CHAPTER 6

Data Files

A *file* is a collection of data that is named and saved in the secondary storage (like on a disk or tape). The contents of a file can be retrieved and modified as per the requirements of the storage.

Every piece of data that is loaded in the primary or secondary memory of a computer is not a file. It is only when you save that data on the disk and name it suitably that the collection of data assumes the status of a file. Why use files? Read on. Primary memory is volatile; when you switch off the computer, everything that is stored in primary memory is lost. Therefore, it is necessary to save the data on the secondary storage (because the secondary storage is not volatile). It is also necessary to offer some suitable name to that collection of data so anyone can refer to that collection of data unambiguously. When you do this (i.e., save the collection of data on the secondary storage and name it), you get a file. A file is also called a *disk file* in order to distinguish it from a device file.

6-1. Read a Text File Character by Character

Problem

You want to read a text file character by character.

Solution

Write a C program that reads a text file character by character, with the following specifications:

- The program opens and reads an existing text file called `test.txt` that is stored in the default folder of `C:\Compiler`.
- The program uses the function `fgetc()` to read a character from a file and uses the function `putchar()` to display the character on the screen.

Create a small text file, named `test.txt`, with the following contents:

```
Welcome to C programming.
Thank you.
```

© Shirish Chavan 2017
S. Chavan, *C Recipes*, DOI 10.1007/978-1-4842-2967-5_6

Place the compiler in the folder C:\Compiler. Place the text file test.txt in this folder. This is the folder from which the compiler gets launched every time you start it because the main program file of the compiler rests in this folder.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder C:\Code with the file name files1.c:

```
/* This program reads the contents of the text file test.txt and displays */
/* these contents on the screen.  */
                                                                    /* BL */
#include <stdio.h>                                                  /* L1 */
                                                                    /* BL */
main()                                                              /* L2 */
{                                                                   /* L3 */
 int num;                                                           /* L4 */
 FILE *fptr;                                                        /* L5 */
 fptr = fopen("test.txt", "r");                                     /* L6 */
 num = fgetc(fptr);                                                 /* L7 */
                                                                    /* BL */
 while(num != EOF) {                                                /* L8 */
  putchar(num);                                                     /* L9 */
  num = fgetc(fptr);                                                /* L10 */
 }                                                                  /* L11 */
                                                                    /* BL  */
 fclose(fptr);                                                      /* L12 */
 return(0);                                                         /* L13 */
}                                                                   /* L14 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
Welcome to C programming.
Thank you.
```

How It Works

Now let's see how this program works. In LOC 4, an int variable called num is declared. In LOC 5, a pointer to the FILE variable fptr is declared. FILE is a derived type. To use this type (i.e., FILE) effectively, you are not required to know its composition or internal details. LOC 5 is reproduced here for your quick reference:

```
FILE *fptr;                                                          /* L5 */
```

After reading LOC 5, you should be expecting the following LOCs to follow LOC 5:

```
FILE var;                                                   /*  LOC A */
fptr = &var;                                                /*  LOC B */
```

In LOC A, a `FILE` variable named `var` is declared, and in LOC B, the address of `var` is assigned to `fptr`. After all, this is the generic procedure of using the pointers in C. Contrary to your expectations, instead of LOCs A and B, LOC 6 follows LOC 5, which is reproduced here for your quick reference:

```
fptr = fopen("test.txt", "r");                                /*  L6 */
```

By and large, LOC 6 performs, among other things, everything that LOCs A and B are supposed to perform. In LOC 6, a call is made to the function `fopen()`, which is used to “open” a file. Before using any file in a program, you are required to open it using the function `fopen()`.

The function `fopen()` creates an anonymous variable of type `FILE`, sets a pointer pointing to that variable, and then returns the pointer that is assigned to `fptr`. The function `fopen()` also creates a special pointer and sets it pointing to the first character in the file `test.txt`; this is not the usual pointer in C but just a “marker” that always points to the next character in the file to be read. To avoid confusion, I will call this special pointer a *marker*. When the first character in the file is read, the marker is automatically made to point to the second character in the file. When the second character in the file is read, the marker is automatically made to point to the third character in the file. And so on.

Also note that this “marker” is not an official term in C language. When the marker points to the first character in a file, then according to standard terminology in the C language, you can say that the file is positioned to the first character of the file. When the marker points to the second character in a file, then according to standard terminology in the C language, you say that the file is positioned to the second character of the file. And so on.

■ **Note** Whenever a file is opened using the function `fopen()`, the file is positioned to the first character of the file.

Notice that two arguments are passed to the function `fopen()`, and both these arguments are strings.

The first argument represents the file name: `"test.txt"`.

The second argument represents the mode: `"r"`.

The first argument represents the name of a file to be opened. The second argument represents the mode in which a file will be opened. The mode `"r"` indicates that this file will be opened for reading only. You just cannot modify the contents of this file. The generic syntax of a statement that uses the function `fopen()` is given here:

```
fptr = fopen(filename, "mode")
```

Here, fptr is the pointer to the FILE variable, the file name is an expression that evaluates to a string constant that consists of the name of a file (with or without the path) to be opened, and "mode" is a string constant that consists of one of the file-opening modes. The function fopen() creates an anonymous variable of type FILE, associates the file being opened with this variable, and then returns a pointer to FILE pointing to this anonymous variable, which, in turn, is assigned to the pointer variable fptr. If file opening fails (i.e., a file cannot be opened), then fopen() returns a NULL pointer. Also, once a file is opened, thereafter it is referred to using the pointer variable fptr. If a file name is devoid of a path, then it is assumed that the file rests in the default folder, which is C:\Compiler.

After the execution of LOC 6, the file test.txt opens successfully. Hereafter, you will not use the file name test.txt to refer to this file; instead, you will use the pointer variable fptr to refer to this file. Also, the marker is now set pointing to the first character in the file test.txt. This means the file is positioned to the first character of the file.

In LOC 7, the first character in the file test.txt is read, and its ASCII value is assigned to the int variable num. The contents of the file test.txt are reproduced here for your quick reference:

```
Welcome to C programming.
Thank you.
```

Notice that the first character in the file is 'W' and its ASCII value is 87. LOC 7 is reproduced here for your quick reference:

```
num = fgetc(fptr);                                                /*  L7 */
```

LOC 7 consists of a call to the function fgetc(). This function reads a character from the file represented by fptr and returns its ASCII value, which, in turn, is assigned to the int variable num. As the marker is pointing to the first character in the file (it is 'W'), fgetc() reads it and returns its ASCII value (it is 87), which, in turn, is assigned to num. Also, the marker is now advanced so as to point to the second character in the file, i.e., 'e'. All this takes place as LOC 7 executes.

The generic syntax of a statement that uses the function fgetc() is given here:

```
intN = fgetc(fptr);
```

Here, intN is an int variable, and fptr is a pointer to the FILE variable. The function fgetc() reads the character pointed to by the marker, from the file specified by fptr. After reading the character, fgetc() returns its ASCII value, which is assigned to the int variable intN and sets the marker pointing to the next character. If fgetc() encounters an end-of-file-character (which is the character ^Z whose ASCII value is 26, pronounced as "Control-Z"), then instead of returning its ASCII value 26, it returns the value of the symbolic constant EOF, which is the int value -1. EOF stands for "end of file." However, apart from the end-of-file situation, this value is also returned by some functions when an error occurs. It is a symbolic constant defined in the file <stdio.h> as follows:

```
#define EOF (-1)                                    /* End of file indicator */
```

EOF represents an int value -1. Do not think that the value of EOF is stored at the end of file. Actually, it is not. Character ^Z is stored at the end of file to mark the end of a text file.

You must have noticed that the function fgetc() works like the function getchar() that you use to read a character from the keyboard. However, the function getchar() doesn't expect any argument, whereas the function fgetc() expects a pointer to the FILE variable as an argument.

C also offers the function getc() that is identical to the function fgetc(); the only difference is that the function getc() is implemented as a macro, whereas the function fgetc() is implemented as a function.

LOCs 8 to 11 consist of a while loop. The rest of the file is read in this loop. LOC 8 consists of a continuation condition of a loop, and it is reproduced here for your quick reference:

```
while(num != EOF) {                                          /* L8 */
```

You can read the expression in parentheses (which represents a continuation condition of a loop) because while num is not equal to EOF, looping is permitted. In other words, LOC 8 tells you that looping is permitted as long as the value of num is not equal to -1. However, looping terminates once num becomes equal to -1. For now, the value of num is 87; hence, looping is permitted. Now the first iteration begins. The body of this while loop consists of only LOCs 9 and 10. Notice LOC 9, which is reproduced here for your quick reference:

```
putchar(num);                                                /* L9 */
```

The function putchar() converts the int value 87 stored in num to the corresponding char constant 'W' and sends this char constant to the screen for display. LOC 10 is same as LOC 7 and is reproduced here for your quick reference:

```
num = fgetc(fptr);                                          /* L10 */
```

I have already discussed how LOC 7 works. Before the execution of LOC 10, the marker was pointing to the second character in the file, i.e., 'e'. After the execution of LOC 10, the ASCII value of 'e' (which is 101) is assigned to num, and the marker is advanced so as to point to the next (third) character in the file.

LOCs 9 and 10 are executed repeatedly as many times as there are characters in a file. This loop performs 37 iterations as there are 26 characters in the first line and 11 characters in the second line, including the newline character at the end of each line. Consider the 37th iteration of the loop. In LOC 9, the character constant newline (ASCII value 10) is sent to the screen for display. This is the last character in the second line and also the last useful (useful from the point of view of the user of the file) character in the file. In LOC 10, the end-of-file character ^Z is read by fgetc(); however, its ASCII value 26 is not returned by fgetc(). Instead, a special value -1 (the value of the symbolic constant EOF) is returned by fgetc(), and it is assigned to num. At the beginning of the 38th iteration, the continuation condition (num != EOF) turns out to be false, and iteration is not permitted. Then the next LOC, which follows the while loop, is executed. This next LOC is LOC 12, which is reproduced here for your quick reference:

```
fclose(fptr);                                               /* L12 */
```

The function `fclose()` is used to close a file. It is advisable that an opened file should be closed when it is no longer needed in a program. Think of the function `fclose()` as a counterpart of the function `fopen()`. The function `fclose()` accepts only one argument; this argument is a pointer to the `FILE` variable and then closes the file specified by that pointer to the `FILE` variable. It returns the value `0` if the operation (of closing the file) is successful and returns the value `EOF` (i.e., `-1`) if the operation fails. After the execution of LOC 12, the file specified by `fptr` (which is `test.txt`) is closed. The generic syntax of a statement that uses the function `fclose()` is given here:

```
intN = fclose(pointer_to_FILE_variable);
```

Here, `pointer_to_FILE_variable` is a pointer to the `FILE` variable, and `intN` is an `int` type variable. The value returned by `fclose()`, which is either `0` (if the operation is successful) or `-1` (if the operation fails), is stored in `intN`.

The program terminates after the execution of LOC 12.

Finally, notice that even if you don't close a file, it is closed automatically when the program is terminated. However, it is advisable to close a file when it is no longer needed in a program because of these reasons:

- There is an upper limit on the number of files that can remain open at a time.
- When a file is closed, some housekeeping actions are performed by the programming environment that are badly needed.
- Some memory is freed.

6-2. Handle Errors When File Opening Fails

Problem

You want to handle the situation safely when file opening fails.

Solution

Write a C program that safely handles the situation when file opening fails, with the following specifications:

- The program opens the text file `satara.txt` that is placed in the folder `C:\Code`. The program checks whether the file opening is successful with the help of the function `feof()`. If file opening fails, then the program ensures a safe exit and avoids crashing.
- The program reads the file, displays its contents on the screen, and then closes the file. If file closing fails, the program reports it.

In the preceding recipe, I did not take into account the possibility that file opening (and therefore file closing) can fail. If file closing fails, then sometimes you can ignore it, because when a program is terminated, all open files are closed automatically. But if file opening fails, then the program will certainly not work as per your expectations. Therefore, it is absolutely necessary in a program to check whether file opening is successful. Also, it is advisable to check whether file closing is successful.

Create a small text file, named `satara.txt`, with the following contents:

```
Satara is surrounded by mountains.
Satara was capital of Maratha empire for many years.
```

Place this text file in the folder `C:\Code`.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files2.c`:

```
/* This program reads the contents of a text file and displays these
contents on the screen. */
/* File-opening and file-closing is checked for success. File is placed in the */
/* desired folder. Function feof() is used to detect the end of file. */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
 int num, k = 0;                                                   /* L4 */
 FILE *fptr;                                                       /* L5 */
 fptr = fopen("C:\\Code\\satara.txt", "r");                        /* L6 */
 if (fptr != NULL) {                                               /* L7 */
  puts("File satara.txt is opened successfully");                  /* L8 */
  puts("Contents of file satara.txt:");                            /* L9 */
  num = fgetc(fptr);                                               /* L10 */
                                                                   /* BL  */
  while(!feof(fptr)) {                                             /* L11 */
    putchar(num);                                                  /* L12 */
    num = fgetc(fptr);                                             /* L13 */
  }                                                                /* l14 */
                                                                   /* BL  */
  k = fclose(fptr);                                                /* L15 */
  if(k == -1)                                                      /* L16 */
    puts("File-closing failed");                                   /* L17 */
  else                                                             /* L18 */
    puts("File satara.txt is closed successfully");                /* L19 */
 }                                                                 /* L20 */
 else                                                              /* L21 */
  puts("File-opening failed");                                     /* L22 */
 return(0);                                                        /* L23 */
}                                                                  /* L24 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File satara.txt is opened successfully
Contents of file satara.txt:
Satara is surrounded by mountains.
Satara was capital of Maratha empire for many years.
File satara.txt is closed successfully
```

How It Works

`EOF` is a value (-1) that is returned when the end of file occurs. `EOF` stands for "end of file." In addition to an end-of-file situation, this value is also returned by some functions when an error occurs. To differentiate between these two causes of a returned `EOF`, two functions are available in C: `feof()` and `ferror()`. The generic syntax of a statement that uses the function `feof()` is given here:

```
intN = feof(fptr);
```

Here, `intN` is an `int` variable, and `fptr` is a pointer to the `FILE` variable. This function returns a nonzero (true) value when the end of file has occurred on the file specified by `fptr`; otherwise, it returns a zero (false) value.

Also, the generic syntax of a statement that uses the function `ferror()` is given here:

```
intN = ferror(fptr);
```

Here, `intN` is an `int` variable, and `fptr` is a pointer to `FILE` variable. This function returns a nonzero (true) value if an error has occurred on the file specified by `fptr`; otherwise, it returns a zero (false) value.

Notice LOC 6, which is reproduced here for your quick reference:

```
fptr = fopen("C:\\Code\\satara.txt", "r");                          /* L6 */
```

The name of file to be opened, with the path, is given here:

```
C:\Code\satara.txt
```

In LOC 6, instead of a single backslash, a double backslash is used in the string file name because it (the double backslash) is an escape sequence.

Notice LOC 7, which is reproduced here for your quick reference:

```
if (fptr != NULL) {                                                 /* L7 */
```

In this LOC, the value of `fptr` is checked. Only if the file opening is successful is the code block spanning LOCs 8 to 19 executed; otherwise, LOC 22 is executed, which displays the following message on the screen:

```
File-opening failed
```

In LOC 15, the value returned by `fclose()` is assigned to an `int` variable named `k`. If the file closing fails, then the value of -1 is assigned to the `int` variable `k`, and in that case the following message appears on the screen:

```
File-closing failed
```

However, failure messages did not appear in the output of this program because both operations (file opening and file closing) were successful.

Notice LOC 11, which is reproduced here for your quick reference:

```
while(!feof(fptr)) {                                                /* L11 */
```

The function `feof()` returns a nonzero (true) value when the end of file occurs; otherwise, it returns a zero (false) value. Notice the logical negation operator `!` prefixed to `feof()`. As a result, iterations are discontinued after the occurrence of the end of file. You can read LOC 11 as follows: while not the end of file, iterations are permitted.

6-3. Write to a Text File in Batch Mode

Problem

You want to write to a text file in a batch mode.

Solution

Write a C program that writes to a text file in batch mode, with the following specifications:

- The program writes to a file using the function `fputs()`.
- The program creates a text file, namely, `kolkata.txt`, in a folder called `C:\Code` and writes the following couple of lines to it:

  ```
  Kolkata is very big city.
  It is also very nice city.
  ```

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files3.c`:

```
/* This program creates a text file kolkata.txt using the function fputs(). */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
```

```
{                                                              /* L3 */
 int k = 0;                                                    /* L4 */
 FILE *fptr;                                                   /* L5 */
 fptr = fopen("C:\\Code\\kolkata.txt", "w");                   /* L6 */
 if (fptr != NULL) {                                           /* L7 */
  puts("File kolkata.txt is opened successfully.");            /* L8 */
  fputs("Kolkata is very big city.\n", fptr);                  /* L9 */
  fputs("It is also very nice city.\n", fptr);                 /* L10 */
  k = fclose(fptr);                                            /* L11 */
  if(k == -1)                                                  /* L12 */
    puts("File-closing failed");                               /* L13 */
  if(k == 0)                                                   /* L14 */
    puts("File is closed successfully.");                      /* L15 */
 }                                                             /* L16 */
 else                                                          /* L17 */
  puts("File-opening failed");                                 /* L18 */
 return(0);                                                    /* L19 */
}                                                              /* L20 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File kolkata.txt is opened successfully.
File is closed successfully.
```

Open the file `kolkata.txt`, just created, in a suitable text editor and verify that its contents are as expected.

How It Works

Notice LOC 6, which is reproduced here for your quick reference:

```
fptr = fopen("C:\\Code\\kolkata.txt", "w");                     /* L6 */
```

Here, the file name is `"C:\\Code\\kolkata.txt"`, and the mode is `"w"`. As you are going to write to a file, file-opening mode must be `"w"`. This means a file named `kolkata.txt` will be opened for writing in the specified folder `C:\Code`. Notice LOCs 9 and 10, which are reproduced here for your quick reference:

```
fputs("Kolkata is very big city.\n", fptr);                    /* L9 */
fputs("It is also very nice city.\n", fptr);                   /* L10 */
```

For the sake of clarity, I have used two statements; otherwise, a single statement is sufficient. Either LOC consists of a call to the function `fputs()`, which is used for writing a string to a file. How this function works closely resembles the function `puts()`, which is used to display a string on the screen. However, there are three main differences between how these functions work, as follows:

- The function `puts()` writes (displays) an argument string on the screen. The function `fputs()` writes an argument string to a file.
- The function `puts()` expects only one argument, and it is string. The function `fputs()` expects two arguments: a string and a pointer to the `FILE` variable.
- The function `puts()` replaces the string-terminating character `'\0'` in the string with the newline character `'\n'` before displaying that string on the screen. The function `fputs()` simply throws away the string-terminating character `'\0'` and writes the remaining string to the file.

Notice the generic syntax of a statement that uses the function `fputs()` given here:

```
intN = fputs(string, fptr);
```

Here, `intN` is an `int` variable, the string is an expression that evaluates to a string constant, and `fptr` is a pointer to the `FILE` variable; the string constant is written to a file specified by `fptr`. If the operation succeeds, then a nonnegative value is returned by this function; otherwise, `EOF` is returned. In LOCs 9 and 10, I have preferred to ignore the value returned by this function. However, in a professional program, you should catch the returned value and see whether the operation is successful. Write errors are common while writing to disk.

In LOC 9, the string `"Kolkata is very big city.\n"` is written to a file specified by `fptr` (i.e., `kolkata.txt`).

■ **Note** When you read a file, the marker is advanced accordingly so that it always points to the next character to be read. Similarly, when you write to a file, the marker is advanced accordingly so that it always points to the location in the file where the next character will be written.

Before the execution of LOC 9, the file positions to the first character of the file. The string `"Kolkata is very big city.\n"` consists of 26 characters. After the execution of LOC 9, the file positions to the 27th character of the file.

In LOC 10, the string `"It is also very nice city.\n"` is written to the file. This string consists of 27 characters. Therefore, after the execution of LOC 10, the file positions to the 54th character of the file.

Finally, notice that when a string is written to a file using the function `fputs()`, then the character `'\0'`, which is a string-terminating character, is not written to the file, nor it is replaced by `'\n'` as in the case of `puts()`.

6-4. Write to a Text File in Interactive Mode

Problem

You want to write to a text file in interactive mode.

Solution

Write a C program that writes to a text file in interactive mode, with the following specifications:

- The program writes to a file using the function `fputs()`.
- The program creates the text files in an interactive manner. It accepts the name and text of the file in an interactive manner.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files4.c`:

```
/* This program creates a text file in an interactive session
using the function fputs(). */
                                                                 /* BL */
#include <stdio.h>                                               /* L1 */
#include <string.h>                                              /* L2 */
                                                                 /* BL */
main()                                                           /* L3 */
{                                                                /* L4 */
 int k = 0, n = 0;                                               /* L5 */
 char filename[40], temp[15], store[80];                         /* L6 */
 FILE *fptr;                                                     /* L7 */
 printf("Enter filename (AAAAAAA.AAA) extension optional: ");    /* L8 */
 scanf("%s", temp);                                              /* L9 */
 strcpy(filename, "C:\\Code\\");                                 /* L10 */
 strcat(filename, temp);                                         /* L11 */
 fptr = fopen(filename, "w");                                    /* L12 */
 if (fptr != NULL) {                                             /* L13 */
  printf("File %s is opened successfully.\n", filename);         /* L14 */
  puts("Enter the lines of text to store in the file.");         /* L15 */
  puts("Strike Enter key twice to end the line-entry-session."); /* L16 */
  fflush(stdin);                                                 /* L17 */
  gets(store);                                                   /* L18 */
  n = strlen(store);                                             /* L19 */
                                                                 /* BL  */
  while(n != 0){                                                 /* L20 */
   fputs(store, fptr);                                           /* L21 */
   fputs("\n", fptr);                                            /* L22 */
   gets(store);                                                     /* L23 */
   n = strlen(store);                                               /* L24 */
  }                                                                 /* L25 */
                                                                    /* BL  */
  k = fclose(fptr);                                                 /* L26 */
  if(k == -1)                                                       /* L27 */
    puts("File-closing failed");                                    /* L28 */
  if(k == 0)                                                        /* L29 */
    puts("File is closed successfully.");                           /* L30 */
 }                                                                  /* L31 */
 else                                                               /* L32 */
  puts("File-opening failed");                                      /* L33 */
 return(0);                                                         /* L34 */
}                                                                   /* L35 */
```

Compile and execute this program. A couple of runs of this program are given here. Here is the first run:

```
Enter filename (AAAAAAA.AAA) extension optional: Mumbai.txt   ↵
File C:\Code\Mumbai.txt is opened successfully.
Enter the lines of text to store in the file.
Strike Enter key twice to end the line-entry-session.
Mumbai is capital of Maharashtra.    ↵
Mumbai is financial capital of India.    ↵
↵
File is closed successfully.
```

Here is the second run:

```
Enter filename (AAAAAAA.AAA) extension optional: wai    ↵
File C:\Code\wai is opened successfully.
Enter the lines of text to store in the file.
Strike Enter key twice to end the line-entry-session.
Wai is a small town in Satara district.    ↵
There are good number of temples in Wai.    ↵
↵
File is closed successfully.
```

How It Works

Now let's discuss how this program works with reference to the second run. In LOC 6, three char arrays are declared, namely, `filename`, `temp`, and `store`. The file name you entered (`"wai"`) is stored in the array `temp`, to which the path is added, and then the file name with the path `"C:\\Code\\wai"` is stored in the array file name. Notice LOC 11, which is reproduced here for your quick reference:

```
strcat(filename, temp);                                            /* L11 */
```

In LOC 11, the function `strcat()` is called. This function is used for concatenating the strings. Before the execution of LOC 11, `filename` and `temp` contains the following strings:

- File name: `"C:\\Code\\"`
- Temp: `"wai"`

After the execution of LOC 11, `filename` and `temp` contains the following strings:

File name: `"C:\\Code\\wai"`
Temp: `"wai"`

Notice that the string stored in `temp` is appended to the string stored in the file name. The generic syntax of a statement that uses the function `strcat()` is given here:

```
storage = strcat(destination, source);
```

Here, `storage` is a pointer-to-`char` variable, the destination is a `char` array (or pointer-to-`char` variable), and the source is an expression that evaluates to a string constant. The string constant in source is appended to the string constant in the destination, and a copy of the resulting string constant is stored in the destination and returned. The returned value is generally ignored, and I have chosen to ignore it in LOC 11.

Next, notice LOC 17, which is reproduced here for your quick reference:

```
fflush(stdin);                                              /* L17 */
```

The function `fflush()` is used to flush out the stray characters loitering in the passage between the keyboard and the central processing unit (technically speaking, in the input buffer). You enter the file name `wai` and then press the Enter key. The file name entered is read and assigned to `temp` by the programming environment, but the Enter key stroke (i.e., newline character) remains in the passage (between the keyboard and the CPU), and it needs to be flushed out. This flushing is performed by the function `fflush()` in LOC 17.

Next, notice LOCs 18 and 19, which are reproduced here for your quick reference:

```
gets(store);                                                /* L18 */
n = strlen(store);                                          /* L19 */
```

The function `gets()` in LOC 18 reads the first string typed, which is `"Wai is a small town in Satara district."`, and places it in the `char` array `store`. In LOC 19, the function `strlen()` computes and returns the length of this string (which is 39), which is assigned to the `int` variable `n`.

Next, the `while` loop begins. Notice LOC 20, the first LOC of the `while` loop, which is reproduced here for your quick reference:

```
while(n != 0) {                                             /* L20 */
```

Notice the continuation condition of the `while` loop. Looping is permitted only if `n` is not equal to zero. Now the value of `n` is 39; hence, iteration is permitted. The first iteration begins. Notice LOCs 21 and 22, which are reproduced here for your quick reference:

```
fputs(store, fptr);                                         /* L21 */
fputs("\n", fptr);                                          /* L22 */
```

Both LOCs call the function `fputs()` that, in turn, writes an argument string to the file specified by the pointer to the `FILE` variable `fptr`. LOC 21 writes the string stored in the store to the file specified by `fptr`. LOC 22 writes the string `"\n"` to the file specified by `fptr`. In LOC 22, you are appending the newline character to the string manually. If you don't do this, then retrieving the strings (from a file) using the function `fgets()` becomes difficult as the function `fgets()` assumes that strings stored in a file are terminated with newline characters.

Next, LOCs 23 and 24 are executed, which are the same as LOCs 18 and 19. In LOC 23, the second string typed, which is `"There are good number of temples in Wai."`, is read by the function `gets()` and is stored in the store. In LOC 24, its length is computed, which is 40, and is returned by the function `strlen()`, which in turn is assigned to the `int` variable `n`.

As execution of the first iteration is complete, computer control goes to LOC 20, the first LOC of the `while` loop. As the value of `n` is 40 and not zero, the second iteration is permitted. In LOC 21, the string stored in store, which is `"There are good number of temples in Wai."`, is written to the file specified by `fptr`. In LOC 22, the newline character is written to the file specified by `fptr`.

Next, LOCs 23 and 24 are executed. In LOC 23, the third string typed, which is the null string because you pressed the Enter key at the beginning of the line, is read by the function `fgets()`. In LOC 24, the length of the null string, which is the zero, is returned by the function `strlen()`, which in turn is assigned to the `int` variable `n`.

As execution of the second iteration is complete, computer control goes to LOC 20, the first LOC of the `while` loop. As the value of `n` is zero, the third iteration is not permitted. Computer control then goes to LOC 26, in which the file specified by `fptr` is closed.

6-5. Read a Text File String by String

Problem

You want to read a text file string by string.

Solution

Write a C program that reads a text file string by string, with the following specifications:

- The program reads the text file `kolkata.txt` using the function `fgets()` and then displays the text in the file on the screen using the function `printf()`.
- The program checks for successful file opening and file closing.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files5.c`:

```
/* This program reads the file kolkata.txt using the function fgets(). */
                                                                    /* BL */
#include <stdio.h>                                                  /* L1 */
                                                                    /* BL */
```

```
main()                                                     /* L2 */
{                                                          /* L3 */
 int k = 0;                                                /* L4 */
 char *cptr;                                               /* L5 */
 char store[80];                                           /* L6 */
 FILE *fptr;                                               /* L7 */
 fptr = fopen("C:\\Code\\kolkata.txt", "r");               /* L8 */
 if (fptr != NULL) {                                       /* L9 */
  puts("File kolkata.txt is opened successfully.");        /* L10 */
  puts("Contents of this file:");                          /* L11 */
  cptr = fgets(store, 80, fptr);                           /* L12 */
                                                           /* BL  */
  while (cptr != NULL) {                                   /* L13 */
   printf("%s", store);                                    /* L14 */
   cptr = fgets(store, 80, fptr);                          /* L15 */
  }                                                        /* L16 */
                                                           /* BL  */
  k = fclose(fptr);                                        /* L17 */
  if(k == -1)                                              /* L18 */
    puts("File-closing failed");                           /* L19 */
  if(k == 0)                                               /* L20 */
    puts("\nFile is closed successfully.");                /* L21 */
 }                                                         /* L22 */
 else                                                      /* L23 */
  puts("File-opening failed");                             /* L24 */
 return(0);                                                /* L25 */
}                                                          /* L26 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File kolkata.txt is opened successfully.
Contents of this file:
Kolkata is very big city.
It is also very nice city.
File is closed successfully.
```

How It Works

Think of the function `fgets()` as the counterpart of the function `fputs()`. The function `fgets()` is used for reading the strings from a file. The generic syntax of a statement that uses the function `fgets()` is given here:

```
cptr = fgets(storage, n, fptr);
```

Here, `cptr` is a pointer-to-`char` variable, `storage` is a `char` type array, `n` is an expression that evaluates to an integer constant, and `fptr` is a pointer to the `FILE` variable. The function `fgets()` reads a string from the file specified by `fptr` and stores that string in the array storage. At most (n - 1) characters are read by fgets(), and the string stored in storage is always terminated with '\0'. Then the function fgets() returns a pointer to char pointing to the first character in array storage when the string-reading operation is successful; otherwise, it returns NULL. Also, notice that the function fgets() reads a string starting at the current location (pointed to by the marker), up to and including the first newline character it encounters, unless it reaches an EOF or has read (n - 1) characters before that point. It then appends the '\0' character (string-terminating character) to that string before storing it in storage.

In LOC 5, a pointer-to-char variable called cptr is declared. In LOC 6, the char array store is declared. In LOC 8, the file kolkata.txt is opened for reading, and it is associated with a pointer to the FILE variable fptr. Notice LOC 12, which is reproduced here for your quick reference:

```
cptr = fgets(store, 80, fptr);                                   /* L12 */
```

In LOC 12, the following tasks are performed:

- The first string stored in the file specified by fptr is read, and it is placed in the char array called store. Notice that this string is "Kolkata is very big city."
- The marker is advanced so as to point to the next string available in the file specified by fptr.
- The function fgets() returns a pointer to char pointing to the first character in the char array store, which is assigned to cptr. You need this value only to detect the end of file. When the end of file occurs, fgets() returns NULL.

The second argument to fgets() in LOC 12 is the integer value 80. It indicates that fgets() will read at most 79 characters when called (80 - 1) . This means if a string stored in a file consists of more than 79 characters, then the remaining characters will not be read. In the next call to fgets(), it will start reading the next string.

Next, there is a while loop in LOCs 13 to 16, which is reproduced here for your quick reference:

```
while (cptr != NULL) {                                           /* L13 */
 printf("%s", store);                                            /* L14 */
 cptr = fgets(store, 80, fptr);                                  /* L15 */
}                                                                /* L16 */
```

In LOC 13, the value stored in cptr is tested for its equivalence with NULL in order to detect the occurrence of the end of file. In LOC 14, the string stored in the char array store (which is "Kolkata is very big city.") is displayed on the screen. Notice that this is the first iteration of the while loop. In LOC 15 (which is the same as LOC 12), the next string in the file (which is "It is also very nice city.") is read and stored in the char array store. Also, the pointer to the first character of store is returned, which is assigned to cptr. As execution of first iteration is complete, computer control goes to LOC 13 again.

Next, LOC 13 is executed. No NULL value is still assigned to cptr; hence, the second iteration is permitted. Next, LOC 14 is executed in which the string stored in store (which is "It is also very nice city.") is displayed on the screen. Next, LOC 15 is executed in which fgets() tries to read the next string (third string) in the file. But as this file (kolkata.txt) consists of only two strings, this reading operation fails, and fgets() returns a NULL value, which is assigned to cptr. As the execution of the second iteration is complete, computer control goes to LOC 13 again.

Next, LOC 13 is executed. As a NULL value is assigned to cptr, the third iteration is not permitted, and execution of the loop terminates. Next, computer control passes to LOC 17 in which the file specified by fptr is closed.

6-6. Write to a Text File Character by Character

Problem

You want to write to a text file character by character.

Solution

Write a C program that writes to a text file character by character, with the following specifications:

- The program opens the text file jaipur.txt in write mode.
- The program writes to this file in interactive mode using the function fputc().

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder C:\Code with the file name files6.c:

```
/* This program creates a text file in an interactive session using the
function fputc(). */
                                                                  /* BL */
#include <stdio.h>                                                /* L1 */
                                                                  /* BL */
main()                                                            /* L2 */
{                                                                 /* L3 */
 int k = 0, n = 0;                                                /* L4 */
 FILE *fptr;                                                      /* L5 */
 fptr = fopen("C:\\Code\\jaipur.txt", "w");                       /* L6 */
 if (fptr != NULL) {                                              /* L7 */
  puts("File jaipur.txt is opened successfully.");                /* L8 */
  puts("Enter text to be written to file. Enter * to");           /* L9 */
  puts("terminate the text-entry-session.");                      /* L10 */
  n = getchar();                                                  /* L11 */
                                                                  /* BL  */
  while(n != '*'){                                                 /* L12 */
   fputc(n, fptr);                                                 /* L13 */
   n = getchar();                                                  /* L14 */
  }                                                                /* L15 */
                                                                   /* BL  */
  k = fclose(fptr);                                                /* L16 */
  if(k == -1)                                                      /* L17 */
    puts("File-closing failed");                                   /* L18 */
  if(k == 0)                                                       /* L19 */
    puts("File is closed successfully.");                          /* L20 */
 }                                                                 /* L21 */
 else                                                              /* L22 */
  puts("File-opening failed");                                     /* L23 */
 return(0);                                                        /* L24 */
}                                                                  /* L25 */
```

Compile and execute this program. A run of this program is given here:

```
File jaipur.txt is opened successfully.
Enter text to be written to file. Enter * to
terminate the text-entry-session.
Jaipur is capital of Rajsthan.   ↵
Jaipur is famous for historical Hawamahal.   ↵
*   ↵
File is closed successfully.
```

How It Works

You have typed two lines of text to be written to the file `jaipur.txt`. When you type a character, it is not processed by the CPU. Characters typed stand in a queue. It is only when you press the Enter key that these characters (standing in a queue) are processed by the CPU, one by one. First, type the following line of text and press Enter:

```
Jaipur is capital of Rajsthan.
```

Notice LOC 11, which is reproduced here for your quick reference:

```
n = getchar();                                                     /* L11 */
```

The function `getchar()` reads the first character in the line of text, which is `'J'`, and returns its ASCII value (it is 74), which is assigned to the `int` variable `n`.

Next, the execution of the `while` loop begins, which spans LOCs 12 to 15. LOC 12 is reproduced here for your quick reference:

```
while(n != '*') {                                                  /* L12 */
```

Notice the continuation condition of the while loop in LOC 12. This loop iterates as long as n is not equal to character '*'. (Or more correctly, it iterates as long as n is not equal to 64, the ASCII value of '*'. As n is equal to 74 and not equal to 64, iteration is permitted.) Now the first iteration begins.

Next, LOC 13 is executed, which is reproduced here for your quick reference:

```
fputc(n, fptr);                                              /* L13 */
```

The function fputc() writes the character, whose ASCII value is stored in the int variable n, in the file specified by fptr. As the ASCII value of 'J' is stored in n, the character 'J' is written to the file specified by fptr (i.e., to the file jaipur.txt).

The generic syntax of a statement that uses the function fputc() is given here:

```
intN = fputc (n, fptr);
```

Here, intN is an int variable, n is an expression that evaluates to an integer value, and fptr is a pointer to the FILE variable. The function fputc() writes the character, whose ASCII value is n, to the file specified by fptr. The function fputc() returns the value of EOF if the operation fails and returns the value of n if operation is successful.

C also offers the function putc(), which is identical to the function fputc(); the only difference is that the function putc() is implemented as a macro, whereas the function fputc() is implemented as a function.

Next, LOC 14 is executed, which is same as LOC 11. In LOC 14, the next character in the line of text is read, which is 'a', and its ASCII value (97) is assigned to variable n. As execution of the first iteration is complete, computer control goes to LOC 12 again. As the value of n is not equal to 64 (ASCII value of *), the second iteration is permitted, and so on. In this manner, the characters in the line of text are written to the file. When the character is *, then the continuation condition in LOC 12 fails, and further iterations are not permitted.

In this program, you use the character * as a terminating character. What if * is part of the text? Ideally, the terminating character should not be a printable character. You can use ^Z (Control-Z) as a terminating character, which can be passed to the program simply by pressing the function key F6.

6-7. Write Integers to a Text File

Problem

You want to write the integers to a text file.

Solution

Write a C program that writes the integers to the text file numbers.dat, with the following specifications:

- The program uses the function fprintf() to write to the file.
- The program writes integer values to the file.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files7.c`:

```
/* This program writes data to a file using the function fprintf() */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
 int i, k = 0;                                                     /* L4 */
 FILE *fptr;                                                       /* L5 */
 fptr = fopen("C:\\Code\\numbers.dat", "w");                       /* L6 */
 if (fptr != NULL) {                                               /* L7 */
  puts("File numbers.dat is opened successfully.");                /* L8 */
                                                                   /* BL */
  for(i = 0; i < 10; i++)                                          /* L9 */
    fprintf(fptr, "%d ", i+1);                                     /* L10 */
                                                                   /* BL  */
  puts("Data written to file numbers.dat successfully.");          /* L11 */
                                                                   /* BL  */
  k = fclose(fptr);                                                /* L12 */
  if(k == -1)                                                      /* L13 */
    puts("File-closing failed");                                   /* L14 */
  if(k == 0)                                                       /* L15 */
    puts("File is closed successfully.");                          /* L16 */
 }                                                                 /* L17 */
 else                                                              /* L18 */
  puts("File-opening failed");                                     /* L19 */
 return(0);                                                        /* L20 */
}                                                                  /* L21 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File numbers.dat is opened successfully.
Data written to file numbers.dat successfully.
File is closed successfully.
```

Open the file `numbers.dat` in a suitable text editor and verify that its contents are as follows:

```
1  2  3  4  5  6  7  8  9  10
```

How It Works

In the preceding recipes you used the functions `fputc()` and `fputs()` to write the characters and strings to a file, respectively. But if you want to write the data of other data types (e.g., `int`, `float`, etc.) to a file, then you need to use the function `fprintf()`.

■ **Note** Using the function `fprintf()`, you can write the data items of different data types to a file in a single statement. Also, `fprintf()` can be used to write the formatted data to a file. `fprintf()` writes the data to a file in text format.

LOCs 9 and 10 consist of a `for` loop, and in this loop data is written to the file `numbers.dat`. These LOCs are reproduced here for your quick reference:

```
for(i = 0; i < 10; i++)                                          /* L9 */
  fprintf(fptr, "%d ", i+1);                                     /* L10 */
```

This `for` loop iterates ten times and writes ten numbers to the file, one number per iteration. The numbers are actually written to the file in LOC 10, in which a call is made to the function `fprintf()`. You should also note that the `fprintf()` function always writes the data to a file in character format (or text format).

The function `fprintf()` works like the function `printf()`. There are two main differences, however, as follows:

- The function `printf()` sends the data to the screen for display, whereas the function `fprintf()` sends the data to the file for writing it in that file.
- Like the function `printf()`, the function `fprintf()` also expects the control string and a comma-separated list of arguments. However, the function `fprintf()` expects one more argument compared to the function `printf()`. This extra argument is a pointer to the `FILE` variable (i.e., `fptr`), and this must be a first argument.

Before proceeding, notice the generic syntax of a statement that uses the function `fprintf()` given here:

```
intN = fprintf(fptr, "control string", arg1, arg2, ..., argN);
```

Here, `intN` is an `int` variable, `fptr` is a pointer to the `FILE` variable, `"control string"` is a control string that appears in the `printf()` function, and `arg1`, `arg2`, ..., `argN` is a comma-separated list that appears in the `printf()` function. The values of arguments are inserted in the control string (to replace the corresponding conversion specifications), and the resulting string is written to the file specified by `fptr`. The function `fprintf()` returns an `int` value that indicates the number of characters written to the file if operation is successful; otherwise, it returns `EOF`. For example, except for the tenth iteration of the `for` loop, the `fprintf()` function in LOC 10 returns 2 (one digit + one space), and in the tenth iteration it returns 3 (two digits + one space).

6-8. Write Structures to a Text File

Problem

You want to write the structures to a text file.

Solution

Write a C program that writes the structures to the text file `agents.dat`, with the following specifications:

- The program opens the file `agents.dat` in write mode.
- The program accepts the data for structures (shown in Figure 6-1) in interactive mode.

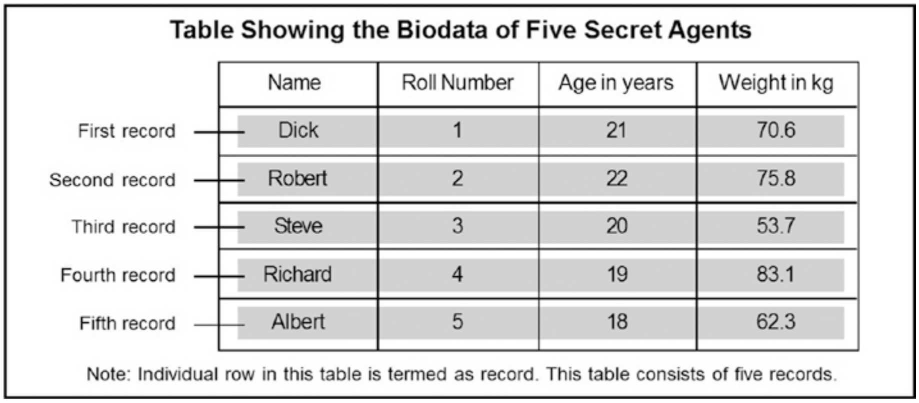

Table Showing the Biodata of Five Secret Agents

	Name	Roll Number	Age in years	Weight in kg
First record	Dick	1	21	70.6
Second record	Robert	2	22	75.8
Third record	Steve	3	20	53.7
Fourth record	Richard	4	19	83.1
Fifth record	Albert	5	18	62.3

Note: Individual row in this table is termed as record. This table consists of five records.

Figure 6-1. *Table showing the biodata of five secret agents*

- The program uses the function `fprintf()` to write the structures to the file.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files8.c`:

```
/* This program writes a structure to a file using the function fprintf() */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
```

```
 int k = 0;                                                         /* L4 */
 char flag = 'y';                                                   /* L5 */
 FILE *fptr;                                                        /* L6 */
 struct biodata{                                                    /* L7 */
  char name[15];                                                    /* L8 */
  int rollno;                                                       /* L9 */
  int age;                                                          /* L10 */
  float weight;                                                     /* L11 */
 };                                                                 /* L12 */
 struct biodata sa;                                                 /* L13 */
 fptr = fopen("C:\\Code\\agents.dat", "w");                         /* L14 */
 if (fptr != NULL) {                                                /* L15 */
  printf("File agents.dat is opened successfully.\n");              /* L16 */
                                                                    /* BL */
  while(flag == 'y'){                                               /* L17 */
   printf("Enter name, roll no, age, and weight of agent: ");       /* L18 */
   scanf("%s %d %d %f", sa.name,                                    /* L19 */
                        &sa.rollno,                                 /* L20 */
                        &sa.age,                                    /* L21 */
                        &sa.weight);                                /* L22 */
   fprintf(fptr, "%s %d %d %.1f", sa.name,                          /* L23 */
                                  sa.rollno,                        /* L24 */
                                  sa.age,                           /* L25 */
                                  sa.weight);                       /* L26 */
   fflush(stdin);                                                   /* L27 */
   printf("Any more records(y/n): ");                               /* L28 */
   scanf(" %c", &flag);                                             /* L29 */
  }                                                                 /* L30 */
                                                                    /* Bl  */
  k = fclose(fptr);                                                 /* L31 */
  if(k == -1)                                                       /* L32 */
    puts("File-closing failed");                                    /* L33 */
  if(k == 0)                                                        /* L34 */
    puts("File is closed successfully.");                           /* L35 */
 }                                                                  /* L36 */
 else                                                               /* L37 */
  puts("File-opening failed");                                      /* L38 */
 return(0);                                                         /* L39 */
}                                                                   /* L40 */
```

Compile and execute this program. A run of this program is given here:

```
File agents.dat is opened successfully.
Enter name, roll no, age, and weight of agent: Dick  1  21  70.6    ↵
Any more records (y/n): y    ↵
Enter name, roll no, age, and weight of agent: Robert  2  22  75.8    ↵
Any more records (y/n): y    ↵
```

```
Enter name, roll no, age, and weight of agent: Steve  3  20  53.7    ↵
Any more records (y/n): y    ↵
Enter name, roll no, age, and weight of agent: Richard  4  19  83.1    ↵
Any more records (y/n): y    ↵
Enter name, roll no, age, and weight of agent: Albert  5  18  62.3    ↵
Any more records (y/n): n    ↵
File is closed successfully.
```

Open the file `agents.dat` in a suitable text editor and verify that its contents are as shown here:

```
Dick 1 21 70.6Robert 2 22 75.8Steve 3 20 53.7Richard 4 19 83.1Albert 5 18 62.3
```

How It Works

In this recipe, you use the function `fprintf()` to write the biodata of five secret agents shown in Figure 6-1 to a file. LOCs 7 to 12 consist of the declaration of the structure `biodata`. In LOC 13, the variable `sa` of type `struct biodata` is declared. The data of a secret member typed in through the keyboard is saved in this variable before writing it to the file. LOCs 17 to 30 consist of a `while` loop in which the main activity (i.e., accepting the data typed through keyboard and writing it to the file) takes place. Notice the continuation condition of the loop in LOC 17, which is reproduced here for your quick reference:

```
while(flag == 'y'){                                                /* L17 */
```

Iterations are permitted while the value of the `char` variable flag is `'y'`. As the value of the `char` variable flag is already `'y'` (see LOC 5), the first iteration of the loop begins. LOCs 19 to 22 consist of a `scanf` statement. It is a single statement, but as the statement is long, it is split into four LOCs for better readability. The data of Dick typed (i.e., Dick 1 21 70.6) is read by this `scanf` statement and assigned to the variable `sa`.

LOCs 23 to 26 consist of the `fprintf` statement. Like the preceding statement, this is also a single statement but split into four LOCs for better readability. In this statement, the data stored in the variable `sa` is written to the file specified by `fptr`. In LOC 27, the function `fflush()` is called to flush out the newline character loitering in the passage between the keyboard and the CPU. In addition, a single space is prefixed to the conversion specification `%c` in LOC 29 to deal with this unwanted newline character. In fact, one of the provisions is enough to deal with this unwanted newline character:

- Provision of LOC 27
- Single space prefixed to `%c` in LOC 29

In LOC 29, the character typed (either `'y'` or `'n'`) as a reply to the question `"Any more records (y/n)"` is read and assigned to the `char` variable `flag`. If your reply is `'y'`, then further iterations of the `while` loop are permitted; otherwise, iterations are discontinued.

6-9. Read Integers Stored in a Text File

Problem

You want to read the integers stored in a text file.

Solution

Write a C program that reads the integers stored in a text file, with the following specifications:

- The program opens the file `numbers.dat` in reading mode. In this file, the integer values are already stored.
- The program reads the file `numbers.dat` using the function `fscanf()` and displays its contents on the screen.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files9.c`:

```
/* This program reads a file using the function fscanf() */
                                                                    /* BL */
#include <stdio.h>                                                  /* L1 */
                                                                    /* BL */
main()                                                              /* L2 */
{                                                                   /* L3 */
 int m = 0, n, k = 0;                                               /* L4 */
 FILE *fptr;                                                        /* L5 */
 fptr = fopen("C:\\Code\\numbers.dat", "r");                        /* L6 */
 if (fptr != NULL) {                                                /* L7 */
  puts("File numbers.dat is opened successfully.");                 /* L8 */
  puts("Contents of file numbers.dat:");                            /* L9 */
  m = fscanf(fptr, "%d", &n);                                       /* L10 */
                                                                    /* BL  */
  while(m != EOF){                                                  /* L11 */
   printf("%d ", n);                                                /* L12 */
   m = fscanf(fptr, "%d", &n);                                      /* L13 */
  }                                                                 /* L14 */
                                                                    /* BL  */
  printf("\n");                                                     /* L15 */
  k = fclose(fptr);                                                 /* L16 */
  if(k == -1)                                                       /* L17 */
    puts("File-closing failed");                                    /* L18 */
  if(k == 0)                                                        /* L19 */
```

```
    puts("File is closed successfully.");                              /* L20 */
 }                                                                     /* L21 */
 else                                                                  /* L22 */
  puts("File-opening failed");                                         /* L23 */
 return(0);                                                            /* L24 */
}                                                                      /* L25 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File numbers.dat is opened successfully.
Contents of file numbers.dat:
1 2 3 4 5 6 7 8 9 10
File is closed successfully.
```

How It Works

Think of the function `fscanf()` as the counterpart of the function `fprintf()`. How the function `fscanf()` works is analogous to that of the function `scanf()`. There are two main differences in how these functions work, however, as follows:

- The function `scanf()` reads the data coming from the keyboard, whereas the function `fscanf()` reads the data stored in a file.
- Like the function `scanf()`, the function `fscanf()` also expects the control string and a comma-separated list of arguments. However, the function `fscanf()` expects one more argument compared to the function `scanf()`. This extra argument is a pointer to the `FILE` variable (i.e., `fptr`), and this must be a first argument.

Before proceeding, notice the generic syntax of a statement that uses the function `fscanf()` given here:

```
intN = fscanf(fptr, "control string", arg1, arg2, ..., argN);
```

Here, `intN` is an `int` variable; `fptr` is a pointer to the `FILE` variable; `"control string"` is a control string that appears in the `scanf()` function; and `arg1`, `arg2`, …., `argN` is a comma-separated list of arguments that appears in the `scanf()` function. The values of the data items read from a file are assigned to respective arguments. This function returns an `int` value, which is the number of successful field conversions if the reading operation is successful; otherwise, it returns `EOF`.

Now let's discuss how this program works. Notice LOC 10, which is reproduced here, for your quick reference:

```
m = fscanf(fptr, "%d", &n);                                            /* L10 */
```

The function `fscanf()` in LOC 10 reads the first integer stored in the file specified by `fptr` and assigns this integer to the `int` variable `n`. The first integer stored in the file specified by `fptr` is 1, and this value is assigned to `n`. Also, one field conversion is successfully read; hence, value 1 is returned, which is assigned to the `int` variable `m`.

Next, there is a `while` loop spanning LOCs 11 to 14. LOC 11 contains the continuation condition of the `while` loop, which is reproduced here for your quick reference:

```
while(m != EOF){                                                      /* L11 */
```

As the value of `m` is now 1 and not `EOF`, iteration is permitted, and the first iteration begins. Next, LOC 12 is executed, which is reproduced here for your quick reference:

```
printf("%d ", n);                                                     /* L12 */
```

In LOC 12, the value stored in the `int` variable `n`, which is 1, is displayed on the screen. Next, LOC 13 is executed, which is the same as LOC 10. In LOC 13, the next `int` value stored in the file, which is 2, is read and assigned to the `int` variable `n`. Next, as the execution of the first iteration is complete, computer control goes to LOC 11. As the value of `m` is now 2 and not `EOF`, the second iteration begins, and so on. Proceeding in this manner, the complete file is read.

When all the `int` values stored in the file are read and `fscanf()` tries to read the next `int` value (which is nonexistent), then the reading operation fails, and the function `fscanf()` returns the `EOF` value, which is assigned to `m`, and then the iterations are terminated.

6-10. Read Structures Stored in a Text File

Problem

You want to read the structures stored in a text file.

Solution

Write a C program that reads the structures stored in a text file, with the following specifications:

- The program opens the file `agents.dat` in reading mode. This file consists of the structures and biodata of five secret agents.
- The program reads the file `agents.dat` using the function `fscanf()` and displays its contents on the screen.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files10.c`:

```
/* This program reads the records stored in a file using the function
fscanf(). */
                                                                     /* BL */
#include <stdio.h>                                                   /* L1 */
                                                                     /* BL */
```

```
main()                                                          /* L2 */
{                                                               /* L3 */
 int k = 0, m = 0;                                              /* L4 */
 FILE *fptr;                                                    /* L6 */
 struct biodata{                                                /* L7 */
  char name[15];                                                /* L8 */
  int rollno;                                                   /* L9 */
  int age;                                                      /* L10 */
  float weight;                                                 /* L11 */
 };                                                             /* L12 */
 struct biodata sa;                                             /* L13 */
 fptr = fopen("C:\\Code\\agents.dat", "r");                     /* L14 */
 if (fptr != NULL) {                                            /* L15 */
  printf("File agents.dat is opened successfully.\n");          /* L16 */
  m = fscanf(fptr, "%s %d %d %f", sa.name,                      /* L19 */
                                          &sa.rollno,           /* L20 */
                                          &sa.age,              /* L21 */
                                          &sa.weight);          /* L22 */
                                                                /* BL  */
  while(m != EOF){                                              /* L23 */
   printf("Name: %s, Roll no: %d, Age: %d, Weight: %.1f\n",     /* L24 */
                sa.name, sa.rollno,sa.age, sa.weight);          /* L25 */
   m = fscanf(fptr, "%s %d %d %f", sa.name,                     /* L26 */
                                           &sa.rollno,          /* L27 */
                                           &sa.age,             /* L28 */
                                           &sa.weight);         /* L29 */
  }                                                             /* L30 */
                                                                /* BL  */
  k = fclose(fptr);                                             /* L31 */
  if(k == -1)                                                   /* L32 */
    puts("File-closing failed");                                /* L33 */
  if(k == 0)                                                    /* L34 */
    puts("File is closed successfully.");                       /* L35 */
 }                                                              /* L36 */
 else                                                           /* L37 */
  puts("File-opening failed");                                  /* L38 */
 return(0);                                                     /* L39 */
}                                                               /* L40 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File agents.dat is opened successfully.
Name: Dick, Roll no: 1, Age: 21, Weight: 70.6
Name: Robert, Roll no: 2, Age: 22, Weight: 75.8
Name: Steve, Roll no: 3, Age: 20, Weight: 53.7
Name: Richard, Roll no: 4, Age: 19, Weight: 83.1
Name: Albert, Roll no: 5, Age: 18, Weight: 62.3
File is closed successfully.
```

How It Works

Now let's discuss how this program works. Notice LOCs 19 to 22, which contain a single `fscanf` statement, but because the statement is long, it is split into four LOCs for better readability. These LOCs are reproduced here for your quick reference:

```
m = fscanf(fptr, "%s %d %d %f", sa.name,                          /* L19 */
                                       &sa.rollno,                /* L20 */
                                       &sa.age,                   /* L21 */
                                       &sa.weight);               /* L22 */
```

This statement, after execution, reads the data of Dick (i.e., Dick 1 21 70.6) stored in the file specified by `fptr` and assigns that data to the variable `sa`. As four conversion specifications are read successfully, the function `fscanf()` returns the `int` value 4, which is assigned to the `int` variable `m`.

Next, there is a `while` loop spanning LOCs 23 to 30. LOC 23 contains the continuation condition of the `while` loop; it is reproduced here for your quick reference:

```
while(m != EOF){                                                  /* L23 */
```

As the value of `m` is now 4 and not `EOF`, the first iteration begins. Next, the `printf` statement is executed, which spans LOCs 24 and 25, which are reproduced here for your quick reference:

```
printf("Name: %s, Roll no: %d, Age: %d, Weight: %.1f\n",          /* L24 */
                sa.name, sa.rollno,sa.age, sa.weight);            /* L25 */
```

The data items stored in the variable `sa` (i.e., the data of Dick) are displayed on the screen in LOCs 24 and 25. Next, the `fscanf` statement is executed, which spans LOCs 26 to 29. This statement is precisely the same as the statement spanning LOCs 19 to 22. It reads the data of the second secret agent (i.e., the data of Robert) stored in the file and assigns that data to the variable `sa`. Proceeding in this manner, the complete file is read. When the complete file is read and the function `fscanf()` tries to read the further data, which simply doesn't exist, then the reading operation fails and the `EOF` value is returned by this function, which is assigned to the `int` variable `m`. Then the iterations are terminated.

6-11. Write Integers to a Binary File

Problem

You want to write the integers to a binary file.

Solution

Write a C program that writes the integers to a binary file, with the following specifications:

- The program opens the binary file `num.dat` in writing mode and creates an array of integers.
- The program writes the integers to the file `num.dat` using the function `fwrite()`.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files11.c`:

```
/* This program writes an array of int values to a binary file using the
function fwrite(). */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
 int i, k, m, a[20];                                               /* L4 */
 FILE *fptr;                                                       /* L5 */
                                                                   /* BL */
 for(i = 0; i < 20; i++)                                           /* L6 */
  a[i] = 30000 + i;                                                /* L7 */
                                                                   /* BL */
 fptr = fopen("C:\\Code\\num.dat", "wb");                          /* L8 */
 if (fptr != NULL) {                                               /* L9 */
  puts("File num.dat is opened successfully.");                    /* L10 */
  m = fwrite(a, sizeof(int), 10, fptr);                            /* L11 */
  if (m == 10)                                                     /* L12 */
    puts("Data written to the file successfully.");                /* L13 */
  k = fclose(fptr);                                                /* L14 */
  if(k == -1)                                                      /* L15 */
    puts("File-closing failed");                                   /* L16 */
  if(k == 0)                                                       /* L17 */
   puts("File is closed successfully.");                           /* L18 */
 }                                                                 /* L19 */
 else                                                                 /* L20 */
  puts("File-opening failed");                                        /* L21 */
                                                                      /* BL */
 return(0);                                                           /* L22 */
}                                                                     /* L23 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File num.dat is opened successfully.
Data written to the file successfully.
File is closed successfully.
```

This program has written the ten integer values (30000, 30001, 30002, 30003, 30004, 30005, 30006, 30007, 30008, 30009) to the binary file `num.dat`.

Open the file `num.dat` in a suitable text editor, and you will find that its contents are as follows:

```
0u1u2u3u4u5u6u7u8u9u
```

The contents are unreadable (at least meaningless, if not readable) as the file is binary. You can read the contents of this file using the function `fread()`. The size of this file is 20 bytes as ten `int` values are stored in it (10 × 2 bytes = 20 bytes). A similar text file would end up consuming 50 bytes (10 × 5 bytes).

How It Works

The function `fwrite()` is used to write the data to a file in binary format. The function `fwrite()` is particularly suitable for writing an array to a file in binary format. The file must be opened in binary mode, if the function used for file writing is `fwrite()`. When you use the function `fwrite()` instead of the function `fprintf()`, there are two possible benefits: there is saving of storage space, in general, and the `fwrite` statement is less complex than the `fprintf` statement.

The generic syntax of a statement that uses the function `fwrite()` for writing an array to a file is given here:

```
m = fwrite(arrayName, sizeof(dataType), n, fptr);
```

Here, `m` is an `int` variable, `arrayName` is the name of the array to be written to a file, `dataType` is the data type of the array, `n` is the number of elements in the array to be written to the file, and `fptr` is a pointer to the `FILE` variable. The function `fwrite()` writes the `n` elements of the array `arrayName` to the file specified by `fptr` and returns a number that indicates the number of array elements successfully written. If the writing operation fails, then a zero value is returned. The returned value is generally ignored in small programs, but in professional programs it is collected and inspected to find out whether the writing operation was successful.

In LOCs 6 and 7, the `int` array `a` is populated with suitable values ranging from 30000 to 30019.

Next, consider LOC 8, which is reproduced here for your quick reference:

```
fptr = fopen("C:\\Code\\num.dat", "wb");                            /* L8 */
```

Notice that the file-opening mode is "wb". It means the file num.dat is opened for writing in binary mode. Next, consider LOC 11, which is reproduced here for your quick reference:

```
m = fwrite(a, sizeof(int), 10, fptr);                               /* L11 */
```

Notice that four arguments are passed to fwrite(). The first argument, a, is the name of the array to be written to the file; the second argument indicates the size of the array element (it is 2 bytes); the third argument, 10, indicates the number of elements of the array to be written to a file; and the fourth argument, fptr, specifies the file to which the array a is to be written. As the size of int is 2, you can replace the the second argument simply with 2. Also, the array a consists of 20 elements, but here we have chosen to write only 10 elements of this array to a file (see the third argument; it is 10). LOC 11, after execution, writes the first ten elements of the array a to the file specified by fptr and returns an int value 10, which is assigned to m. Notice the comfort of writing a complete array in a single statement to a file.

6-12. Write Structures to a Binary File

Problem

You want to write structures to a binary file.

Solution

Write a C program that writes structures to a binary file, with the following specifications:

- The program opens the binary file agents2.dat in writing mode.
- The program writes the structures to agents2.dat using the function fwrite().

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder C:\Code with the file name files12.c:

```
/* This program writes structures to a binary file using the function fwrite(). */
                                                                    /* BL */
#include <stdio.h>                                                  /* L1 */
                                                                    /* BL */
```

```
main()                                                          /* L2 */
{                                                               /* L3 */
 int k = 0;                                                     /* L4 */
 char flag = 'y';                                               /* L5 */
 FILE *fptr;                                                    /* L6 */
 struct biodata {                                               /* L7 */
  char name[15];                                                /* L8 */
  int rollno;                                                   /* L9 */
  int age;                                                      /* L10 */
  float weight;                                                 /* L11 */
 };                                                             /* L12 */
 struct biodata sa;                                             /* L13 */
 fptr = fopen("C:\\Code\\agents2.dat", "wb");                   /* L14 */
 if (fptr != NULL) {                                            /* L15 */
  printf("File agents2.dat is opened successfully.\n");         /* L16 */
                                                                /* BL  */
  while(flag == 'y'){                                           /* L17 */
   printf("Enter name, roll no, age, and weight of agent: ");   /* L18 */
   scanf("%s %d %d %f", sa.name,                                /* L19 */
                        &sa.rollno,                             /* L20 */
                        &sa.age,                                /* L21 */
                        &sa.weight);                            /* L22 */
   fwrite(&sa, sizeof(sa), 1, fptr);                            /* L23 */
   fflush(stdin);                                               /* L24 */
   printf("Any more records(y/n): ");                           /* L25 */
   scanf(" %c", &flag);                                         /* L26 */
  }                                                             /* L27 */
                                                                /* Bl  */
  k = fclose(fptr);                                             /* L28 */
  if(k == -1)                                                   /* L29 */
    puts("File-closing failed");                                /* L30 */
  if(k == 0)                                                    /* L31 */
    puts("File is closed successfully.");                       /* L32 */
 }                                                              /* L33 */
 else                                                           /* L34 */
  puts("File-opening failed");                                  /* L35 */
 return(0);                                                     /* L36 */
}                                                               /* L37 */
```

Compile and execute this program. A run of this program is given here:

```
File agents2.dat is opened successfully.
Enter name, roll no, age, and weight of agent: Dick  1  21  70.6    ↵
Any more records(y/n): y    ↵
Enter name, roll no, age, and weight of agent: Robert  2  22  75.8    ↵
Any more records(y/n): y    ↵
Enter name, roll no, age, and weight of agent: Steve  3  20  53.7    ↵
Any more records(y/n): y    ↵
```

```
Enter name, roll no, age, and weight of agent: Richard  4  19  83.1    ↵
Any more records(y/n): y    ↵
Enter name, roll no, age, and weight of agent: Albert  5  18  62.3    ↵
Any more records(y/n): n    ↵
File is closed successfully.
```

Open the file agents2.dat in a suitable text editor, and you will find that its contents are as follows:

```
Dick  Å
```

The contents are unreadable (at least meaningless, if not readable) as the file is binary. You can read the contents of this file using the function fread(). Also, compare these contents with the contents of the text file agents.dat, which was written using the function fprintf(), given here:

```
Dick 1 21 70.6Robert 2 22 75.8Steve 3 20 53.7Richard 4 19 83.1Albert 5 18 62.3
```

How It Works

The generic syntax of a statement that uses the function fwrite() for writing a single variable to a file is given here:

```
m = fwrite(&var, sizeof(var), 1, fptr);
```

Here, m is an int variable, var is a variable of any data type, and & is an address operator. The third argument, 1, indicates that the value of only one object is to be written to a file (here var is the object), and fptr is a pointer to the FILE variable. The function fwrite() writes the value of the variable var to a file specified by fptr and returns the int value 1 if the operation is successful; otherwise, it returns zero. The returned value is generally ignored in small programs, and it is ignored in the next program.

This program is a remake of the program files8. The only differences are the following:

- Instead of opening a file in text mode, here it is opened in binary mode.
- Instead of using the function fprintf() for writing the data to a file, the function fwrite() is used.

Notice LOC 14, which is reproduced here for your quick reference:

```
fptr = fopen("C:\\Code\\agents2.dat", "wb");                          /* L14 */
```

In LOC 14, you can see that file-opening mode is `"wb"`. It means the file `agents2.dat` is opened for writing in binary mode. Notice LOC 23, which is reproduced here for your quick reference:

```
fwrite(&sa, sizeof(sa), 1, fptr);                                         /* L23 */
```

This LOC, after execution, writes the data stored in the variable `sa` to the file specified by `fptr`. The third argument, `1`, indicates that only one object is to be written to a file (here `sa` is an object).

6-13. Read Integers Written to a Binary File

Problem

You want to read the integers written to a binary file.

Solution

Write a C program that reads the integers written to a binary file, with the following specifications:

- The program opens the binary file `num.dat` in reading mode.
- The program reads the integers written to the file using the function `fread()`.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files13.c`:

```
/* This program reads the binary file num.dat using the function fread() */
                                                                          /* BL */
#include <stdio.h>                                                        /* L1 */
                                                                          /* BL */
main()                                                                    /* L2 */
{                                                                         /* L3 */
 int i, k;                                                                /* L4 */
 int a[10];                                                               /* L5 */
 FILE *fptr;                                                              /* L6 */
                                                                          /* BL */
 fptr = fopen("C:\\Code\\num.dat", "rb");                                 /* L7 */
 if (fptr != NULL) {                                                      /* L8 */
  puts("File num.dat is opened successfully.");                           /* L9 */
                                                                          /* L10 */
 fread(a, sizeof(int), 10, fptr);                                         /* L11 */
```

```
                                                                 /* BL  */
 puts("Contents of file num.dat:");                             /* L12 */
                                                                 /* BL  */
 for(i = 0; i < 10; i++)                                        /* L13 */
 printf("%d\n", a[i]);                                          /* L14 */
                                                                 /* BL  */
 k = fclose(fptr);                                              /* L15 */
  if(k == -1)                                                   /* L16 */
    puts("File-closing failed");                                /* L17 */
  if(k == 0)                                                    /* L18 */
   puts("File is closed successfully.");                        /* L19 */
 }                                                              /* L20 */
 else                                                           /* L21 */
  puts("File-opening failed");                                  /* L22 */
                                                                 /* BL  */
 return(0);                                                     /* L23 */
}                                                               /* L24 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File num.dat is opened successfully.
Contents of file num.dat:
30000
30001
30002
30003
30004
30005
30006
30007
30008
30009
File is closed successfully.
```

How It Works

Think of the function `fread()`as a counterpart of the function `fwrite()`. It is used to read binary files. The generic syntax of a statement that uses the function `fread()` for reading an array stored in a file is given here:

```
m = fread(arrayName, sizeof(dataType), n, fptr);
```

Here, `m` is an `int` variable, `arrayName` is the name of the array in which the values read will be stored, `dataType` is the data type of an array (e.g., `int`, `float`, etc.), `n` is the number of values to be read, and `fptr` is a pointer to `FILE` that specifies the file to be read. The function `fread()` reads the `n` values of the data type `dataType` from the file specified by `fptr`, stores them in the array `arrayName`, and returns a number that indicates the number of values read. If the reading operation is quite successful, then the value n is returned. If the reading operation fails and no value is read, then zero is returned. In small programs, the returned value is generally ignored.

Consider LOC 7, which is reproduced here for your quick reference:

```
fptr = fopen("C:\\Code\\num.dat", "rb");                              /* L7 */
```

Notice that file-opening mode is "rb". It means the file num.dat is opened for reading in binary mode. Next, notice LOC 11, which is reproduced here for your quick reference:

```
fread(a, sizeof(int), 10, fptr);                                     /* L11 */
```

In this LOC, the function fread() reads the ten int values stored in the file specified by fptr and then stores these values in the first ten elements of the array a. (Why ten? The third argument says so, and why int? The second argument says so.) Notice the comfort of reading a complete array in a single statement.

Values stored in the array a are then displayed on the screen using the for loop that spans LOCs 13 and 14, reproduced here for your quick reference:

```
for(i = 0; i < 10; i++)                                              /* L13 */
printf("%d\n", a[i]);                                                /* L14 */
```

This for loop performs ten iterations and displays the values stored in the ten elements of array a.

6-14. Read Structures Written to a Binary File

Problem

You want to read the structures written to a binary file.

Solution

Write a C program that reads the structures written to the binary file, with the following specifications:

- The program opens the binary file agents2.dat in reading mode.
- The program reads the structures written to the file using the function fread().

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files14.c`:

```
/* This program reads the structures stored in the binary file agents2.dat */
/* using the function fread() */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
 int k = 0, m = 0;                                                 /* L4 */
 FILE *fptr;                                                       /* L5 */
 struct biodata{                                                   /* L6 */
  char name[15];                                                   /* L7 */
  int rollno;                                                      /* L8 */
  int age;                                                         /* L9 */
  float weight;                                                    /* L10 */
 };                                                                /* L11 */
 struct biodata sa;                                                /* L12 */
 fptr = fopen("C:\\Code\\agents2.dat", "rb");                      /* L13 */
 if (fptr != NULL) {                                               /* L14 */
  printf("File agents2.dat is opened successfully.\n");            /* L15 */
  m = fread(&sa, sizeof(sa), 1, fptr);                             /* L16 */
                                                                   /* BL  */
  while(m != 0){                                                   /* L17 */
   printf("Name: %s, Roll no: %d, Age: %d, Weight: %.1f\n",        /* L18 */
                sa.name, sa.rollno,sa.age, sa.weight);             /* L19 */
   m = fread(&sa, sizeof(sa), 1, fptr);                            /* L20 */
  }                                                                /* L21 */
                                                                   /* BL  */
  k = fclose(fptr);                                                /* L22 */
  if(k == -1)                                                      /* L23 */
    puts("File-closing failed");                                   /* L24 */
  if(k == 0)                                                       /* L25 */
    puts("File is closed successfully.");                          /* L26 */
 }                                                                 /* L27 */
 else                                                              /* L28 */
  puts("File-opening failed");                                     /* L29 */
 return(0);                                                        /* L30 */
}                                                                  /* L31 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File agents2.dat is opened successfully.
Name: Dick, Roll no: 1, Age: 21, Weight: 70.6
Name: Robert, Roll no: 2, Age: 22, Weight: 75.8
Name: Steve, Roll no: 3, Age: 20, Weight: 53.7
Name: Richard, Roll no: 4, Age: 19, Weight: 83.1
Name: Albert, Roll no: 5, Age: 18, Weight: 62.3
File is closed successfully.
```

How It Works

This program is a remake of the program `files10` with the following differences:

- In this program, file-opening mode is `"rb"` (i.e., the file `agents2.dat` will be opened for reading in binary mode).
- The function `fread()` is used instead of the function `fscanf()` to read the file.

Now let's discuss how this program works. Notice LOC 16, which is reproduced here for your quick reference:

```
m = fread(&sa, sizeof(sa), 1, fptr);                                    /* L16 */
```

This LOC, after execution, reads the data of Dick stored in the file specified by `fptr` and stores that data in the variable `sa`. The second argument indicates the size of the variable `sa`. The third argument indicates that only one object (here data that will be stored in the variable `sa` is the object) is to be read from the file. The fourth argument indicates that the file to be read is specified by `fptr`. As one object is successfully read, the integer value 1 is returned by this function, which is assigned to `m`. You are interested in the returned value so that you can detect the end of file. When the end of file occurs, then no object can be read, and `fread()` returns zero.

Next, there is a `while` loop that spans LOCs 17 to 21. Notice LOC 17, which is reproduced here for your quick reference:

```
while(m != 0){                                                          /* L17 */
```

Notice the continuation condition that says iterations are permitted, while `m` is not equal to zero. As the value of `m` is now 1, the first iteration is permitted. Now the first iteration begins. Next, the `printf` statement spanning LOCs 18 to 19 is executed. This is a single statement, but it is put onto two LOCs because it is long. This `printf` statement displays the data of Dick on the screen. Next, LOC 20 is executed, which is the same as LOC 16. LOC 20, after execution, reads the data of Robert from the file and stores it in the variable `sa`. This is how the data of successive members is read from the file. When the end of file occurs and `fread()` tries to read from the file, then the reading operation fails and zero is returned (`m` becomes equal to zero), which terminates the execution of the loop.

6-15. Rename a File

Problem

You want to rename a file.

Solution

Write a C program that renames the file `kolkata.txt` as `city.dat`.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files15.c`:

```
/* This program changes the name of file kolkata.txt to city.dat. */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
 int m;                                                            /* L4 */
 m = rename("C:\\Code\\kolkata.txt", "C:\\Code\\city.dat");        /* L5 */
 if (m == 0)                                                       /* L6 */
   puts("Operation of renaming a file is successful.");            /* L7 */
 if (m != 0)                                                       /* L8 */
   puts("Operation of renaming a file failed.");                   /* L9 */
 return(0);                                                        /* L10 */
}                                                                  /* L11 */
```

Compile and execute this program, and the following line of text appears on the screen:

```
Operation of renaming a file is successful.
```

Open the file `city.dat` in a suitable text editor and verify that its contents are as shown here:

```
Kolkata is very big city.
It is also very nice city.
```

How It Works

In C, you can rename a file using the function rename(). Also, you can delete a file using the function remove(). The generic syntax of a statement that uses the function rename() is as follows:

```
n = rename(oldFilename, newFilename);
```

Here, oldFilename and newFilename are expressions that evaluate to the string constants, which in turn consist of the old file name and new file name, respectively; n is an int variable. The function rename() changes the name of the file from oldFilename to newFilename and returns an integer value, which is assigned to n. If the operation of renaming is successful, then the zero value is returned; otherwise, a nonzero value is returned.

In this program, the file kolkata.txt is renamed as city.dat in LOC 5. LOC 5 is reproduced here for your quick reference:

```
m = rename("C:\\Code\\kolkata.txt", "C:\\Code\\city.dat");       /* L5 */
```

The first argument to the function rename() is an old file name, and the second argument to the function rename() is a new file name.

6-16. Delete a File

Problem

You want to delete a file.

Solution

Write a C program that deletes the file city.dat.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder C:\Code with the file name files16.c:

```
/* This program deletes the file city.dat. */
                                                                      /* BL */
#include <stdio.h>                                                    /* L1 */
                                                                      /* BL */
main()                                                                /* L2 */
{                                                                     /* L3 */
 int m;                                                               /* L4 */
 m = remove("C:\\Code\\city.dat");                                    /* L5 */
 if (m == 0)                                                /* L6 */
   puts("Operation of deletion of file is successful.");    /* L7 */
 if (m != 0)                                                /* L8 */
   puts("Operation of deletion of file failed.");           /* L9 */
 return(0);                                                 /* L10 */
}                                                           /* L11 */
```

Compile and execute this program, and the following line of text appears on the screen:

```
Operation of deletion of file is successful.
```

The file `city.dat` is now deleted, and you can verify it by suitable means.

How It Works

The function `remove()` is used to delete (i.e., remove) a file. The generic syntax of a statement that uses the function `remove()` is as follows:

```
n = remove(filename) ;
```

Here, `filename` is an expression that evaluates to a string constant that consists of the name of the file to be deleted, and `n` is an `int` variable. The function `remove()` deletes the file name and returns an integer value that is assigned to `n`. If the operation of deleting a file is successful, then a zero value is returned; otherwise, a nonzero value is returned.

LOC 5 deletes the file `city.dat`. LOC 5 is reproduced here for your quick reference:

```
m = remove("C:\\Code\\city.dat");                           /* L5 */
```

The only argument to the function `remove()` is the name of the file to be deleted with the path.

6-17. Copy a Text File

Problem

You want to copy a text file.

Solution

Write a C program that creates a copy of the text file `satara.txt` with the file name `town.dat`.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files17.c`:

```
/* This program creates a copy of the text file satara.txt
with the filename town.dat. */
                                                                   /* BL */
#include <stdio.h>                                                 /* L1 */
                                                                   /* BL */
main()                                                             /* L2 */
{                                                                  /* L3 */
 FILE *fptrSource, *fptrTarget;                                    /* L4 */
 int m, n, p;                                                      /* L5 */
                                                                   /* BL */
 fptrSource = fopen("C:\\Code\\satara.txt", "r");                  /* L6 */
 if(fptrSource == NULL){                                           /* L7 */
  puts("Source-file-opening failed");                              /* L8 */
  exit(1);                                                         /* L9 */
 }                                                                 /* L10 */
 puts("Source-file satara.txt opened successfully");               /* L11 */
                                                                   /* BL  */
 fptrTarget = fopen("C:\\Code\\town.dat", "w");                    /* L12 */
 if(fptrTarget == NULL){                                           /* L13 */
  puts("Target-file-opening failed");                              /* L14 */
  exit(2);                                                         /* L15 */
 }                                                                 /* L16 */
 puts("Target-file town.dat opened successfully");                 /* L17 */
                                                                   /* BL  */
 m = fgetc(fptrSource);                                            /* L18 */
                                                                   /* BL  */
 while(m != EOF){                                                  /* L19 */
  fputc(m, fptrTarget);                                            /* L20 */
  m = fgetc(fptrSource);                                           /* L21 */
 }                                                                 /* L22 */
                                                                   /* BL  */
 puts("File copied successfully");                                 /* L23 */
                                                                   /* BL  */
 n = fclose(fptrSource);                                           /* L24 */
 if(n == -1)                                                       /* L25 */
  puts("Source-file-closing failed");                              /* L26 */
 if(n == 0)                                                        /* L27 */
  puts("Source-file closed successfully");                         /* L28 */
                                                                   /* BL  */
 p = fclose(fptrTarget);                                           /* L29 */
 if(p == -1)                                                       /* L30 */
  puts("Target-file-closing failed");                              /* L31 */
 if(p == 0)                                                     /* L32 */
  puts("Target-file closed successfully");                      /* L33 */
                                                                /* BL  */
 return(0);                                                     /* L34 */
}                                                               /* L35 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
Source-file satara.txt opened successfully
Target-file town.dat opened successfully
File copied successfully
Source-file closed successfully
Target-file closed successfully
```

Open the file `town.dat` in a suitable text editor and ensure that its contents are as shown here:

```
Satara is surrounded by mountains.
Satara was capital of Maratha empire for many years.
```

This confirms that the newly created file `town.dat` is an exact replica of the file `satara.txt`.

How It Works

The generic steps involved in the process of creating a copy of a text file are as follows:

1. Open the source file for reading in text mode.
2. Open the target file for writing in text mode.
3. Read a character in the source file and write it to the target file.
4. Repeat the step 3 until the end of source file occurs.
5. Close the files.

The source file `satara.txt` is opened in LOC 6. If file opening fails, then LOCs 8 and 9 are executed. Of particular interest is LOC 9, which causes the program to terminate, and it is reproduced here for your quick reference:

```
exit(1);                                                       /* L9 */
```

The function `exit()` causes the program to terminate. It is like an emergency exit.

The generic syntax of a statement that uses the function `exit()` is given here:

```
exit(n);
```

Here, n is an expression that evaluates to an integer constant. If the value of n is zero, then it indicates the normal termination of a program. If the value of n is nonzero, then it indicates the abnormal termination of a program. This indication is meant for the caller program. Before the termination of the program, the function exit() performs the following tasks:

- Flushes out the input and output buffers
- Closes all the open files

In LOC 9, you pass a nonzero argument to the function exit() indicating the abnormal termination of the program. Now notice LOC 15, which is reproduced here for your quick reference:

```
exit(2);                                                                /* L15 */
```

This time you have also passed a nonzero argument to the function exit() to indicate the abnormal termination of the program. But this time you have chosen another nonzero value, 2. Now the caller program is able to know the precise cause of program termination. The caller program inspects the argument; if it is 1, then it concludes that the program terminated because the source file opening failed, and if it is 2, then it concludes that the program terminated because the target file opening failed. Here the caller program is the function main().

The functions fgetc() and fputc() are used in this program to read a character from a file and to write a character to a file, respectively.

6-18. Copy a Binary File

Problem

You want to copy a binary file.

Solution

Write a C program that creates a copy of the binary file hello.exe with the file name world.exe. Also, ensure that the executable file hello.exe is available in the folder C:\Output. This file displays the text “hello, world” on the screen after execution, and it is the executable version of the ubiquitous hello program coded by Brian Kernighan.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder C:\Code with the file name files18.c:

```
/* This program creates a copy of the binary file hello.exe named as world.exe. */
                                                                       /* BL */
#include <stdio.h>                                                     /* L1 */
                                                                       /* BL */
```

```
main()                                                             /* L2 */
{                                                                  /* L3 */
 FILE *fptrSource, *fptrTarget;                                    /* L4 */
 int m, n, p;                                                      /* L5 */
                                                                   /* BL */
 fptrSource = fopen("C:\\Output\\hello.exe", "rb");                /* L6 */
 if(fptrSource == NULL){                                           /* L7 */
  puts("Source-file-opening failed");                              /* L8 */
  exit(1);                                                         /* L9 */
 }                                                                 /* L10 */
 puts("Source-file Hello.exe opened successfully");                /* L11 */
                                                                   /* BL  */
 fptrTarget = fopen("C:\\Output\\world.exe", "wb");                /* L12 */
 if(fptrTarget == NULL){                                           /* L13 */
  puts("Target-file-opening failed");                              /* L14 */
  exit(2);                                                         /* L15 */
 }                                                                 /* L16 */
 puts("Target-file World.exe opened successfully");                /* L17 */
                                                                   /* BL  */
 m = fgetc(fptrSource);                                            /* L18 */
                                                                   /* BL  */
 while(m != EOF){                                                  /* L19 */
  fputc(m, fptrTarget);                                            /* L20 */
  m = fgetc(fptrSource);                                           /* L21 */
 }                                                                 /* L22 */
                                                                   /* BL  */
 puts("File copied successfully");                                 /* L23 */
                                                                   /* BL  */
 n = fclose(fptrSource);                                           /* L24 */
 if(n == -1)                                                       /* L25 */
  puts("Source-file-closing failed");                              /* L26 */
 if(n == 0)                                                        /* L27 */
  puts("Source-file closed successfully");                         /* L28 */
                                                                   /* BL  */
 p = fclose(fptrTarget);                                           /* L29 */
 if(p == -1)                                                       /* L30 */
  puts("Target-file-closing failed");                              /* L31 */
 if(p == 0)                                                        /* L32 */
  puts("Target-file closed successfully");                         /* L33 */
                                                                   /* BL  */
 return(0);                                                        /* L34 */
}                                                                  /* L35 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
Source-file Hello.exe opened successfully
Target-file World.exe opened successfully
File copied successfully
Source-file closed successfully
Target-file closed successfully
```

Using a suitable Command Prompt window, execute the file `world.exe` and ensure that the following output is displayed on the screen:

```
hello, world
```

This confirms that the file `world.exe` is an exact replica of the file `hello.exe`.

How It Works

This program is a remake of the program `files17`. The program `files17` creates a copy of the text file, whereas this program (i.e., `files18`) creates a copy of the binary file. The only differences are the following:

- In the program `files17`, the source file (`satara.txt`) is opened for reading in text mode, whereas in the program `files18`, the source file (`hello.exe`) is opened for reading in binary mode.
- In the program `files17`, the target file (`town.dat`) is opened for writing in text mode, whereas in the program `files18`, the target file (`world.exe`) is opened for writing in binary mode.

LOCs 6 and 12 are reproduced here for your quick reference:

```
fptrSource = fopen("C:\\Output\\hello.exe", "rb");                /* L6 */
fptrTarget = fopen("C:\\Output\\world.exe", "wb");                /* L12 */
```

The first argument to the function `fopen()` is the name of the file to be opened with the path, and the second argument is the mode in which the file is to be opened. You can see that the opening mode for the file `hello.exe` is “reading and binary,” and the opening mode for the file `world.exe` is “writing and binary.”

6-19. Write to a File and Then Read from That File

Problem

You want to write to a file and also to read that file.

Solution

Write a C program that writes to a file and also reads that file, with the following specifications:

- The program opens the text file `pune.txt` in writing mode. The program writes some text to this file.
- The program rewinds the file `pune.txt` using the function `rewind()`.
- The program reads the file `pune.txt` and displays the text on the screen.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files19.c`:

```
/* This program performs write and read operations on a file. */
                                                               /* BL */
#include <stdio.h>                                             /* L1 */
                                                               /* BL */
main()                                                         /* L2 */
{                                                              /* L3 */
 FILE *fptr;                                                   /* L4 */
 char store[80];                                               /* L5 */
 int k;                                                        /* L6 */
 fptr = fopen("C:\\Code\\pune.txt", "w+");                     /* L7 */
                                                               /* BL */
 if(fptr != NULL){                                             /* L8 */
  puts("File pune.txt opened successfully");                   /* L9 */
  fputs("Pune is very nice city.", fptr);                      /* L10 */
  puts("Text written to file pune.txt successfully");          /* L11 */
  rewind(fptr);                                                /* L12 */
  fgets(store, 80, fptr);                                      /* L13 */
  puts("Contents of file pune.txt:");                          /* L14 */
  puts(store);                                                 /* L15 */
  k = fclose(fptr);                                            /* L16 */
  if(k == -1)                                                  /* L17 */
   puts("File-closing failed");                                /* L18 */
  if(k == 0)                                                   /* L19 */
   puts("File closed successfully");                           /* L20 */
 }                                                             /* L21 */
```

```
  else                                                              /* L22 */
   puts("File-opening failed");                                     /* L23 */
                                                                    /* BL  */
 return(0);                                                         /* L24 */
}                                                                   /* L25 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File pune.txt opened successfully
Text written into file pune.txt successfully
Contents of file pune.txt:
Pune is very nice city.
File closed successfully.
```

How It Works

Hitherto, in every program, you have either written to a file or read a file. However, in this program, you have written to a file, and then the same file is read. This can be done in number of ways. Let's note couple of methods of doing so.

For the first method, follow these steps:

1. Open a file in "w" mode using the function `fopen()`.
2. Write to the file the desired data.
3. Close the file using the function `fclose()`.
4. Open the file again in "r" mode using the function `fopen()`.
5. Read the file.
6. Close the file using the function `fclose()`.

For the second method (used in program `files19`), follow these steps:

1. Open a file in "w+" mode using the function `fopen()`.
2. Write to the file the desired data.
3. Rewind the file using the function `rewind()`. To rewind the file means to position the file to its first character.
4. Read the file.
5. Close the file using the function `fclose()`.

When you open a file using the function `fopen()`, then the file is always positioned to its first character (i.e., the marker is pointing to the first character of the file). As you read the file (or write to the file), then the marker marches ahead accordingly. In the second method, in step 3, you have rewound the file using the function `rewind()`. This step is necessary because when you write to the file in step 2, then the marker is pointing to the `(n + 1)`th byte of the file provided that you have written `n` characters to the file. However, before you read the file, the latter must be positioned to its first character. This can be done simply by closing the file and then opening it again. Alternatively, this can be done simply by calling the function `rewind()`, which positions the file to its first character.

Notice the simplified syntax of a statement that uses the function `rewind()` given here:

```
rewind(fptr);
```

Here, `fptr` is a pointer to the `FILE` variable. The function `rewind()` rewinds the file specified by `fptr`.

In LOC 10 a string is written to the file `pune.txt`. In LOC 12 the file is rewound using the function `rewind()`. After the execution of LOC 12, the file is positioned to its first character. In LOC 13, a file is read, and the contents of the file (which are nothing but a single string) are stored in the `char` array `store`. In LOC 15, the string stored in `store` is displayed on the screen.

6-20. Position a Text File to a Desired Character

Problem

You want to position a text file to a desired character in that file.

Solution

Write a C program that positions a text file to a desired character in that file, with the following specifications:

- The program opens the text file `pune.txt` in reading mode.
- The program uses the function `ftell()` to find out the current position of the file. Also, the program uses the function `fseek()` to position the file to the desired character in that file.
- The program positions the file to various desired characters in the file using the functions `fseek()` and `ftell()`.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files20.c`:

```
/* This program postions the file to desired characters in that file using */
/* the functions fseek() and ftell(). */
                                                                  /* BL */
#include <stdio.h>                                                /* L1 */
                                                                  /* BL */
main()                                                            /* L2 */
{                                                                 /* L3 */
```

```
FILE *fptr;                                                     /* L4 */
int m, n, k, p;                                                 /* L5 */
fptr = fopen("C:\\Code\\pune.txt", "r");                        /* L6 */
                                                                /* BL */
if(fptr != NULL){                                               /* L7 */
 puts("File pune.txt opened successfully");                     /* L8 */
 puts("Let n denotes current file position");                   /* L9 */
 n = ftell(fptr);                                               /* L10 */
 printf("Now value of n is %d\n", n);                           /* L11 */
 printf("Let us read a single character and it is: ");          /* L12 */
 m = fgetc(fptr);                                               /* L13 */
 putchar(m);                                                    /* L14 */
 printf("\n");                                                  /* L15 */
 n = ftell(fptr);                                               /* L16 */
 printf("Now value of n is %d\n", n);                           /* L17 */
 fseek(fptr, 8, 0);                                             /* L18 */
 puts("Statement \"fseek(fptr, 8, 0);\" executed");             /* L19 */
 n = ftell(fptr);                                               /* L20 */
 printf("Now value of n is %d\n", n);                           /* L21 */
 fseek(fptr, 3, 1);                                             /* L22 */
 puts("Statement \"fseek(fptr, 3, 1);\" executed");             /* L23 */
 n = ftell(fptr);                                               /* L24 */
 printf("Now value of n is %d\n", n);                           /* L25 */
 fseek(fptr, -5, 1);                                            /* L26 */
 puts("Statement \"fseek(fptr, -5, 1);\" executed");            /* L27 */
 n = ftell(fptr);                                               /* L28 */
 printf("Now value of n is %d\n", n);                           /* L29 */
 fseek(fptr, -3, 2);                                            /* L30 */
 puts("Statement \"fseek(fptr, -3, 2);\" executed");            /* L31 */
 n = ftell(fptr);                                               /* l32 */
 printf("Now value of n is %d\n", n);                           /* L33 */
 fseek(fptr, 0, 2);                                             /* L34 */
 puts("Statement \"fseek(fptr, 0, 2);\" executed");             /* L35 */
 n = ftell(fptr);                                               /* L36 */
 printf("Now value of n is %d\n", n);                           /* L37 */
 puts("Now let us perform a read operation");                   /* L38 */
 m = fgetc(fptr);                                               /* L39 */
 printf("Value read is %d\n", m);                               /* L40 */
 n = ftell(fptr);                                               /* L41 */
 printf("Now value of n is still %d\n", n);                     /* L42 */
 fseek(fptr, 0, 0);                                             /* L43 */
 puts("Statement \"fseek(fptr, 0, 0);\" executed");             /* L44 */
 n = ftell(fptr);                                               /* L45 */
 printf("Now value of n is %d\n", n);                           /* L46 */
 puts("That's all.");                                           /* L47 */
                                                                /* BL  */
 k = fclose(fptr);                                              /* L48 */
 if(k == -1)                                                    /* L49 */
```

```
   puts("File-closing failed");                                      /* L50 */
  if(k == 0)                                                         /* L51 */
   puts("File closed successfully.");                                /* L52 */
 }                                                                   /* L53 */
  else                                                               /* L54 */
   puts("File-opening failed");                                      /* L55 */
                                                                     /* BL  */
 return(0);                                                          /* L56 */
}                                                                    /* L57 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
File pune.txt opened successfully
Let n denotes current file position
Now value of n is 0
Let us read a single character and it is: P
Now value of n is 1
Statement "fseek (fptr, 8, 0);" executed
Now value of n is 8
Statement "fseek (fptr, 3, 1);" executed
Now value of n is 11
Statement "fseek (fptr, -5, 1);" executed
Now value of n is 6
Statement "fseek (fptr, -3, 2);" executed
Now value of n is 20
Statement "fseek (fptr, 0, 2);" executed
Now value of n is 23
Now let us perform a read operation
Value read is -1
Now value of n is still 23
Statement "fseek (fptr, 0, 0);" executed
Now value of n is 0
That's all.
```

How It Works

The function `fseek()` is used to position a file to the desired character of that file. The function `ftell()` is used to retrieve the current position of a file. Notice LOC 6, which is reproduced here for your quick reference:

```
fptr = fopen("C:\\Code\\pune.txt", "r");                            /* L6 */
```

After the execution of LOC 6, the file `pune.txt` opens in `"r"` mode. Also, the file is positioned to the first character of the file (i.e., `'P'`), as shown in Figure 6-2 (a). Whenever you open a file using the function `fopen()`, the file is always positioned to the first character of the file.

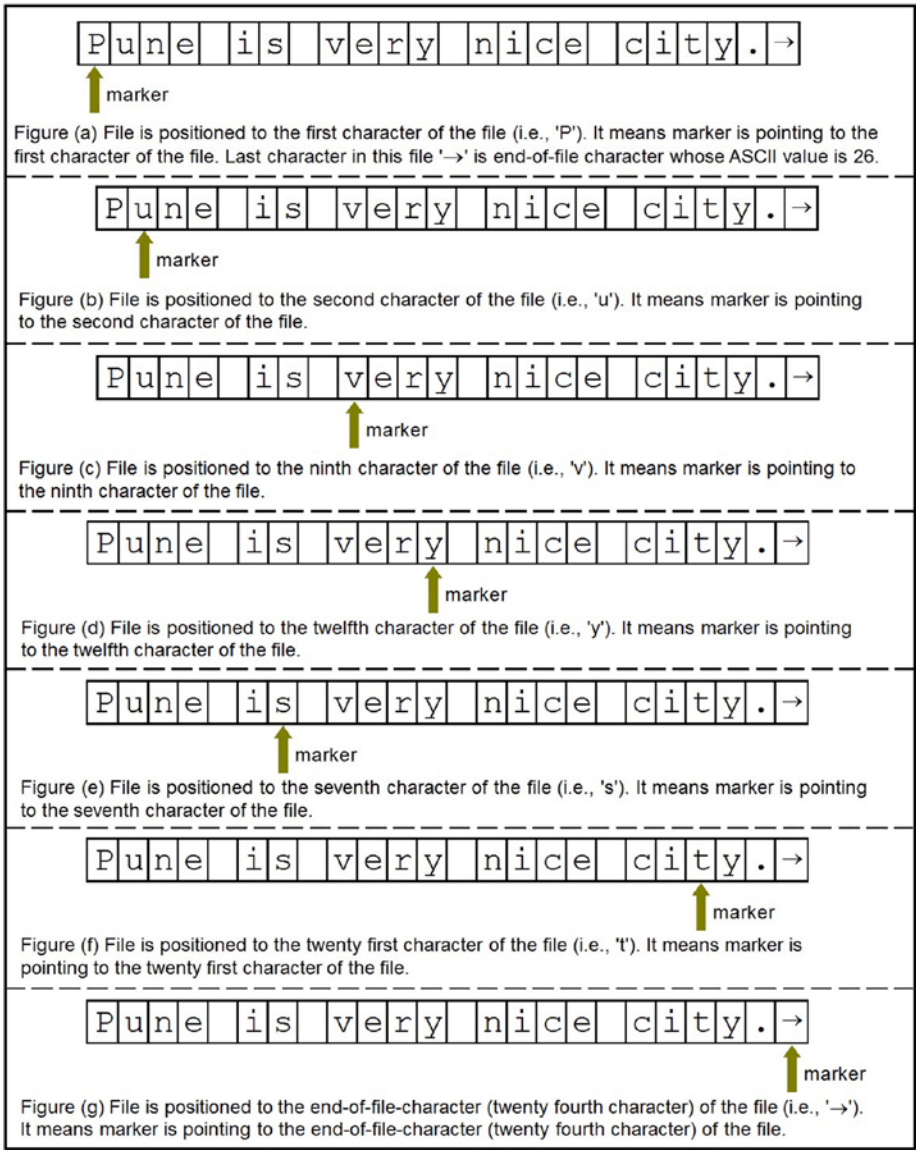
P u n e i s v e r y n i c e c i t y . →
marker
Figure (a) File is positioned to the first character of the file (i.e., 'P'). It means marker is pointing to the first character of the file. Last character in this file '→' is end-of-file character whose ASCII value is 26.
marker
Figure (b) File is positioned to the second character of the file (i.e., 'u'). It means marker is pointing to the second character of the file.
marker
Figure (c) File is positioned to the ninth character of the file (i.e., 'v'). It means marker is pointing to the ninth character of the file.
marker
Figure (d) File is positioned to the twelfth character of the file (i.e., 'y'). It means marker is pointing to the twelfth character of the file.
marker
Figure (e) File is positioned to the seventh character of the file (i.e., 's'). It means marker is pointing to the seventh character of the file.
marker
Figure (f) File is positioned to the twenty first character of the file (i.e., 't'). It means marker is pointing to the twenty first character of the file.
marker
Figure (g) File is positioned to the end-of-file-character (twenty fourth character) of the file (i.e., '→'). It means marker is pointing to the end-of-file-character (twenty fourth character) of the file.

Figure 6-2. *Positioning of the file*

You can retrieve the current position of a file using the function `ftell()`. Notice LOC 10, which is reproduced here for your quick reference:

```
n = ftell(fptr);                                              /* L10 */
```

In LOC 10, the function `ftell()` returns the long `int` value 0 (the index of the first character in the file, i.e., `'P'`), which is assigned to the `long int` variable `n`. Notice that characters in the file are indexed beginning with zero, which is akin to elements in an array. It means the index of the first character `'P'` is 0, the index of the second character `'u'` is 1, the index of the third character `'n'` is 2, and so on.

Notice the generic syntax of a statement that uses the function `ftell()` given here:

```
n = ftell(fptr);
```

Here, `n` is a `long int` variable, and `fptr` is a pointer to the `FILE` variable. The function `ftell()` returns a long `int` value, which indicates the position of a file specified by `fptr`.

A character is read using the function `fgetc()`. Notice LOC 13, which is reproduced here for your quick reference:

```
m = fgetc(fptr);                                        /* L13 */
```

In LOC 13, the function `fgetc()` reads the character `'P'` and returns its ASCII value (it is 80), which is assigned to the `int` variable `m`. After the execution of LOC 13, the file is positioned to the second character of the file (i.e., `'u'`), as shown in Figure 6-2 (b). LOC 16 is same as LOC 10, and it is reproduced here for your quick reference:

```
n = ftell(fptr);                                        /* L16 */
```

In LOC 16, the function `ftell()` returns the `long int` value 1 (the index of the second character of the file, i.e., `'u'`), which is assigned to the `long int` variable `n`. In LOC 18, the file is positioned to the ninth character in the file (i.e., `'v'`), as shown in Figure 6-2 (c). LOC 18 is reproduced here for your quick reference:

```
  fseek(fptr, 8, 0);                                    /* L18 */
```

Because indexing begins with zero, the index of `'v'` is 8, and it appears as the second argument in the function call in LOC 18. The first argument in this function call is `fptr`, a pointer to the `FILE` variable, and it indicates the file to be positioned. The third argument in this function call is the integer value 0, and it indicates that the counting of characters is to be made from the beginning of the file.

Notice the generic syntax of a statement that uses the function `fseek()` given here:

```
p = fseek(fptr, offset, origin);
```

Here, `p` is an `int` variable, `fptr` is a pointer to the `FILE` variable, `offset` is an expression that evaluates to a `long int` value, and `origin` is one of the three values 0, 1, or 2. `offset` indicates the character offset counted from the `origin`; when `origin` is 0, then `offset` is counted from the beginning of the file (as in the case of LOC 18). When `origin` is 1, then `offset` is counted from the current position of the file (i.e., the current position of the character pointed to by the marker). When `origin` is 2, then `offset` is counted from the end of the file. The function `fseek()` positions the file as specified by the arguments and then returns 0 if the operation of positioning the file is successful; otherwise, it returns a nonzero value.

In LOC 20, once again, the current position of the file is retrieved. It is precisely the same as LOC 13 or LOC 16. In LOC 20, the function `ftell()` returns the `long int` value 8 (the index of the ninth character of the file, i.e., `'v'`), which is assigned to the `long int` variable `n`.

In LOC 22, the file is positioned to the character `'y'` (in the word `"very"`), as shown in Figure 6-2 (d). LOC 22 is reproduced here for your quick reference:

```
fseek(fptr, 3, 1);                                              /* L22 */
```

In this function call, `origin` (the third argument) is 1, which means the counting of characters is to be made from the current position of the file (i.e., `'v'`). Also, `offset` (the second argument) is 3. Thus, the third character from `'v'` is `'y'`, and hence the file is positioned to `'y'`.

In LOC 24, once again, the current position of the file is retrieved, and in this case the value of `n` is 11 because `'y'` is the 12th character of the file.

In LOC 26, the file is positioned to the character `'s'` (in the word `"is"`), as shown in Figure 6-2 (e). LOC 26 is reproduced here for your quick reference:

```
fseek(fptr, -5, 1);                                             /* L26 */
```

Notice that in this function call `origin` (the third argument) is 1; it means the counting of characters is to be made from the current position of the file (i.e., `'y'`). Also, `offset` (the second argument) is -5. Notice that `offset` is a negative value, it means counting is to be made in the reverse direction, i.e., to the beginning of file. Thus, the fifth character from `'y'` in the reverse direction is `'s'` (in the word `"is"`), and hence the file is positioned to `'s'`.

In LOC 28, once again, the current position of the file is retrieved, and in this case the value of `n` is 6 because `'s'` is the seventh character of the file.

In LOC 30, the file is positioned to the character `'t'` (in the word `"city"`), as shown in Figure 6-2 (f). LOC 30 is reproduced here for your quick reference:

```
fseek(fptr, -3, 2);                                             /* L30 */
```

Notice that in this function call `origin` (the third argument) is 2; this means the counting of characters is to be made from the end of the file (i.e., from the end-of-file character). Also, `offset` (the second argument) is -3. As offset is a negative value, counting is to be made in the reverse direction, i.e., to the beginning of file. Thus, the third character from the end-of-file-character in the reverse direction is `'t'`, and hence the file is positioned to `'t'`. In this case, you can imagine that the index of the end-of-file character is 0, the index of the period (`.`) is -1, the index of `'y'` is -2, and the index of `'t'` is -3.

Also, notice that when `origin` is 0, then `offset` must be zero or a positive number. When `origin` is 2, then `offset` must be zero or negative number. When `origin` is 1, then `offset` can be a positive or negative number.

In LOC 32, once again, the current position of the file is retrieved, and in this case the value of `n` is 20 because `'t'` is the 21st character of the file.

In LOC 34, the file is positioned to the end-of-file character, as shown in Figure 6-2 (g). LOC 34 is reproduced here for your quick reference:

```
fseek(fptr, 0, 2);                                              /* L34 */
```

In LOC 36, once again, the current position of the file is retrieved, and in this case the value of `n` is 23 because the end-of-file character is the 24th character of the file.

In LOC 39, a read operation is performed. LOC 39 is reproduced here for your quick reference:

```
m = fgetc(fptr);                                                /* L39 */
```

Now instead of the ASCII value of the end-of-file character, a special value `EOF` (its value is `-1`) is returned by the function `fgetc()`, which is assigned to `m`. You inspect the value of `m` after every read operation to find out whether the end of file has occurred.

LOC 40 displays the value of `m` on the screen, and it is -1.

Whenever a read operation is performed using the function `fgetc()`, then the marker is advanced to the next character automatically. But after the execution of LOC 39, the marker is not advanced because the marker is already pointing to the end-of-file character, and it simply cannot be advanced. This is verified in LOC 41, which is reproduced here for your quick reference:

```
n = ftell(fptr);                                                  /* L41 */
```

In LOC 41, the value of `n` turns out to be 23, as expected.

In LOC 43, the file is positioned to the first character of the file, as shown in Figure 6-2 (a). LOC 43 is reproduced here for your quick reference:

```
fseek(fptr, 0, 0);                                                /* L43 */
```

In LOC 45, once again, the current position of the file is retrieved, and in this case the value of `n` is 0 as expected.

Instead of LOC 43, you can use the LOC given here to position the file to the first character of the file:

```
rewind();                                           /* Equivalent to L43 */
```

Imagine that after LOC 45, the following LOC is executed:

```
fseek(fptr, 30, 0);                         /* Imagine this LOC after L45 */
```

After execution of the LOC shown, the file is positioned to the 31st character of the file. But the file doesn't contain 31 characters. The index of the last character (end-of-file character) in the file is 23. Hence, this LOC should be considered erratic even though you can compile and execute this LOC successfully.

■ **Note** Never make the marker point to a character that is not part of the file.

6-21. Read from the Device File Keyboard

Problem

You want to read from the device file keyboard.

Solution

Write a C program with the following specifications:

- The program implements the keyboard using a pointer to the `FILE` constant `stdin`.
- The program reads the data from the keyboard and displays it on the screen.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files21.c`:

```
/* This program reads the device-file "keyboard" and displays */
/* the contents of this file on the screen. */
                                                                    /* BL */
#include <stdio.h>                                                  /* L1 */
                                                                    /* BL */
main()                                                              /* L2 */
{                                                                   /* L3 */
 char text[500];                                                    /* L4 */
 int m, n = 0, p;                                                   /* L5 */
 puts("Type the text. The text you type form the contents");        /* L6 */
 puts("of the device-file keyboard. Strike the function");          /* L7 */
 puts("key F6 to signify the end of this file.");                   /* L8 */
                                                                    /* BL */
 m = fgetc(stdin);                                                  /* L9 */
                                                                    /* BL */
 while(m != EOF){                                                   /* L10 */
  text[n] = m;                                                      /* L11 */
  n = n + 1;                                                        /* L12 */
  m = fgetc(stdin);                                                 /* L13 */
 }                                                                  /* L14 */
```

```
                                                                    /* BL  */
 puts("Contents of device file \"keyboard\":");                     /* L15 */
                                                                    /* BL  */
 for(p = 0; p < n; p++)                                             /* L16 */
  putchar(text[p]);                                                 /* L17 */
                                                                    /* BL  */
 return(0);                                                         /* L18 */
}                                                                   /* L19 */
```

Compile and execute this program.

```
Type the text. The text you type form the contents
of device file keyboard. Strike the function key
F6 to signify the end of this file.
Chavan's Street Principle # 1    ↵
Never stand behind donkey or truck.    ↵
Donkey will kick you.    ↵
Truck will reverse and crush you.    ↵
<F6>    ↵
Contents of device file "keyboard":
Chavan's Street Principle # 1
Never stand behind donkey or truck.
Donkey will kick you.
Truck will reverse and crush you.
```

How It Works

According to C, a *file* is a transmitter or receiver of a stream of characters/bytes to or from the CPU, respectively.

In an interactive program, when you type the text, the keyboard transmits a stream of characters to the central processing unit. Hence, the keyboard fits well in C's model of a file. When program sends the output to the monitor for display, the monitor receives a stream of characters from the central processing unit. Hence, the monitor also fits well in C's model of a file. As a generic term, *device file* is used to refer to a keyboard file or a monitor file.

When you read a file or write to a file, you need a pointer to the `FILE` variable (like `fptr` used in the preceding programs). Are there any predefined pointers to `FILE` variables (like `fptr`) for the device files' keyboard and monitor? Yes, there are pointers to `FILE` constants (instead of variables) predefined for device files, as listed in Table 6-1.

Table 6-1. *Predefined Pointers to FILE Constants for Device Files*

Pointer to FILE Constants	Device File
`stdin`	Keyboard
`stdout`	Monitor
`stderr`	Monitor

Both stdout and stderr specify the same device file, i.e., monitor; but these constants are used in different contexts. To display the normal text on the monitor, you use the constant stdout, whereas to display the error messages on the monitor (e.g., the file opening failed), you use the constant stderr.

In LOC 4, you declare a char type array called text, which can accommodate 500 characters. Next, consider LOC 9, which is reproduced here for your quick reference:

```
m = fgetc(stdin);                                              /* L9 */
```

This LOC, after execution, reads a character from the file specified by the pointer to the FILE constant stdin (i.e., keyboard) and returns its ASCII value, which is assigned to m. You type three lines of text, and the first line of text is given here:

```
Chavan's Street Principle # 1
```

All these characters are transmitted to the CPU only after pressing the Enter key. After the execution of LOC 9, the first character in this line of text (it is 'C') is read by the function fgetc(), and its ASCII value (it is 67) is returned, which is assigned to variable m.

Next, there is the while loop, which spans LOCs 10 to 14. LOC 10 is reproduced here for your quick reference:

```
while(m != EOF){                                              /* L10 */
```

The continuation condition of the while loop in LOC 10 states that iterations of the while loop are permitted, while m is not equal to EOF. As the value of m is 67 and not EOF, the first iteration is permitted. Next, LOC 11 is executed, which is reproduced here for your quick reference:

```
text[n] = m;                                                  /* L11 */
```

In this LOC, the ASCII value of m (which is 67) is assigned to the first element of the array text. (Why the first element? The value of n is 0, which serves as the index of the array text.) As text is a char array, the character 'C' (as its ASCII value is 67) is stored in the first element of text. In LOC 12, the value of n is increased by 1, which serves as the index of the array text. Next, LOC 13 is executed, which is the same as LOC 9. In LOC 13, the next character available in the line of text (it is 'h') is read by the function fgetc(), and so on. When you strike the function key F6, then character Control-Z (its ASCII value is 26) is sent by the keyboard to the CPU. When the function fgetc() reads this character, then instead of returning its ASCII value, it returns the value EOF, and then iterations of the while loop are terminated.

Next, the for loop in LOCs 16 to 17 is executed, which displays the contents of the char array text on the screen.

6-22. Write Text to the Device File Monitor

Problem

You want to write text to the device file monitor.

Solution

Write a C program with the following specifications:

- The program implements the monitor using a pointer to the `FILE` constant `stdout` and also a pointer to the `FILE` constant `stderr`.
- The program reads the text from the text file `satara.txt` and writes it to the device file monitor (`stdout`).
- If the file opening or closing fails, then the program writes the error message to the device file monitor (`stderr`).

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files22.c`:

```
/* This program reads the disk-file satara.txt and writes those */
/* contents to the device-file "monitor." */
                                                                     /* BL */
#include <stdio.h>                                                   /* L1 */
                                                                     /* BL */
main()                                                               /* L2 */
{                                                                    /* L3 */
 int m, k;                                                           /* L4 */
 FILE *fptr;                                                         /* L5 */
 fptr = fopen("C:\\Code\\satara.txt", "r");                          /* L6 */
 if (fptr != NULL){                                                  /* L7 */
  puts("Disk-file kolkata.txt opened successfully.");                /* L8 */
  puts("`Its contents are now written to device file monitor:");     /* L9 */
  m = fgetc(fptr);                                                   /* L10 */
                                                                     /* BL  */
 while(m != EOF){                                                    /* L11 */
  fputc(m, stdout);                                                  /* L12 */
  m = fgetc(fptr);                                                   /* L13 */
 }                                                                   /* L14 */
                                                                     /* BL  */
 k = fclose(fptr);                                                   /* L15 */
 if(k == -1)                                                         /* L16 */
  fprintf(stderr, "Disk-file closing failed\n");                     /* L17 */
 if(k == 0)                                                   /* L18 */
  puts("Disk-file closed successfully.");                     /* L19 */
 }                                                            /* L20 */
 else                                                         /* L21 */
  fprintf(stderr, "Disk-file opening failed\n");              /* L22 */
                                                              /* BL  */
 return(0);                                                   /* L23 */
}                                                             /* L24 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
Disk-file satara.txt opened successfully.
Its contents are now written to device file monitor:
Satara is surrounded by mountains.
Satara was capital of Maratha empire for many years.
Disk-file closed successfully.
```

How It Works

In LOC 6, the disk file `satara.txt` is opened for reading. Next, consider LOC 10, which is reproduced here for your quick reference:

```
m = fgetc(fptr);                                              /* L10 */
```

In this LOC, the function `fgetc()` reads the first character in the file specified by `fptr` (it is `'S'`), and its ASCII value (it is 83) is assigned to the `int` variable `m`. Next, there is a `while` loop that spans LOCs 11 to 14. LOC 11 is reproduced here for your quick reference:

```
while(m != EOF){                                              /* L11 */
```

LOC 11 includes the continuation condition of the `while` loop, which states that iterations of the `while` loop are permitted, while `m` is not equal to `EOF`. As the value of `m` is 83, and not `EOF`, the first iteration of the `while` loop is permitted. Next, LOC 12 is executed, which is reproduced here for your quick reference:

```
fputc(m, stdout);                                             /* L12 */
```

In LOC 12, the function `fputc()` writes the character (whose ASCII value is stored in `m`) to the file specified by the pointer to the `FILE` constant `stdout`, and this file is the monitor. Thus, after the execution of LOC 12, the character `'S'` is displayed on the screen.

Next, LOC 13 is executed, which is the same as LOC 10. In this LOC, the function `fgetc()` reads the second character from the file specified by `fptr`, and so on. Proceeding in this manner, all the readable characters in the file are displayed on the screen.

6-23. Read Text from the Device File Keyboard and Write It to the Device File Monitor

Problem

You want to read the text from the device file keyboard and write it to the device file monitor.

Solution

Write a C program with the following specifications:

- The program implements the keyboard using a pointer to the `FILE` constant `stdin`.
- The program implements the monitor using a pointer to the `FILE` constant `stdout`.
- The program reads the text from the keyboard and writes it to the monitor.

The Code

The following is the code of the C program written with these specifications. Type the following C program in a text editor and save it in the folder `C:\Code` with the file name `files23.c`:

```
/* This program reads the device-file keyboard and writes those */
/* contents to the device-file monitor. */
                                                                        /* BL */
#include <stdio.h>                                                      /* L1 */
                                                                        /* BL */
main()                                                                  /* L2 */
{                                                                       /* L3 */
 char text[500];                                                        /* L4 */
 int m, n = 0, p;                                                       /* L5 */
 puts("Type the text. The text you type form the contents");           /* L6 */
 puts("of the device-file keyboard. Strike the function");             /* L7 */
 puts("key F6 to signify the end of this file.");                      /* L8 */
                                                                        /* BL */
 m = fgetc(stdin);                                                      /* L9 */
                                                                        /* BL */
 while(m != EOF){                                                       /* L10 */
  text[n] = m;                                                          /* L11 */
  n = n + 1;                                                            /* L12 */
  m = fgetc(stdin);                                                     /* L13 */
 }                                                                      /* L14 */
                                                                        /* BL  */
 puts("Contents of the device-file keyboard are now");               /* L15 */
 puts("written to the device-file monitor.");                         /* L16 */
                                                                      /* BL  */
 for(p = 0; p < n; p++)                                               /* L17 */
  fputc(text[p], stdout);                                             /* L18 */
                                                                      /* BL  */
 return(0);                                                           /* L19 */
}                                                                     /* L20 */
```

Compile and execute this program, and the following lines of text appear on the screen:

```
Type the text. The text you type form the contents
of the device-file keyboard. Strike the function
key F6 to signify the end of this file.
I am a born writer.    ↵
I inherited the art of writing from my country.    ↵
<F6>   ↵
Contents of the device-file keyboard are now
written into the device-file monitor.
I am a born writer.
I inherited the art of writing from my mother.
```

How It Works

During the execution of the program, the text typed by the user of the program is stored in the array `text`. The block of code spanning LOCs 9 to 14 reads the text from the device file keyboard and stores it in the array `text`. The block of code in LOCs 17 to 18 writes the text stored in the array `text` to the device file monitor.