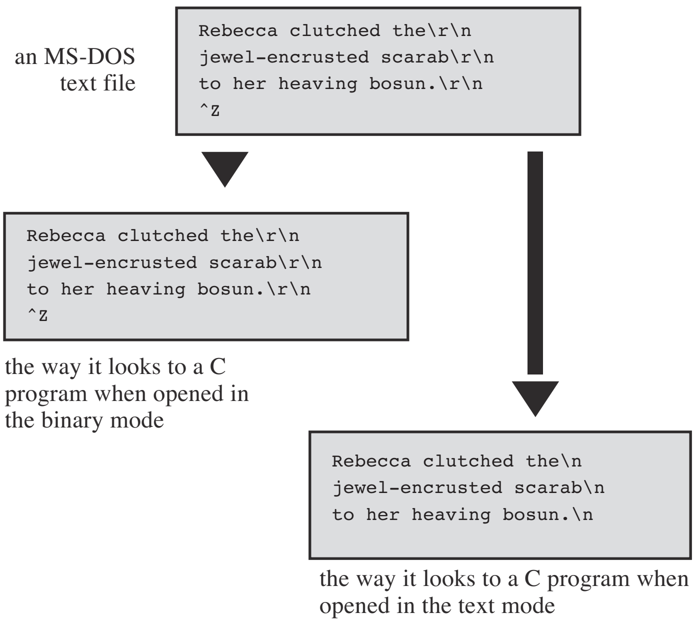
an MS-DOS text file
Rebecca clutched the\r\n
jewel-encrusted scarab\r\n
to her heaving bosun.\r\n
^Z
Rebecca clutched the\r\n
jewel-encrusted scarab\r\n
to her heaving bosun.\r\n
^Z
the way it looks to a C program when opened in the binary mode
Rebecca clutched the\n
jewel-encrusted scarab\n
to her heaving bosun.\n
the way it looks to a C program when opened in the text mode

Figure 13.1 Binary view and text view.

Although C provides for both a binary view and a text view, these views can be implemented identically. As mentioned, because Unix uses just one file structure, both views are the same for Unix implementations. And this is true for Linux, too.

Levels of I/O

In addition to selecting the view of a file, you can, in most cases, choose between two levels of I/O (that is, between two levels of handling access to files). *Low-level I/O* uses the fundamental I/O services provided by the operating system. *Standard high-level I/O* uses a standard package of C library functions and `stdio.h` header file definitions. The C standard supports only the standard I/O package because there is no way to guarantee that all operating systems can be represented by the same low-level I/O model. Particular implementations may also provide low-level libraries, but, because the C standard establishes a portable I/O model, we will concentrate on it.

Standard Files

C programs automatically open three files on your behalf. They are termed the *standard input*, the *standard output*, and the *standard error output*. The standard input, by default, is the normal input device for your system, usually your keyboard. Both the standard output and the standard error output, by default, are the normal output device for your system, usually your display screen.

The standard input, naturally, provides input to your program. It's the file that is read by `getchar()` and `scanf()`. The standard output is where normal program output goes. It is used by `putchar()`, `puts()`, and `printf()`. Redirection, as you learned in Chapter 8, causes other files to be recognized as the standard input or standard output. The purpose of the standard error output file is to provide a logically distinct place to send error messages. If, for example, you use redirection to send output to a file instead of to the screen, output sent to the standard error output still goes to the screen. This is good because if the error messages were routed to the file, you would not see them until you viewed the file.

Standard I/O

The standard I/O package has two advantages, besides portability, over low-level I/O. First, it has many specialized functions that simplify handling different I/O problems. For example, `printf()` converts various forms of data to string output suitable for terminals. Second, input and output are *buffered*. That is, information is transferred in large chunks (typically 512 bytes at a time or more) instead of a byte at a time. When a program reads a file, for example, a chunk of data is copied to a buffer—an intermediate storage area. This buffering greatly increases the data transfer rate. The program can then examine individual bytes in the buffer. The buffering is handled behind the scenes, so you have the illusion of character-by-character access. (You can also buffer low-level I/O, but you have to do much of the work yourself.) Listing 13.1 shows how to use standard I/O to read a file and count the number of characters in the file. We'll discuss the features of Listing 13.1 in the next several sections. (This program uses command-line arguments. If you're a Windows user, you might have to run the program in a command-prompt window after compiling. If you're a Macintosh user, the simplest approach is to compile and run the program in command-line form using Terminal. Or, as described in Chapter 11, "Character Strings and String Functions," you can use the Xcode Product menu to provide command-line arguments for a program run in the IDE. Alternatively, you can alter the program to use `puts()` and `fgets()` instead of command-line arguments to get the filename.)

Listing 13.1 **The `count.c` Program**

```c
/* count.c -- using standard I/O */
#include <stdio.h>
#include <stdlib.h> // exit() prototype

int main(int argc, char *argv[])
{
    int ch;         // place to store each character as read
    FILE *fp;       // "file pointer"
    unsigned long count = 0;
    if (argc != 2)
    {
        printf("Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fp = fopen(argv[1], "r")) == NULL)
    {
        printf("Can't open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    while ((ch = getc(fp)) != EOF)
    {
        putc(ch,stdout);  // same as putchar(ch);
        count++;
    }
    fclose(fp);
    printf("File %s has %lu characters\n", argv[1], count);

    return 0;
}
```

Checking for Command-Line Arguments

First, the program in Listing 13.1 checks the value of `argc` to see if there is a command-line argument. If there isn't, the program prints a usage message and exits. The string `argv[0]` is the name of the program. Using `argv[0]` instead of the program name explicitly causes the error message to change automatically if you change the name of the executable file. This feature is also handy in environments such as Unix that permit multiple names for a single file. But beware—some operating systems may not recognize `argv[0]`, so this usage is not completely portable.

The `exit()` function causes the program to terminate, closing any open files. The argument to `exit()` is passed on to some operating systems, including Unix, Linux, Windows, and MS-DOS, where it can be used by other programs. The usual convention is to pass a value of `0` for programs that terminate normally and to pass nonzero values for abnormal termination. Different exit values can be used to distinguish between different causes of failure, and this is the usual practice in Unix and DOS programming. However, not all operating systems recognize the same range of possible return values. Therefore, the C standard mandates a rather restricted minimum range. In particular, the standard requires that the value `0` or the macro `EXIT_SUCCESS` be used to indicate successful termination, and the macro `EXIT_FAILURE` be used to indicate unsuccessful termination. These macros, along with the `exit()` prototype, are found in the `stdlib.h` header file.

Under ANSI C, using `return` in the initial call to `main()` has the same effect as calling `exit()`. Therefore, in `main()`, the statement

```
return 0;
```

which you've been using all along, is equivalent in effect to this statement:

```
exit(0);
```

Note, however, the qualifying phrase "the initial call." If you make `main()` into a recursive program, `exit()` still terminates the program, but `return` passes control to the previous level of recursion until the original level is reached. Then `return` terminates the program. Another difference between `return` and `exit()` is that `exit()` terminates the program even if called in a function other than `main()`.

The `fopen()` Function

Next, the program uses `fopen()` to open the file. This function is declared in `stdio.h`. Its first argument is the name of the file to be opened; more exactly, it is the address of a string containing that name. The second argument is a string identifying the mode in which the file is to be opened. The C library provides for several possibilities, as shown in Table 13.1.

Table 13.1 **Mode Strings for `fopen()`**

Mode String	Meaning
`"r"`	Open a text file for reading.
`"w"`	Open a text file for writing, truncating an existing file to zero length, or creating the file if it does not exist.

Mode String	Meaning
`"a"`	Open a text file for writing, appending to the end of an existing file, or creating the file if it does not exist.
`"r+"`	Open a text file for update (that is, for both reading and writing).
`"w+"`	Open a text file for update (reading and writing), first truncating the file to zero length if it exists or creating the file if it does not exist.
`"a+"`	Open a text file for update (reading and writing), appending to the end of an existing file, or creating the file if it does not yet exist; the whole file can be read, but writing can only be appended.
`"rb"`, `"wb"`, `"ab"`, `"ab+"`, `"a+b"`, `"wb+"`, `"w+b"`, `"ab+"`, `"a+b"`	Like the preceding modes, except they use binary mode instead of text mode.
`"wx"`, `"wbx"`, `"w+x"`, `"wb+x"` or `"w+bx"`	(C11) Like the non-x modes, except they fail if the file already exists and they open a file in exclusive mode, if possible.

For systems such as Unix and Linux that have just one file type, the modes with the `b` are equivalent to the corresponding modes lacking the `b`.

The new C11 write modes with `x` provide a couple of features compared to the older write modes. First, if you try to open an existing file in one of the traditional write modes, `fopen()` truncates the file to zero length, thus losing the file contents. But the modes with `x` cause `fopen()` to fail instead, leaving the file unharmed. Second, to the extent that the environment allows, the exclusivity feature of the `x` modes keeps other programs or threads from accessing the file until the current process closes the file.

Caution!

If you use any of the `"w"` modes without an `x` for an existing file, the file contents are truncated so that your program can start with a clean slate. However, if you attempt to open an existing file with one of the C11 modes with an `x`, the attempt fails.

After your program successfully opens a file, `fopen()` returns a *file pointer*, which the other I/O functions can then use to specify the file. The file pointer (`fp` in this example) is of type pointer-to-`FILE`; `FILE` is a derived type defined in `stdio.h`. The pointer `fp` doesn't point to the actual file. Instead, it points to a data object containing information about the file, including information about the buffer used for the file's I/O. Because the I/O functions in the standard library use a buffer, they need to know where the buffer is. They also need to know how full the buffer is and which file is being used. This enables the functions to refill or empty the buffer when necessary. The data object pointed to by `fp` has all that information. (This data object is an example of a C structure, a topic we discuss in Chapter 14, "Structures and Other Data Forms.")

The `fopen()` function returns the null pointer (also defined in `stdio.h`) if it cannot open the file. This program exits if `fp` is `NULL`. The `fopen()` function can fail because the disk is full, because the file is not in the searched directory, because the name is illegal, because access is restricted, or because of a hardware problem, to name just a few reasons, so check for trouble; a little error-trapping can go a long way.

The `getc()` and `putc()` Functions

The two functions `getc()` and `putc()` work very much like `getchar()` and `putchar()`. The difference is that you must tell these newcomers which file to use. So the following old standby means "get a character from the standard input":

```
ch = getchar();
```

However, this statement means "get a character from the file identified by `fp`":

```
ch = getc(fp);
```

Similarly, this statement means "put the character `ch` into the file identified by the `FILE` pointer `fpout`":

```
putc(ch, fpout);
```

In the `putc()` argument list, the character comes first, and then the file pointer.

Listing 13.1 uses `stdout` for the second argument of `putc()`. It is defined in `stdio.h` as being the file pointer associated with the standard output, so `putc(ch,stdout)` is the same as `putchar(ch)`. Indeed, the latter function is normally defined as being the former. Similarly, `getchar()` is defined as being `getc()` using the standard input.

You may wonder why this example uses `putc()` instead of `putchar()`. One reason is to introduce the `putc()` function. The other is that you can easily convert this program to produce file output by using an argument other than `stdout`.

End-of-File

A program reading data from a file needs to stop when it reaches the end of the file. How can a program tell if it has reached the end? The `getc()` function returns the special value `EOF` if it tries to read a character and discovers it has reached the end of the file. So a C program discovers it has reached the end of a file only after it tries to read past the end of the file. (This is unlike the behavior of some languages, which use a special function to test for end-of-file *before* attempting a read.)

To avoid problems attempting to read an empty file, you should use an entry-condition loop (not a `do while` loop) for file input. Because of the design of `getc()` (and other C input functions), a program should attempt the first read before entering the body of the loop. So the following design is good:

```
// good design #1
int ch;             // int to hold EOF
FILE * fp;
fp = fopen("wacky.txt", "r");
ch = getc(fp);      // get initial input
while (ch != EOF)
{
    putchar(ch);    // process input
    ch = getc(fp);  // get next input
}
```

This can be condensed to the following design:

```
// good design #2
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (( ch = getc(fp)) != EOF)
{
    putchar(ch);  // process input
}
```

Because the input statement is part of the `while` test condition, it is executed before the program enters the body of the loop.

You should avoid a design of this sort:

```
// bad design (two problems)
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (ch != EOF)    // ch undetermined value first use
{
    ch = getc(fp);   // get input
    putchar(ch);     // process input
}
```

The first problem is that the first time `ch` is compared with `EOF`, it has not yet been assigned a value. The second problem is that if `getc()` does return `EOF`, the loop tries to process `EOF` as if it were a valid character. These defects are fixable. For example, you could initialize `ch` to a dummy value and stick an `if` statement inside the loop, but why bother when good designs are already available.

These cautions carry over to the other input functions. They also return an error signal (either `EOF` or the `NULL` pointer) after running into the end of a file.

The `fclose()` Function

The `fclose(fp)` function closes the file identified by `fp`, flushing buffers as needed. For a program less casual than this one, you would check to see whether the file had been closed successfully. The function `fclose()` returns a value of `0` if successful, and `EOF` if not:

```
if (fclose(fp) != 0)
    printf("Error in closing file %s\n", argv[1]);
```

The `fclose()` function can fail if, for example, the disk is full, a removable storage device has been removed, or there has been an I/O error.

Pointers to the Standard Files

The `stdio.h` file associates three file pointers with the three standard files automatically opened by C programs:

Standard File	File Pointer	Normally
Standard input	`stdin`	Your keyboard
Standard output	`stdout`	Your screen
Standard error	`stderr`	Your screen

These pointers are all type pointer-to-`FILE`, so they can be used as arguments to the standard I/O functions, just as `fp` was in the example. Let's move on to an example that creates a new file and writes to it.

A Simple-Minded File-Condensing Program

This next program copies selected data from one file to another. It opens two files simultaneously, using the `"r"` mode for one and the `"w"` mode for the other. The program (shown in Listing 13.2) condenses the contents of the first file by the brutal expedient of retaining only every third character. Finally, it places the condensed text into the second file. The name for the second file is the old name with `.red` (for reduced) appended. Using command-line arguments, opening more than one file simultaneously, and filename appending are generally quite useful techniques. This particular form of condensing is of more limited appeal, but it can have its uses, as you will see. (Again, it is a simple matter to modify this program to use standard I/O techniques instead of command-line arguments to provide filenames.)

Listing 13.2 **The `reducto.c` Program**

```
// reducto.c -- reduces your files by two-thirds!
#include <stdio.h>
#include <stdlib.h>    // for exit()
```

```
#include <string.h>

int main(int argc, char *argv[])
{
    FILE  *in, *out;   // declare two FILE pointers
    int ch;
    char name[LEN];    // storage for output filename
    int count = 0;

    // check for command-line arguments
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // set up input
    if ((in = fopen(argv[1], "r")) == NULL)
    {
        fprintf(stderr, "I couldn't open the file \"%s\"\n",
                argv[1]);
        exit(EXIT_FAILURE);
    }
    // set up output
    strncpy(name,argv[1], LEN - 5); // copy filename
    name[LEN - 5] = '\0';
    strcat(name,".red");             // append .red
    if ((out = fopen(name, "w")) == NULL)
    {                        // open file for writing
        fprintf(stderr,"Can't create output file.\n");
        exit(3);
    }
    // copy data
    while ((ch = getc(in)) != EOF)
        if (count++ % 3 == 0)
            putc(ch, out);  // print every 3rd char
    // clean up
    if (fclose(in) != 0 || fclose(out) != 0)
        fprintf(stderr,"Error in closing files\n");

    return 0;
}
```

Suppose the executable file is named `reducto` and that we apply it to a file called `Eddy`, which contains this single line:

```
So even Eddy came oven ready.
```

The command would be as follows:

```
reducto eddy
```

The output is written to a file called `eddy.red`. The program doesn't produce any onscreen output, but displaying the `eddy.red` file should reveal the following:

```
Send money
```

This example illustrates several programming techniques. Let's examine some of them now.

The `fprintf()` function is like `printf()`, except that it requires a file pointer as its first argument. We've used the `stderr` pointer to send error messages to the standard error; this is a standard C practice.

To construct the new name for the output file, the program uses `strncpy()` to copy the name `eddy` into the array `name`. The `LEN - 5` argument leaves room for the `.red` suffix and the final null character. No null character is copied if the `argv[2]` string is longer than `LEN – 5`, so the program adds a null character just in case. The first null character in `name` after the `strncpy()` call then is overwritten by the period in `.red` when the `strcat()` function appends that string, producing, in this case, `eddy.red`. We also checked to see whether the program succeeded in opening a file by that name. This is particularly important in some environments because a filename such as, say, `strange.c.red`, may be invalid. For example, you can't add extensions to extensions under traditional DOS. (The proper MS-DOS approach is to replace any existing extension with `.red`, so the reduced version of `strange.c` would be `strange.red`. You could use the `strchr()` function, for example, to locate the period, if any, in a name and copy only the part of the string before the period.)

This program had two files open simultaneously, so we declared two `FILE` pointers. Note that each file is opened and closed independently of the other. There are limits to how many files you can have open at one time. The limit depends on your system and implementation; the range is often 10 to 20. You can use the same file pointer for different files, provided those files are not open at the same time.

File I/O: `fprintf()`, `fscanf()`, `fgets()`, and `fputs()`

For each of the I/O functions in the preceding chapters, there is a similar file I/O function. The main distinction is that you need to use a `FILE` pointer to tell the new functions with which file to work. Like `getc()` and `putc()`, these functions require that you identify a file by using a pointer-to-`FILE`, such as `stdout`, or that you use the return value of `fopen()`.

The `fprintf()` and `fscanf()` Functions

The file I/O functions `fprintf()` and `fscanf()` work just like `printf()` and `scanf()`, except that they require an additional first argument to identify the proper file. You've already used `fprintf()`. Listing 13.3 illustrates both of these file I/O functions, along with the `rewind()` function.

Listing 13.3 **The `addaword.c` Program**

```
/* addaword.c -- uses fprintf(), fscanf(), and rewind() */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 41

int main(void)
{
    FILE *fp;
    char words[MAX];

    if ((fp = fopen("wordy", "a+")) == NULL)
    {
        fprintf(stdout,"Can't open \"wordy\" file.\n");
        exit(EXIT_FAILURE);
    }

    puts("Enter words to add to the file; press the #");
    puts("key at the beginning of a line to terminate.");
    while ((fscanf(stdin,"%40s", words) == 1)  && (words[0] != '#'))
        fprintf(fp, "%s\n", words);

    puts("File contents:");
    rewind(fp);           /* go back to beginning of file */
    while (fscanf(fp,"%s",words) == 1)
        puts(words);
    puts("Done!");
    if (fclose(fp) != 0)
        fprintf(stderr,"Error closing file\n");

    return 0;
}
```

This program enables you to add words to a file. By using the `"a+"` mode, the program can both read and write in the file. The first time the program is used, it creates the `wordy` file and enables you to place words in it, one word per line. When you use the program subsequently, it enables you to add (append) words to the previous contents. The append mode only enables you to add material to the end of the file, but the `"a+"` mode does enable you to read the whole file. The `rewind()` command takes the program to the file beginning so that the final `while` loop can print the file contents. Note that `rewind()` takes a file pointer argument.

Here's a sample run from a Unix environment (the executable program has been renamed `addaword`):

```
$ addaword
Enter words to add to the file; press the Enter
key at the beginning of a line to terminate.
The fabulous programmer
#
File contents:
The
fabulous
programmer
Done!
$ addaword
Enter words to add to the file; press the Enter
key at the beginning of a line to terminate.
enchanted the
large
#
File contents:
The
fabulous
programmer
enchanted
the
large
Done!
```

As you can see, `fprintf()` and `fscanf()` work like `printf()` and `scanf()`. Unlike `putc()`, the `fprintf()` and `fscanf()` functions take the `FILE` pointer as the first argument instead of as the last argument.

The `fgets()` and `fputs()` Functions

You met `fgets()` in Chapter 11. The first argument, as with the banished `gets()`, is the address (type `char *`) where input should be stored. The second argument is an integer representing the maximum size of the input string. The final argument is the file pointer identifying the file to be read. A function call, then, looks like this:

```
fgets(buf, STLEN, fp);
```

Here, `buf` is the name of a `char` array, `STLEN` is the maximum size of the string, and `fp` is the pointer-to-`FILE`.

As we saw earlier, the `fgets()` function reads input through the first newline character, until one fewer than the upper limit of characters is read, or until the end-of-file is found; `fgets()` then adds a terminating null character to form a string. Therefore, the upper limit represents the maximum number of characters plus the null character. If `fgets()` reads in a whole line before running into the character limit, it places the newline character, marking the end of the line into the string, just before the null character. The `fgets()` function returns the value `NULL` when it encounters `EOF`. You can use this to check for the end of a file. Otherwise, it returns the address passed to it.

The `fputs()` function takes two arguments: first, an address of a string and then a file pointer. It writes the string found at the pointed-to location into the indicated file. Unlike `puts()`, `fputs()` does not append a newline when it prints. A function call looks like this:

```
fputs(buf, fp);
```

Here, `buf` is the string address, and `fp` identifies the target file.

Because `fgets()` keeps the newline and `fputs()` doesn't add one, they work well in tandem. As Listing 11.8 showed, they work well together even if `STLEN` is smaller than the input line length.

Adventures in Random Access: `fseek()` and `ftell()`

The `fseek()` function enables you to treat a file like an array and move directly to any particular byte in a file opened by `fopen()`. To see how it works, let's create a program (see Listing 13.4) that displays a file in reverse order. Note that `fseek()` has three arguments and returns an `int` value. The `ftell()` function returns the current position in a file as a `long` value.

Listing 13.4 **The `reverse.c` Program**

```
/* reverse.c -- displays a file in reverse order */
#include <stdio.h>
#include <stdlib.h>
#define CNTL_Z '\032'   /* eof marker in DOS text files */
#define SLEN 81
int main(void)
{
    char file[SLEN];
    char ch;
    FILE *fp;
    long count, last;

    puts("Enter the name of the file to be processed:");
    scanf("%80s", file);
    if ((fp = fopen(file,"rb")) == NULL)
    {                               /* read-only mode   */
        printf("reverse can't open %s\n", file);
        exit(EXIT_FAILURE);
    }
    fseek(fp, 0L, SEEK_END);        /* go to end of file */
    last = ftell(fp);
    for (count = 1L; count <= last; count++)
    {
        fseek(fp, -count, SEEK_END); /* go backward      */
        ch = getc(fp);
        if (ch != CNTL_Z && ch != '\r')  /* MS-DOS files */
            putchar(ch);
    }
    putchar('\n');
    fclose(fp);

    return 0;
}
```

Here is the output for a sample file:

```
Enter the name of the file to be processed:
Cluv

.C ni eno naht ylevol erom margorp a
ees  reven llahs I taht kniht I
```

This program uses the binary mode so that it can deal with both MS-DOS text and Unix files. However, it may not work correctly in an environment that uses some other format for text files.

> **Note**
>
> If you run the program from a command-line environment, this program expects the filename to be in the same directory (or folder) as the executable program. If you run the program from an IDE, where the program looks depend on the implementation. For example, by default Microsoft Visual Studio 2012 looks in the directory containing the source code and Xcode 4.6 looks in the directory containing the executable file.

We now need to discuss three topics: how `fseek()` and `ftell()` work, how to use a binary stream, and how to make the program portable.

How `fseek()` and `ftell()` Work

The first of the three arguments to `fseek()` is a `FILE` pointer to the file being searched. The file should have been opened by using `fopen()`.

The second argument to `fseek()` is called the *offset*. This argument tells how far to move from the starting point (see the following list of mode starting points). The argument must be a `long` value. It can be positive (move forward), negative (move backward), or zero (stay put).

The third argument is the mode, and it identifies the starting point. Since the ANSI standard, the `stdio.h` header file specifies the following manifest constants for the mode:

Mode	Measures Offset From
`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current position
`SEEK_END`	End of file

Older implementations may lack these definitions and, instead, use the numeric values `0L`, `1L`, and `2L`, respectively, for these modes. Recall that the `L` suffix identifies type `long` values. Or the implementation might have the constants defined in a different header file. When in doubt, consult your usage manual or the online manual.

Here are some sample function calls, where `fp` is a file pointer:

```
fseek(fp, 0L, SEEK_SET);   // go to the beginning of the file
fseek(fp, 10L, SEEK_SET);  // go 10 bytes into the file
fseek(fp, 2L, SEEK_CUR);   // advance 2 bytes from the current position
fseek(fp, 0L, SEEK_END);   // go to the end of the file
fseek(fp, -10L, SEEK_END); // back up 10 bytes from the end of the file
```

There are some possible restrictions on these calls; we'll get back to that topic in a moment or two.

The value returned by `fseek()` is `0` if everything is okay, and `-1` if there is an error, such as attempting to move past the bounds of the file.

The `ftell()` function is type `long`, and it returns the current file location. Under ANSI C, it is declared in `stdio.h`. As originally implemented in Unix, `ftell()` specifies the file position by returning the number of bytes from the beginning, with the first byte being byte 0, and so on. Under ANSI C, this definition applies to files opened in the binary mode, but not necessarily to files opened in the text mode. That is one reason Listing 13.4 uses the binary mode.

Now we can examine the basic elements of Listing 13.4. First, the statement

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

sets the position to an offset of 0 bytes from the file end. That is, it sets the position to the end of the file. Next, the statement

```
last = ftell(fp);
```

assigns to `last` the number of bytes from the beginning to the end of the file.

Next is this loop:

```
for (count = 1L; count <= last; count++)
{
  fseek(fp, -count, SEEK_END);    /* go backward */
     ch = getc(fp);
 }
```

The first cycle positions the program at the first character before the end of the file (that is, at the file's final character). Then the program prints that character. The next loop positions the program at the preceding character and prints it. This process continues until the first character is reached and printed.

Binary Versus Text Mode

We designed Listing 13.4 to work in both the Unix and the MS-DOS environments. Unix has only one file format, so no special adjustments are needed. MS-DOS, however, does require extra attention. Many MS-DOS editors mark the end of a text file with the character Ctrl+Z. When such a file is opened in the text mode, C recognizes this character as marking the end of the file. When the same file is opened in the binary mode, however, the Ctrl+Z character is just another character in the file, and the actual end-of-file comes later. It might come immediately after the Ctrl+Z, or the file could be padded with null characters to make the size a multiple of, say, 256. Null characters don't print under DOS, and we included code to prevent the program from trying to print the Ctrl+Z character.

Another difference is one we've mentioned before: MS-DOS represents a text file newline with the `\r\n` combination. A C program opening the same file in a text mode "sees" `\r\n` as a simple `\n`, but, when using the binary mode, the program sees both characters. Therefore, we included coding to suppress printing `\r`. Because a Unix text file normally contains neither Ctrl+Z nor `\r`, this extra coding does not affect most Unix text files.

The `ftell()` function may work differently in the text mode than in the binary mode. Many systems have text file formats that are different enough from the Unix model that a byte count from the beginning of the file is not a meaningful quantity. ANSI C states that, for the text mode, `ftell()` returns a value that can be used as the second argument to `fseek()`. For MS-DOS, for example, `ftell()` can return a count that sees `\r\n` as a single byte.

Portability

Ideally, `fseek()` and `ftell()` should conform to the Unix model. However, differences in real systems sometimes make this impossible. Therefore, ANSI provides lowered expectations for these functions. Here are some limitations:

- In the binary mode, implementations need not support the `SEEK_END` mode. Listing 13.4, then, is not guaranteed to be portable. A more portable approach is to read the whole file byte-by-byte until the end. But reading the file sequentially to find the end is slower than simply jumping to the end. The C preprocessor conditional compilation directives, discussed in Chapter 16, "The C Preprocessor and the C Library," provide a systematic way to handle alternative code choices.

- In the text mode, the only calls to `fseek()` that are guaranteed to work are these:

Function Call	Effect
`fseek(file, 0L, SEEK_SET)`	Go to the beginning of the file.
`fseek(file, 0L, SEEK_CUR)`	Stay at the current position.
`fseek(file, 0L, SEEK_END)`	Go to the file's end.
`fseek(file,ftell-pos, SEEK_SET)`	Go to position `ftell-pos` from the beginning; `ftell-pos` is a value returned by `ftell()`.

Fortunately, many common environments allow stronger implementations of these functions.

The `fgetpos()` and `fsetpos()` Functions

One potential problem with `fseek()` and `ftell()` is that they limit file sizes to values that can be represented by type `long`. Perhaps two-billion bytes seem more than adequate, but the ever-increasing capacities of storage devices makes larger files possible. ANSI C introduced two new positioning functions designed to work with larger file sizes. Instead of using a `long` value to represent a position, it uses a new type, called `fpos_t` (for file position type) for that purpose. The `fpos_t` type is not a fundamental type; rather, it is defined in terms of other types. A variable or data object of `fpos_t` type can specify a location within a file, and it cannot be an array type, but its nature is not specified beyond that. Implementations can then provide a type to meet the needs of a particular platform; the type could, for example, be implemented as a structure.

ANSI C does define how `fpos_t` is used. The `fgetpos()` function has this prototype:

```
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

When called, it places an `fpos_t` value in the location pointed to by `pos`; the value describes a location in the file. The function returns zero if successful and a nonzero value for failure.

The `fsetpos()` function has this prototype:

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

When called, it uses the `fpos_t` value in the location pointed to by `pos` to set the file pointer to the location indicated by that value. The function returns zero if successful and a nonzero value for failure. The `fpos_t` value should have been obtained by a previous call to `fgetpos()`.

Behind the Scenes with Standard I/O

Now that you've seen some of the features of the standard I/O package, let's examine a representative conceptual model to see how standard I/O works.

Normally, the first step in using standard I/O is to use `fopen()` to open a file. (Recall, however, that the `stdin`, `stdout`, and `stderr` files are opened automatically.) The `fopen()` function not only opens a file but sets up a buffer (two buffers for read-write modes), and it sets up a data structure containing data about the file and about the buffer. Also, `fopen()` returns a pointer to this structure so that other functions know where to find it. Assume that this value is assigned to a pointer variable named `fp`. The `fopen()` function is said to "open a stream." If the file is opened in the text mode, you get a text stream, and if the file is opened in the binary mode, you get a binary stream.

The data structure typically includes a file position indicator to specify the current position in the stream. It also has indicators for errors and end-of-file, a pointer to the beginning of the buffer, a file identifier, and a count for the number of bytes actually copied into the buffer.

Let's concentrate on file input. Usually, the next step is to call on one of the input functions declared in `stdio.h`, such as `fscanf()`, `getc()`, or `fgets()`. Calling any one of these functions causes a chunk of data to be copied from the file to the buffer. The buffer size is implementation dependent, but it typically is 512 bytes or some multiple thereof, such as 4,096 or 16,384. (As hard drives and computer memories get larger, the choice of buffer size tends to get larger, too.) In addition to filling the buffer, the initial function call sets values in the structure pointed to by `fp`. In particular, the current position in the stream and the number of bytes copied into the buffer are set. Usually the current position starts at byte 0.

After the data structure and buffer are initialized, the input function reads the requested data from the buffer. As it does so, the file position indicator is set to point to the character following the last character read. Because all the input functions from the `stdio.h` family use the same buffer, a call to any one function resumes where the previous call to any of the functions stopped.

When an input function finds that it has read all the characters in the buffer, it requests that the next buffer-sized chunk of data be copied from the file into the buffer. In this manner, the input functions can read all the file contents up to the end of the file. After a function reads the last character of the final buffer's worth of data, it sets the end-of-file indicator to true. The next call to an input function then returns `EOF`.

In a similar manner, output functions write to a buffer. When the buffer is filled, the data is copied to the file.

Other Standard I/O Functions

The ANSI standard library contains over three dozen functions in the standard I/O family. Although we don't cover them all here, we will briefly describe a few more to give you a better idea of what is available. We'll list each function by its C prototype to indicate its arguments and return values. Of those functions we discuss here, all but `setvbuf()` are also available in pre-ANSI implementations. Reference Section V, "The Standard ANSI C Library with C99 Additions," lists the full ANSI C standard I/O package.

The `int ungetc(int c, FILE *fp)` Function

The `int ungetc()` function pushes the character specified by `c` back onto the input stream. If you push a character onto the input stream, the next call to a standard input function reads that character (see Figure 13.2). Suppose, for example, that you want a function to read characters up to, but not including, the next colon. You can use `getchar()` or `getc()` to read characters until a colon is read and then use `ungetc()` to place the colon back in the input stream. The ANSI C standard guarantees only one pushback at a time. If an implementation permits you to push back several characters in a row, the input functions read them in the reversed order of pushing.

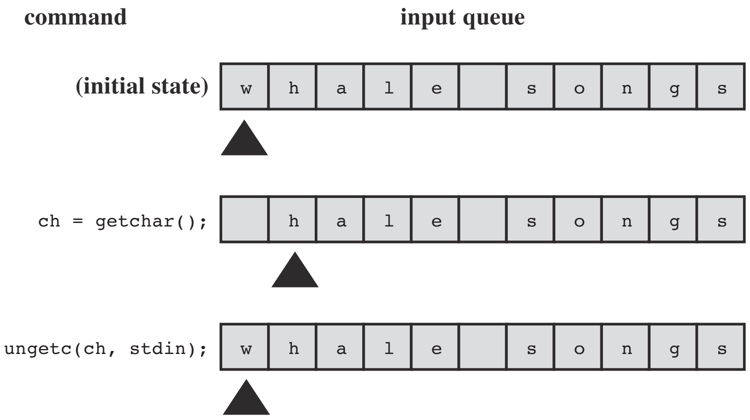

Figure 13.2 The `ungetc()` function.

The `int fflush()` Function

The prototype for `fflush()` is this:

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

Calling the `fflush()` function causes any unwritten data in the output buffer to be sent to the output file identified by `fp`. This process is called *flushing a buffer*. If `fp` is the null pointer, all output buffers are flushed. The effect of using `fflush()` on an input stream is undefined. You can use it with an update stream (any of the read-write modes), provided that the most recent operation using the stream was not input.

The `int setvbuf()` Function

The prototype for `setvbuf()` is this:

```
int setvbuf(FILE * restrict fp, char * restrict buf, int mode, size_t size);
```

The `setvbuf()` function sets up an alternative buffer to be used by the standard I/O functions. It is called after the file has been opened and before any other operations have been performed on the stream. The pointer `fp` identifies the stream, and `buf` points to the storage to be used. If the value of `buf` is not `NULL`, you must create the buffer. For instance, you could declare an array of 1,024 `char`s and pass the address of that array. However, if you use `NULL` for the value of `buf`, the function allocates a buffer itself. The `size` variable tells `setvbuf()` how big the array is. (The `size_t` type is a derived integer type; see Chapter 5, "Operators, Expressions, and Statements.") The `mode` is selected from the following choices: `_IOFBF` means fully buffered (buffer flushed when full), `_IOLBF` means line-buffered (buffer flushed when full or when a newline is written), and `_IONBF` means nonbuffered. The function returns zero if successful, nonzero otherwise.

Suppose you have a program that works with stored data objects having, say, a size of 3,000 bytes each. You could use `setvbuf()` to create a buffer whose size is a multiple of the data object's size.

Binary I/O: `fread()` and `fwrite()`

The `fread()` and `fwrite()` functions are next on the list, but first some background. The standard I/O functions you've used to this point are text oriented, dealing with characters and strings. What if you want to save numeric data in a file? True, you can use `fprintf()` and the `%f` format to save a floating-point value, but then you are saving it as a sequence of characters. For example, the code

```
double num = 1./3.;
fprintf(fp,"%f", num);
```

saves `num` as a sequence of eight characters: `0.333333`. Using a `%.2f` specifier saves it as four characters: `0.33`. Using a `%.12f` specifier saves it as 14 characters: `0.333333333333`. Changing the specifier alters the amount of space needed to store the value; it can also result in different values being stored. After the value of `num` is stored as `0.33`, there is no way to get back the full precision when the file is read. In general, `fprintf()` converts numeric values to character data, possibly altering the value.

The most accurate and consistent way to store a number is to use the same pattern of bits that the computer does. Therefore, a `double` value should be stored in a size `double` unit. When data is stored in a file using the same representation that the program uses, we say that the data is stored in *binary form*. There is no conversion from numeric forms to character sequences. For standard I/O, the `fread()` and `fwrite()` functions provide this binary service (see Figure 13.3).

Actually, as you probably recall, all data is stored in binary form. Even characters are stored using the binary representation of the character code. However, if all data in the file is interpreted as character codes, we say that the file contains text data. If some or all of the data is interpreted as numeric data in binary form, we say that the file contains binary data. (Also, files in which the data represents machine-language instructions are binary files.)

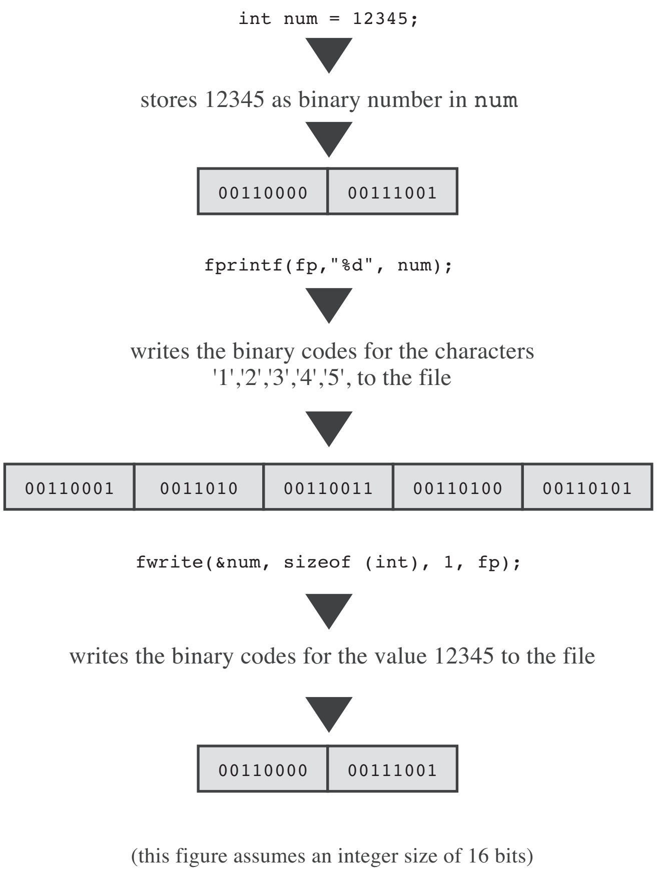
int num = 12345;
stores 12345 as binary number in num
00110000
00111001
fprintf(fp,"%d", num);
writes the binary codes for the characters '1','2','3','4','5', to the file
00110001
0011010
00110011
00110100
00110101
fwrite(&num, sizeof (int), 1, fp);
writes the binary codes for the value 12345 to the file
00110000
00111001
(this figure assumes an integer size of 16 bits)

Figure 13.3 Binary and text output.

The uses of the terms *binary* and *text* can get confusing. ANSI C recognizes two modes for opening files: binary and text. Many operating systems recognize two file formats: binary and text. Information can be stored or read as binary data or as text data. These are all related, but not identical. You can open a text format file in the binary mode. You can store text in a binary format file. You can use `getc()` to copy files containing binary data. In general, however, you use the binary mode to store binary data in a binary format file. Similarly, you most often use text data in text files opened in the text format. (Files produced by word processors typically are binary files because they contain a lot of nontext information describing fonts and formatting.)

The `size_t fwrite()` Function

The prototype for `fwrite()` is this:

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
                FILE * restrict fp);
```

The `fwrite()` function writes binary data to a file. The `size_t` type is defined in terms of the standard C types. It is the type returned by the `sizeof` operator. Typically, it is `unsigned int`, but an implementation can choose another type. The pointer `ptr` is the address of the chunk of data to be written. Also, `size` represents the size, in bytes, of the chunks to be written, and `nmemb` represents the number of chunks to be written. As usual, `fp` identifies the file to be written to. For instance, to save a data object (such as an array) that is 256 bytes in size, you can do this:

```
char buffer[256];
fwrite(buffer, 256, 1, fp);
```

This call writes one chunk of 256 bytes from `buffer` to the file. Or, to save an array of 10 `double` values, you can do this:

```
double earnings[10];
fwrite(earnings, sizeof (double), 10, fp);
```

This call writes data from the `earnings` array to the file in 10 chunks, each of size `double`.

You probably noticed the odd declaration of `const void * restrict ptr` in the `fwrite()` prototype. One problem with `fwrite()` is that its first argument is not a fixed type. For instance, the first example used `buffer`, which is type pointer-to-`char`, and the second example used `earnings`, which is type pointer-to-`double`. Under ANSI C function prototyping, these actual arguments are converted to the pointer-to-`void` type, which acts as a sort of catch-all type for pointers. (Pre-ANSI C uses type `char *` for this argument, requiring you to typecast actual arguments to that type.)

The `fwrite()` function returns the number of items successfully written. Normally, this equals `nmemb`, but it can be less if there is a write error.

The `size_t fread()` Function

The prototype for `fread()` is this:

```
size_t fread(void * restrict ptr, size_t size, size_t nmemb,
                FILE * restrict fp);
```

The `fread()` function takes the same set of arguments that `fwrite()` does. This time `ptr` is the address of the memory storage into which file data is read, and `fp` identifies the file to be read. Use this function to read data that was written to a file using `fwrite()`. For example, to recover the array of 10 `double`s saved in the previous example, use this call:

```
double earnings[10];
fread(earnings, sizeof (double), 10, fp);
```

This call copies 10 size `double` values into the `earnings` array.

The `fread()` function returns the number of items successfully read. Normally, this equals `nmemb`, but it can be less if there is a read error or if the end-of-file is reached.

The `int feof(FILE *fp)` and `int ferror(FILE *fp)` Functions

When the standard input functions return `EOF`, this usually means they have reached the end of a file. However, it can also indicate that a read error has occurred. The `feof()` and `ferror()` functions enable you to distinguish between the two possibilities. The `feof()` function returns a nonzero value if the last input call detected the end-of-file, and it returns zero otherwise. The `ferror()` function returns a nonzero value if a read or write error has occurred, and it returns zero otherwise.

An `fread()` and `fwrite()` Example

Let's use some of these functions in a program that appends the contents from a list of files to the end of another file. One problem is passing the file information to the program. This can be done interactively or by using command-line arguments. We'll take the first approach, which suggests a plan along the following lines:

- Request a name for the destination file and open it.
- Use a loop to request source files.
- Open each source file in turn in the read mode and add it to the append file.

To illustrate `setvbuf()`, we'll use it to specify a different buffer size. The next stage of refinement examines opening the append file. We will use the following steps:

1. Open the destination file in the append mode.
2. If this cannot be done, quit.
3. Establish a 4,096-byte buffer for this file.
4. If this cannot be done, quit.

Similarly, we can refine the copying portion by doing the following for each file:

- If it is the same as the append file, skip to the next file.
- If it cannot be opened in the read mode, skip to the next file.
- Add the contents of the file to the append file.

For a grand finale, the program rewinds the append file to the beginning and displays the contents.

For practice, we'll use `fread()` and `fwrite()` for the copying. Listing 13.5 shows the result.

Listing 13.5 **The `append.c` Program**

```
/* append.c -- appends files to a file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 4096
#define SLEN 81
void append(FILE *source, FILE *dest);
char * s_gets(char * st, int n);

int main(void)
{
    FILE *fa, *fs;    // fa for append file, fs for source file
    int files = 0;  // number of files appended
    char file_app[SLEN];  // name of append file
    char file_src[SLEN];  // name of source file
    int ch;

    puts("Enter name of destination file:");
    s_gets(file_app, SLEN);
    if ((fa = fopen(file_app, "a+")) == NULL)
    {
        fprintf(stderr, "Can't open %s\n", file_app);
        exit(EXIT_FAILURE);
    }
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
        fputs("Can't create output buffer\n", stderr);
        exit(EXIT_FAILURE);
    }
    puts("Enter name of first source file (empty line to quit):");
    while (s_gets(file_src, SLEN) && file_src[0] != '\0')
    {
        if (strcmp(file_src, file_app) == 0)
            fputs("Can't append file to itself\n",stderr);
        else if ((fs = fopen(file_src, "r")) == NULL)
            fprintf(stderr, "Can't open %s\n", file_src);
        else
        {
            if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
            {
                fputs("Can't create input buffer\n",stderr);
                continue;
            }
            append(fs, fa);
            if (ferror(fs) != 0)
                fprintf(stderr,"Error in reading file %s.\n",
                        file_src);
            if (ferror(fa) != 0)
                fprintf(stderr,"Error in writing file %s.\n",
                        file_app);
            fclose(fs);
            files++;
            printf("File %s appended.\n", file_src);
            puts("Next file (empty line to quit):");
        }
    }
    printf("Done appending. %d files appended.\n", files);
    rewind(fa);
    printf("%s contents:\n", file_app);
    while ((ch = getc(fa)) != EOF)
        putchar(ch);
    puts("Done displaying.");
    fclose(fa);

    return 0;
}

void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE]; // allocate once

    while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
        fwrite(temp, sizeof (char), bytes, dest);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

If `setvbuf()` is unable to create the buffer, it returns a nonzero value, and the code then terminates the program. Similar coding establishes a 4,096-byte buffer for the file currently being copied. By using `NULL` as the second argument to `setvbuf()`, we let that function allocate storage for the buffer.

The program uses `s_gets()` instead of `scanf()` to get the file name because `scanf()` skips over whitespace and thus doesn't detect an empty line. It uses `s_gets()` instead of a simple `fgets()` because the latter keeps the newline in the string.

This code prevents the program from trying to append a file to itself:

```
if (strcmp(file_src, file_app) == 0)
    fputs("Can't append file to itself\n",stderr);
```

The argument `file_app` represents the name of the destination file, and `file_src` represents the name of the file currently being processed.

The `append()` function does the copying. Instead of copying a byte at a time, it uses `fread()` and `fwrite()` to copy 4,096 bytes at a time:

```
void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE]; // allocate once

    while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
        fwrite(temp, sizeof (char), bytes, dest);
}
```

Because the file specified by `dest` is opened in the append mode, each source file is added to the end of the destination file, one after the other. Note that the `temp` array is static duration (meaning it's allocated at compile time, not each time the `append()` function is called) and block scope (meaning that it is private to the function).

The example uses text-mode files; by using the `"ab+"` and `"rb"` modes, it could handle binary files.

Random Access with Binary I/O

Random access is most often used with binary files written using binary I/O, so let's look at a short example. The program in Listing 13.6 creates a file of `double` numbers and then lets you access the contents.

Listing 13.6 **The `randbin.c` Program**

```
/* randbin.c -- random access, binary i/o */
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000

int main()
{
    double numbers[ARSIZE];
    double value;
    const char * file = "numbers.dat";
    int i;
    long pos;
    FILE *iofile;

    // create a set of double values
    for(i = 0; i < ARSIZE; i++)
        numbers[i] = 100.0 * i + 1.0 / (i + 1);
    // attempt to open file
    if ((iofile = fopen(file, "wb")) == NULL)
    {
        fprintf(stderr, "Could not open %s for output.\n", file);
        exit(EXIT_FAILURE);
    }
    // write array in binary format to file
    fwrite(numbers, sizeof (double), ARSIZE, iofile);
    fclose(iofile);
    if ((iofile = fopen(file, "rb")) == NULL)
    {
        fprintf(stderr,
                "Could not open %s for random access.\n", file);
        exit(EXIT_FAILURE);
    }
    // read selected items from file
    printf("Enter an index in the range 0-%d.\n", ARSIZE - 1);
    while (scanf("%d", &i) == 1 && i >= 0 && i < ARSIZE)
    {
        pos = (long) i * sizeof(double); // calculate offset
        fseek(iofile, pos, SEEK_SET);    // go there
        fread(&value, sizeof (double), 1, iofile);
        printf("The value there is %f.\n", value);
        printf("Next index (out of range to quit):\n");
    }
    // finish up
    fclose(iofile);
    puts("Bye!");

    return 0;
}
```

First, the program creates an array and places some values into it. Then it creates a file called numbers.dat in binary mode and uses `fwrite()` to copy the array contents to the file. The 64-bit pattern for each double value is copied from memory to the file. You can't read the resulting binary file with a text editor because the values are not translated to strings. However, each value is stored in the file precisely as it was stored in memory, so there is no loss of precision. Furthermore, each value occupies exactly 64 bits of storage in the file, so it is a simple matter to calculate the location of each value.

The second part of the program opens the file for reading and asks the user to enter the index for a value. Multiplying the index times the number of bytes per `double` yields the location in the file. The program then uses `fseek()` to go to that location and `fread()` to read the value there. Note that there are no format specifiers. Instead, `fread()` copies the 8 bytes, starting at that location, into the memory location indicated by `&value`. Then the program can use `printf()` to display `value`. Here is a sample run:

```
Enter an index in the range 0-999.
500
The value there is 50000.001996.
Next index (out of range to quit):
900
The value there is 90000.001110.
Next index (out of range to quit):
0
The value there is 1.000000.
Next index (out of range to quit):
-1
Bye!
```

Key Concepts

A C program views input as a stream of bytes; the source of this stream could be a file, an input device (such as a keyboard), or even the output of another program. Similarly, a C program views output as a stream of bytes; the destination could be a file, a video display, and so on.

How C interprets an input stream or output stream of bytes depends on which input/output functions you use. A program can read and store the bytes unaltered, or it can interpret the bytes as characters, which, in turn, can be interpreted as ordinary text or as the text representation of numbers. Similarly, on output, the functions you use determine whether binary values are transferred unaltered or converted to text or textual representations of numbers. If you have numeric data that you want to save and recover with no loss of precision, use the binary mode and the `fread()` and `fwrite()` functions. If you're saving text information and want to create files that can be viewed with ordinary text editors, use the text mode and functions such as `getc()` and `fprintf()`.

To access a file, you need to create a file pointer (type `FILE *`) and associate the pointer with a particular filename. Subsequent code then uses the pointer, not the filename, when dealing with the file.

It's important to understand how C handles the end-of-file concept. Typically, a file-reading program uses a loop to read input until reaching the end of file. The C input functions don't detect end-of-file until they attempt to read past the end. This means that testing for end-of-file should occur immediately *after* an attempted read. You can use the two-file-input models labeled "good design" in the "End-of-File" section of this chapter as a guide.

Summary

Writing to and reading from files is essential for most C programs. Most C implementations offer both low-level I/O services and standard high-level I/O services for these purposes. Because the ANSI C library includes the standard I/O services but not the low-level services, the standard package is more portable.

The standard I/O package automatically creates input and output buffers to speed up data transfer. The `fopen()` function opens a file for standard I/O and creates a data structure designed to hold information about the file and the buffer. The `fopen()` function returns a pointer to that data structure, and this pointer is used by other functions to identify the file to be processed. The `feof()` and `ferror()` functions report the reason an I/O operation failed.

C views input as a stream of bytes. If you use `fread()`, C views the input as binary values to be placed into whichever storage location you indicate. If you use `fscanf()`, `getc()`, `fgets()`, or any of the related functions, C views each byte as being a character code. The `fscanf()` and `scanf()` functions then attempt to translate the character code into other types, as indicated by the format specifiers. For example, the `%f` specifier would translate an input of `23` into a floating-point value, the `%d` specifier would translate the same input into an integer value, and the `%s` specifier would save the character input as a string. The `getc()` and `fgets()` family of functions leave the input as character code and store it either in `char` variables as individual characters or in `char` arrays as strings. Similarly, `fwrite()` places binary data directly into the output stream, whereas the other output functions convert noncharacter data to character representations before placing it in the output stream.

ANSI C provides two file-opening modes: binary and text. When a file is opened in binary mode, it can be read byte-for-byte. When a file is opened in text mode, its contents may be mapped from the system representation of text to the C representation. For Unix and Linux systems, the two modes are identical.

The input functions `getc()`, `fgets()`, `fscanf()`, and `fread()` normally read a file sequentially, starting at the beginning of the file. However, the `fseek()` and `ftell()` functions let a program move to an arbitrary position in a file, enabling random access. Both `fgetpos()` and `fsetpos()` extend similar capabilities to larger files. Random access works better in the binary mode than in the text mode.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. What's wrong with this program?

```
int main(void)
{
   int * fp;
   int k;

   fp = fopen("gelatin");
   for (k = 0; k < 30; k++)
       fputs(fp, "Nanette eats gelatin.");
   fclose("gelatin");
   return 0;
}
```

2. What would the following program do? (Assume it's run in a command-line environment.)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(int argc, char *argv[])
{
    int ch;
    FILE *fp;

    if (argc < 2)
      exit(EXIT_FAILURE);
    if ( (fp = fopen(argv[1], "r")) == NULL)
         exit(EXIT_FAILURE);
    while ( (ch= getc(fp)) != EOF )
        if( isdigit(ch) )
```

```
            putchar(ch);
        fclose (fp);

        return 0;
}
```

3. Suppose you have these statements in a program:

   ```
   #include <stdio.h>
   FILE * fp1,* fp2;
   char ch;

   fp1 = fopen("terky", "r");
   fp2 = fopen("jerky", "w");
   ```

   Also, suppose that both files were opened successfully. Supply the missing arguments in the following function calls:

   a. `ch = getc();`

   b. `fprintf( ,"%c\n", );`

   c. `putc( , );`

   d. `fclose(); /* close the terky file */`

4. Write a program that takes zero command-line arguments or one command-line argument. If there is one argument, it is interpreted as the name of a file. If there is no argument, the standard input (`stdin`) is to be used for input. Assume that the input consists entirely of floating-point numbers. Have the program calculate and report the arithmetic mean (the average) of the input numbers.

5. Write a program that takes two command-line arguments. The first is a character, and the second is a filename. The program should print only those lines in the file containing the given character.

Note

Lines in a file are identified by a terminating `'\n'`. Assume that no line is more than 256 characters long. You might want to use `fgets()`.

6. What's the difference between binary files and text files on the one hand versus binary streams and text streams on the other?

7. a. What is the difference between saving `8238201` by using `fprintf()` and saving it by using `fwrite()`?

   b. What is the difference between saving the character *S* by using `putc()` and saving it by using `fwrite()`?

8. What's the difference among the following?

```
printf("Hello, %s\n", name);
fprintf(stdout, "Hello, %s\n", name);
fprintf(stderr, "Hello, %s\n", name);
```

9. The `"a+"`, `"r+"`, and `"w+"` modes all open files for both reading and writing. Which one is best suited for altering material already present in a file?

Programming Exercises

1. Modify Listing 13.1 so that it solicits the user to enter the filename and reads the user's response instead of using command-line arguments.

2. Write a file-copy program that takes the original filename and the copy file from the command line. Use standard I/O and the binary mode, if possible.

3. Write a file copy program that prompts the user to enter the name of a text file to act as the source file and the name of an output file. The program should use the `toupper()` function from `ctype.h` to convert all text to uppercase as it's written to the output file. Use standard I/O and the text mode.

4. Write a program that sequentially displays onscreen all the files listed in the command line. Use `argc` to control a loop.

5. Modify the program in Listing 13.5 so that it uses a command-line interface instead of an interactive interface.

6. Programs using command-line arguments rely on the user's memory of how to use them correctly. Rewrite the program in Listing 13.2 so that, instead of using command-line arguments, it prompts the user for the required information.

7. Write a program that opens two files. You can obtain the filenames either by using command-line arguments or by soliciting the user to enter them.

   a. Have the program print line 1 of the first file, line 1 of the second file, line 2 of the first file, line 2 of the second file, and so on, until the last line of the longer file (in terms of lines) is printed.

   b. Modify the program so that lines with the same line number are printed on the same line.

8. Write a program that takes as command-line arguments a character and zero or more filenames. If no arguments follow the character, have the program read the standard input. Otherwise, have it open each file in turn and report how many times the character appears in each file. The filename and the character itself should be reported along with the count. Include error-checking to see whether the number of arguments is correct and whether the files can be opened. If a file can't be opened, have the program report that fact and go on to the next file.

9. Modify the program in Listing 13.3 so that each word is numbered according to the order in which it was added to the list, starting with 1. Make sure that, when the program is run a second time, new word numbering resumes where the previous numbering left off.

10. Write a program that opens a text file whose name is obtained interactively. Set up a loop that asks the user to enter a file position. The program then should print the part of the file starting at that position and proceed to the next newline character. Let negative or nonnumeric input terminate the user-input loop.

11. Write a program that takes two command-line arguments. The first is a string; the second is the name of a file. The program should then search the file, printing all lines containing the string. Because this task is line oriented rather than character oriented, use `fgets()` instead of `getc()`. Use the standard C library function `strstr()` (briefly described in exercise 7 of Chapter 11) to search each line for the string. Assume no lines are longer than 255 characters.

12. Create a text file consisting of 20 rows of 30 integers. The integers should be in the range 0–9 and be separated by spaces. The file is a digital representation of a picture, with the values `0` through `9` representing increasing levels of darkness. Write a program that reads the contents of the file into a 20-by-30 array of `int`s. In a crude approach toward converting this digital representation to a picture, have the program use the values in this array to initialize a 20-by-31 array of `char`s, with a `0` value corresponding to a space character, a `1` value to the period character, and so on, with each larger number represented by a character that occupies more space. For example, you might use # to represent 9. The last character (the 31st) in each row should be a null character, making it an array of 20 strings. Have the program display the resulting picture (that is, print the strings) and also store the result in a text file. For example, suppose you start with this data:

```
0 0 9 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 5 8 9 9 8 5 5 2 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 1 9 8 5 4 5 2 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 5 8 9 9 8 5 0 4 5 2 0 0 0 0 0 0 0 0
0 0 9 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 4 5 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 1 8 5 0 0 0 4 5 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 4 5 2 0 0 0 0 0
5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5 5
```

```
8 8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 0 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 3 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8
5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5 5
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 6 6 0 0 0 0 0 0
0 0 0 0 2 2 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0 0
0 0 0 0 3 3 0 0 0 0 0 0 5 8 9 9 8 5 0 5 6 1 1 1 1 6 5 0 0 0
0 0 0 0 4 4 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0 0
0 0 0 0 5 5 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 6 6 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
```

For one particular choice of output characters, the output looks like this:

```
  #         *%##%*'
    #       *%##%**'
            *%.#%*~*'
    #       *%##%* ~*'
  #         *%##%*  ~*'
            *%#.%*   ~*'
            *%##%*    ~*'
*************%##%*************
%%%%%%%%%%%%*%##%*%%%%%%%%%%%%
#### #################:#######
%%%%%%%%%%%%*%##%*%%%%%%%%%%%%
*************%##%*************
            *%##%*
            *%##%*    ==
    ''      *%##%*  *=  =*
    ::      *%##%* *=....=*
    ~~      *%##%*  *=  =*
    **      *%##%*    ==
            *%##%*
            *%##%*
```

13. Do Programming Exercise 12, but use variable-length arrays (VLAs) instead of standard arrays.

14. Digital images, particularly those radioed back from spacecraft, may have glitches. Add a de-glitching function to programming exercise 12. It should compare each value to its immediate neighbors to the left and right, above and below. If the value differs by more than 1 from each of its neighbors, replace the value with the average of the neighboring values. You should round the average to the nearest integer value. Note that the points along the boundaries have fewer than four neighbors, so they require special handling.