Chapter 11
C File Processing

C How to Program, 8/e, GE

Objectives

In this chapter, you'll:

- Understand the concepts of files and streams.
- Create and read data using sequential-access file processing.
- Create, read and update data using random-access file processing.
- Develop a substantial transaction-processing program.
- Study Secure C programming in the context of file processing.

11.1 Introduction

- Storage of data in variables and arrays is temporary—such data is lost when a program terminates.
- Files are used for *permanent* retention of data.
- Computers store files on secondary storage devices, such as hard drives, CDs, DVDs and flash drives.
- In this chapter, we explain how data files are created, updated and processed by C programs.
- We both consider sequential-access and random-access file processing.

11.2 Files and Streams

- C views each file simply as a sequential stream of bytes (Fig. 11.1).
- Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained, administrative data structure.
- When a file is opened, a stream is associated with it.
- Three files and their associated streams are automatically opened when program execution begins—the standard input, the standard output and the standard error.
- Streams provide communication channels between files and programs.

11.2 Files and Streams (Cont.)

- For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables a program to print data on the screen.
- Opening a file returns a pointer to a `FILE` structure (defined in `<stdio.h>`) that contains information used to process the file.
- Tin some systems, this structure includes a file descriptor, i.e., an index into an operating system array called the open file table.
- Each array element contains a file control block (FCB) that the operating system uses to administer a particular file.
- The standard input, standard output and standard error are manipulated using file pointers `stdin`, `stdout` and `stderr`.

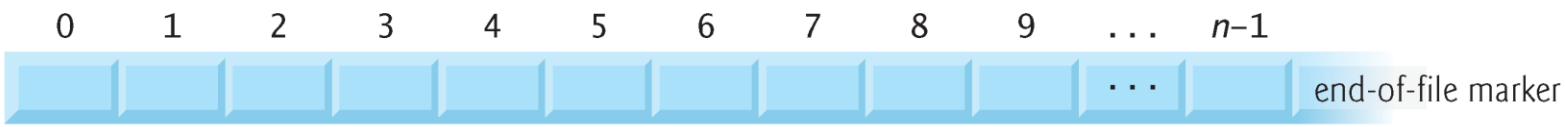

Fig. 11.1 | C's view of a file of *n* bytes.

11.2 Files and Streams (Cont.)

- The standard library provides many functions for reading data from files and for writing data to files.
- Function `fgetc`, like `getchar`, reads one character from a file.
- Function `fgetc` receives as an argument a `FILE` pointer for the file from which a character will be read.
- The call `fgetc(stdin)` reads one character from `stdin`—the standard input.
- This call is equivalent to the call `getchar()`.
- Function `fputc`, like `putchar`, writes one character to a file.
- Function `fputc` receives as arguments a character to be written and a pointer for the file to which the character will be written.

11.2 Files and Streams (Cont.)

- The function call `fputc('a', stdout)` writes the character `'a'` to `stdout`—the standard output.
- This call is equivalent to `putchar('a')`.
- Several other functions used to read data from standard input and write data to standard output have similarly named file-processing functions.
- The `fgets` and `fputs` functions, for example, can be used to *read a line from a file* and *write a line to a file*, respectively.
- In the next several sections, we introduce the file-processing equivalents of functions `scanf` and `printf`—`fscanf` and `fprintf`.

11.3 Creating a Sequential-Access File

- C imposes no structure on a file.
- Thus, notions such as a record of a file do not exist as part of the C language.
- The following example shows how you can impose your own record structure on a file.
- Figure 11.2 creates a simple sequential-access file that might be used in an accounts receivable system to help keep track of the amounts owed by a company's credit clients.

11.3 Creating a Sequential-Access File (Cont.)

- For each client, the program obtains an *account number*, the *client's name* and the *client's balance* (i.e., the amount the client owes the company for goods and services received in the past).
- The data obtained for each client constitutes a "record" for that client.
- The account number is used as the record key in this application—the file will be created and maintained in account-number order.

11.3 Creating a Sequential-Access File (Cont.)

- This program assumes the user enters the records in account-number order.
- In a comprehensive accounts receivable system, a sorting capability would be provided so the user could enter the records in any order.
- The records would then be sorted and written to the file.
- [*Note:* Figures 11.6–11.7 use the data file created in Fig. 11.2, so you must run Fig. 11.2 before Figs. 11.6–11.7.]

```c
// Fig. 11.2: fig11_02.c
// Creating a sequential file
#include <stdio.h>

int main(void)
{
   FILE *cfPtr; // cfPtr = clients.txt file pointer

   // fopen opens file. Exit program if unable to create file
   if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
      puts("File could not be opened");
   }
   else {
      puts("Enter the account, name, and balance.");
      puts("Enter EOF to end input.");
      printf("%s", "? ");

      unsigned int account; // account number
      char name[30]; // account name
      double balance; // account balance

      scanf("%d%29s%lf", &account, name, &balance);
```

Fig. 11.2 | Creating a sequential file. (Part 1 of 2.)

```
23
24          // write account, name and balance into file with fprintf
25          while (!feof(stdin)  ) {
26             fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
27             printf("%s", "? ");
28             scanf("%d%29s%lf", &account, name, &balance);
29          }
30
31          fclose(cfPtr); // fclose closes file
32       }
33    }
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 11.2 | Creating a sequential file. (Part 2 of 2.)

11.3 Creating a Sequential-Access File (Cont.)

- Now let's examine this program.
- `cfptr` is a *pointer to a `FILE` structure*.
- A C program administers each file with a separate `FILE` structure.
- You need not know the specifics of the `FILE` structure to use files, but you can study the declaration in `stdio.h` if you like.
- We'll soon see precisely how the `FILE` structure leads indirectly to the operating system's file control block (FCB) for a file.
- Each open file must have a separately declared pointer of type `FILE` that's used to refer to the file.

11.3 Creating a Sequential-Access File (Cont.)

- The file name—`"clients.dat"`—is used by the program and establishes a "line of communication" with the file.
- The file pointer `cfPtr` is assigned *a pointer to the `FILE` structure* for the file opened with `fopen`.
- Function `fopen` takes two arguments: a filename (which can include path information leading to the file's location) and a file open mode.
- The file open mode `"w"` indicates that the file is to be opened for writing.
- If a file *does not* exist and it's opened for writing, `fopen` creates the file.

11.3 Creating a Sequential-Access File (Cont.)

- If an existing file is opened for writing, the contents of the file are *discarded without warning*.
- In the program, the `if` statement is used to determine whether the file pointer `cfPtr` is `NULL` (i.e., the file is not opened).
- If it's `NULL`, the program prints an error message and terminates.
- Otherwise, the program processes the input and writes it to the file.

Common Programming Error 11.1

Opening an existing file for writing ("w") when, in fact, the user wants to preserve the file, discards the contents of the file without warning.

Common Programming Error 11.2

Forgetting to open a file before attempting to reference it in a program is a logic error.

11.3 Creating a Sequential-Access File (Cont.)

- The program prompts the user to enter the fields for each record or to enter *end-of-file* when data entry is complete.
- Figure 11.3 lists the key combinations for entering end-of-file for various computer systems.
- Function `feof` to determine whether the end-of-file indicator is set for the file to which `stdin` refers.
- The *end-of-file* indicator informs the program that there's no more data to be processed.
- In Fig. 11.2, the *end-of-file indicator* is set for the standard input when the user enters the end-of-file key combination.
- The argument to function `feof` is a pointer to the file being tested for the end-of-file indicator (`stdin` in this case).

Operating system	Key combination
Linux/Mac OS X/UNIX	*<Ctrl> d*
Windows	*<Ctrl> z* then press *Enter*

Fig. 11.3 | End-of-file key combinations for various popular operating systems.

11.3 Creating a Sequential-Access File (Cont.)

- The function returns a nonzero (true) value when the end-of-file indicator has been set; otherwise, the function returns zero.
- The `while` statement that includes the `feof` call in this program continues executing while the end-of-file indicator is not set.
- The data may be retrieved later by a program designed to read the file (see Section 11.4).

11.3 Creating a Sequential-Access File (Cont.)

- Function `fprintf` is equivalent to `printf` except that `fprintf` also receives as an argument a file pointer for the file to which the data will be written.
- Function `fprintf` can output data to the standard output by using `stdout` as the file pointer, as in:
  - `fprintf(stdout, "%d %s %.2f\n", account, name, balance);`

11.3 Creating a Sequential-Access File (Cont.)

- After the user enters end-of-file, the program closes the `clients.dat` file with `fclose` and terminates.
- Function `fclose` also receives the file pointer (rather than the filename) as an argument.
- *If function `fclose` is not called explicitly, the operating system normally will close the file when program execution terminates.*
- This is an example of operating system "housekeeping."

Performance Tip 11.1

Closing a file can free resources for which other users or programs may be waiting, so you should close each file as soon as it's no longer needed rather than waiting for the operating system to close it at program termination.

11.3 Creating a Sequential-Access File (Cont.)

- In the sample execution for the program of Fig. 11.3, the user enters information for five accounts, then enters end-of-file to signal that data entry is complete.
- The sample execution does not show how the data records actually appear in the file.
- To verify that the file has been created successfully, in the next section we present a program that reads the file and prints its contents.
- Figure 11.4 illustrates the relationship between `FILE` pointers, `FILE` structures and FCBs.
- When the file `"clients.dat"` is opened, an FCB for the file is copied into memory.

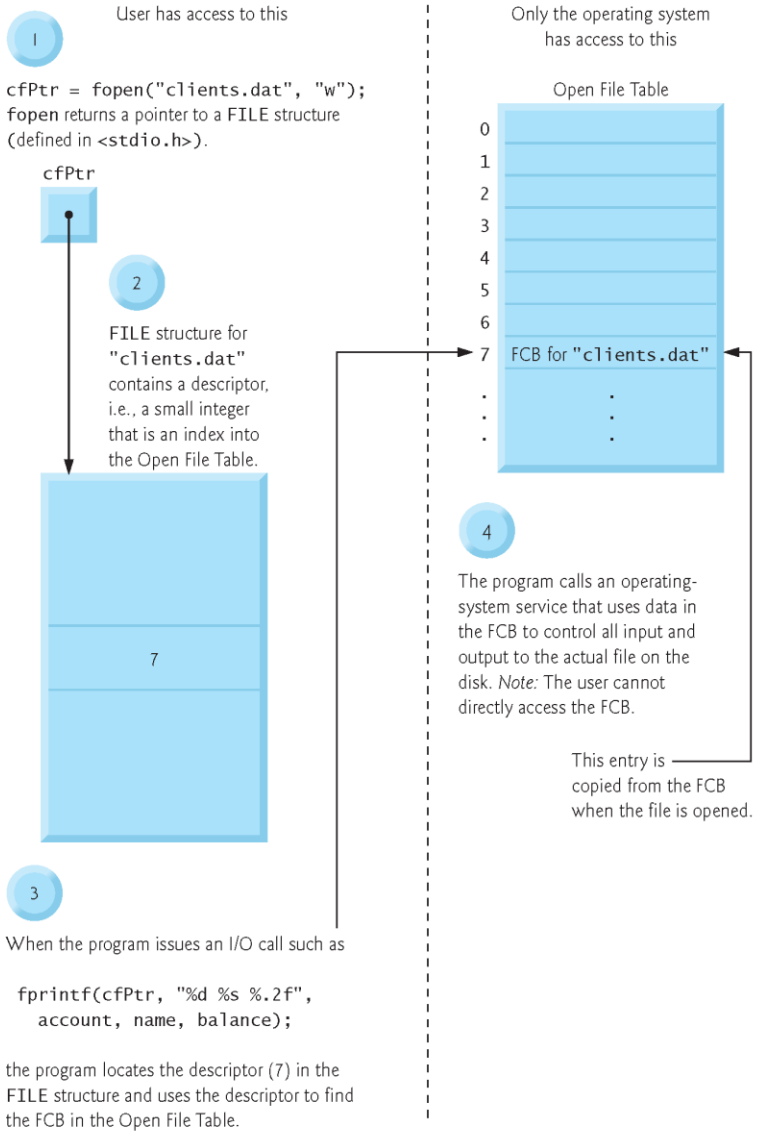

Fig. 11.4 | Relationship between `FILE` pointers, `FILE` structures and FCBs.

11.3 Creating a Sequential-Access File (Cont.)

- The figure shows the connection between the file pointer returned by `fopen` and the FCB used by the operating system to administer the file.
- Programs may process no files, one file or several files.
- Each file used in a program will have a different file pointer returned by `fopen`.
- *All subsequent file-processing functions after the file is opened must refer to the file with the appropriate file pointer.*
- Files may be opened in one of several modes (Fig. 11.5).
- To create a file, or to discard the contents of a file before writing data, open the file for writing (`"w"`).

11.3 Creating a Sequential-Access File (Cont.)

- To read an existing file, open it for reading (`"r"`).
- To add records to the end of an existing file, open the file for appending (`"a"`).
- To open a file so that it may be written to and read from, open the file for updating in one of the three update modes—`"r+"`, `"w+"` or `"a+"`.
- Mode `"r+"` opens an existing file for reading and writing.
- Mode `"w+"` creates a file for reading and writing.
- If the file already exists, it's opened and its current contents are discarded.

11.3 Creating a Sequential-Access File (Cont.)

- Mode `"a+"` opens a file for reading and writing—all writing is done at the end of the file.
- If the file does not exist, it's created.
- Each file open mode has a corresponding binary mode (containing the letter `b`) for manipulating binary files.
- The binary modes are used in Sections 11.5–11.9 when we introduce random-access files.
- In addition, C11 provides *exclusive* write mode, which you indicate by adding an `x` to the end of the `w`, `w+`, `wb` or `wb+` modes.

11.3 Creating a Sequential-Access File (Cont.)

- In exclusive write mode, `fopen` will fail if the file already exists or cannot be created.
- If opening a file in exclusive write mode is successful and the underlying system supports exclusive file access, then only your program can access the file while it's open.
- (Some compilers and platforms do not support exclusive write mode.)
- If an error occurs while opening a file in any mode, `fopen` returns `NULL`.

Mode	Description
`r`	Open an existing file for reading.
`w`	Create a file for writing. If the file already exists, *discard* the current contents.
`a`	Open or create a file for writing at the end of the file—i.e., write operations *append* data to the file.
`r+`	Open an existing file for update (reading and writing).
`w+`	Create a file for reading and writing. If the file already exists, *discard* the current contents.
`a+`	Open or create a file for reading and updating; all writing is done at the end of the file—i.e., write operations *append* data to the file.

Fig. 11.5 | File opening modes. (Part 1 of 2.)

Mode	Description
`rb`	Open an existing file for reading in binary mode.
`wb`	Create a file for writing in binary mode. If the file already exists, discard the current contents.
`ab`	Append: open or create a file for writing at the end of the file in binary mode.
`rb+`	Open an existing file for update (reading and writing) in binary mode.
`wb+`	Create a file for update in binary mode. If the file already exists, discard the current contents.
`ab+`	Append: open or create a file for update in binary mode; writing is done at the end of the file.

Fig. 11.5 | File opening modes. (Part 2 of 2.)

Common Programming Error 11.3

Opening a nonexistent file for reading is an error.

Common Programming Error 11.4

Opening a file for reading or writing without having been granted the appropriate access rights to the file (this is operating-system dependent) is an error.

Common Programming Error 11.5

Opening a file for writing when no space is available is a runtime error.

Common Programming Error 11.6

Opening a file in write mode ("w") when it should be opened in update mode ("r+") causes the contents of the file to be discarded.

Error-Prevention Tip 11.1

Open a file only for reading (and not updating) if its contents should not be modified. This prevents unintentional modification of the file's contents. This is another example of the principle of least privilege.

11.4 Reading Data from a Sequential-Access File

- Data is stored in files so that the data can be retrieved for processing when needed.
- The previous section demonstrated how to create a file for sequential access.
- This section shows how to read data sequentially from a file.
- Figure 11.6 reads records from the file `"clients.dat"` created by the program of Fig. 11.2 and prints their contents.
- `cfPtr` is a pointer to a `FILE`.
- We attempt to open the file `"clients.dat"` for reading (`"r"`) and determine whether it opened successfully (i.e., `fopen` does *not* return `NULL`).

11.4 Reading Data from a Sequential-Access File (Cont.)

- Read a "record" from the file.
  - Function `fscanf` is equivalent to `scanf`, except `fscanf` receives a file pointer for the file being read.
- After this statement executes the first time, `account` will have the value `100`, `name` will have the value `"Jones"` and `balance` will have the value `24.98`.
- Each time the second `fscanf` statement executes, the program reads another record from the file and `account`, `name` and `balance` take on new values.
- When the program reaches the end of the file, the file is closed and the program terminates.
- Function `feof` returns true only *after* the program attempts to read the nonexistent data following the last line.

```c
// Fig. 11.6: fig11_06.c
// Reading and printing a sequential file
#include <stdio.h>

int main(void)
{
   FILE *cfPtr; // cfPtr = clients.txt file pointer

   // fopen opens file; exits program if file cannot be opened
   if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
      puts("File could not be opened");
   }
   else { // read account, name and balance from file
      unsigned int account; // account number
      char name[30]; // account name
      double balance; // account balance

      printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
      fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);

```

Fig. 11.6 | Reading and printing a sequential file. (Part 1 of 2.)

```
21          // while not end of file
22          while (!feof(cfPtr)  ) {
23             printf("%-10d%-13s%7.2f\n", account, name, balance);
24             fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
25          }
26
27          fclose(cfPtr); // fclose closes the file
28       }
29    }
```

```
Account    Name          Balance
100        Jones           24.98
200        Doe            345.67
300        White            0.00
400        Stone          -42.16
500        Rich           224.62
```

Fig. 11.6 | Reading and printing a sequential file. (Part 2 of 2.)

11.4 Reading Data from a Sequential-Access File (Cont.)

Resetting the File Position Pointer

- To retrieve data sequentially from a file, a program normally starts reading from the beginning of the file and reads all data consecutively until the desired data is found.
- It may be desirable to process the data sequentially in a file several times (from the beginning of the file) during the execution of a program.

11.4 Reading Data from a Sequential-Access File (Cont.)

- The statement
  - `rewind(cfPtr);`

  causes a program's file position pointer—which indicates the number of the next byte in the file to be read or written—to be repositioned to the *beginning* of the file (i.e., byte 0) pointed to by `cfPtr`.
- The file position pointer is not really a pointer.
- Rather it's an integer value that specifies the byte in the file at which the next read or write is to occur.
- This is sometimes referred to as the file offset.
- The file position pointer is a member of the `FILE` structure associated with each file.

11.4 Reading Data from a Sequential-Access File (Cont.)

Credit Inquiry Program

- The program of Fig. 11.7 allows a credit manager to obtain lists of customers with zero balances (i.e., customers who do not owe any money), customers with credit balances (i.e., customers to whom the company owes money) and customers with debit balances (i.e., customers who owe the company money for goods and services received).
- A credit balance is a *negative* amount; a debit balance is a *positive* amount.

```c
// Fig. 11.7: fig11_07.c
// Credit inquiry program
#include <stdio.h>

// function main begins program execution
int main(void)
{
   FILE *cfPtr; // clients.txt file pointer

   // fopen opens the file; exits program if file cannot be opened
   if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
      puts("File could not be opened");
   }
   else {

      // display request options
      printf("%s", "Enter request\n"
         " 1 - List accounts with zero balances\n"
         " 2 - List accounts with credit balances\n"
         " 3 - List accounts with debit balances\n"
         " 4 - End of run\n? ");
      unsigned int request; // request number
      scanf("%u", &request);

```

Fig. 11.7 | Credit inquiry program. (Part 1 of 6.)

```
25          // process user's request
26          while (request != 4) {
27             unsigned int account; // account number
28             double balance; // account balance
29             char name[30]; // account name
30
31             // read account, name and balance from file
32             fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
33
```

Fig. 11.7 | Credit inquiry program. (Part 2 of 6.)

```
34            switch (request) {
35               case 1:
36                  puts("\nAccounts with zero balances:");
37
38                  // read file contents (until eof)
39                  while (!feof(cfPtr)) {
40                     // output only if balance is 0
41                     if (balance == 0) {
42                        printf("%-10d%-13s%7.2f\n",
43                           account, name, balance);
44                     }
45
46                     // read account, name and balance from file
47                     fscanf(cfPtr, "%d%29s%lf",
48                        &account, name, &balance);
49                  }
50
51                  break;
```

Fig. 11.7 | Credit inquiry program. (Part 3 of 6.)

```
52                case 2:
53                   puts("\nAccounts with credit balances:\n");
54
55                   // read file contents (until eof)
56                   while (!feof(cfPtr)) {
57                      // output only if balance is less than 0
58                      if (balance < 0) {
59                         printf("%-10d%-13s%7.2f\n",
60                            account, name, balance);
61                      }
62
63                      // read account, name and balance from file
64                      fscanf(cfPtr, "%d%29s%lf",
65                         &account, name, &balance);
66                   }
67
68                   break;
```

Fig. 11.7 | Credit inquiry program. (Part 4 of 6.)

```
69              case 3:
70                 puts("\nAccounts with debit balances:\n");
71
72                 // read file contents (until eof)
73                 while (!feof(cfPtr)) {
74                    // output only if balance is greater than 0
75                    if (balance > 0) {
76                       printf("%-10d%-13s%7.2f\n",
77                          account, name, balance);
78                    }
79
80                    // read account, name and balance from file
81                    fscanf(cfPtr, "%d%29s%lf",
82                       &account, name, &balance);
83                 }
84
85                 break;
86           }
87
88           rewind(cfPtr); // return cfPtr to beginning of file
89
90           printf("%s", "\n? ");
91           scanf("%d", &request);
92        }
```

Fig. 11.7 | Credit inquiry program. (Part 5 of 6.)

```
93
94          puts("End of run.");
95          fclose(cfPtr); // fclose closes the file
96       }
97    }
```

Fig. 11.7 | Credit inquiry program. (Part 6 of 6.)

11.4 Reading Data from a Sequential-Access File (Cont.)

- The program displays a menu and allows the credit manager to enter one of three options to obtain credit information.
- Option 1 produces a list of accounts with zero balances.
- Option 2 produces a list of accounts with *credit balances*.
- Option 3 produces a list of accounts with *debit balances*.
- Option 4 terminates program execution.
- A sample output is shown in Fig. 11.8.

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

? 2

Accounts with credit balances:
400       Stone         -42.16

? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

? 4
End of run.
```

Fig. 11.8 | Sample output of the credit inquiry program of Fig. 11.7.

11.4 Reading Data from a Sequential-Access File (Cont.)

- Data in this type of sequential file cannot be modified without the risk of destroying other data.
- For example, if the name "`White`" needs to be changed to "`Worthington`," the old name cannot simply be overwritten.
- The record for `White` was written to the file as

11.4 Reading Data from a Sequential-Access File (Cont.)

- If the record is rewritten beginning at the same location in the file using the new name, the record will be
  - `300 Worthington 0.00`
- The new record is larger (has more characters) than the original record.
- The characters beyond the second "`o`" in "`Worthington`" will *overwrite* the beginning of the next sequential record in the file.
- The problem here is that in the formatted input/output model using `fprintf` and `fscanf`, fields—and hence records—can vary in size.

11.4 Reading Data from a Sequential-Access File (Cont.)

- For example, the values 7, 14, –117, 2074 and 27383 are all `int`s stored in the same number of bytes internally, but they're different-sized fields when displayed on the screen or written to a file as text.
- Therefore, sequential access with `fprintf` and `fscanf` is *not* usually used to *update records in place*.
- Instead, the entire file is usually *rewritten*.

11.4 Reading Data from a Sequential-Access File (Cont.)

- To make the preceding name change, the records before `300 White 0.00` in such a sequential-access file would be copied to a new file, the new record would be written and the records after `300 White 0.00` would be copied to the new file.
- This requires processing every record in the file to update one record.

11.5 Random-Access Files

- As we stated previously, records in a file created with the formatted output function `fprintf` are not necessarily the same length.
- However, individual records of a random-access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.
- This makes random-access files appropriate for airline reservation systems, banking systems, point-of-sale systems, and other kinds of transaction-processing systems that require rapid access to specific data.

11.5 Random-Access Files (Cont.)

- There are other ways of implementing random-access files, but we'll limit our discussion to this straightforward approach using fixed-length records.
- Because every record in a random-access file normally has the same length, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key.
- We'll soon see how this facilitates *immediate* access to specific records, even in large files.
- Figure 11.9 illustrates one way to implement a random-access file.
- Such a file is like a freight train with many cars—some empty and some with cargo.

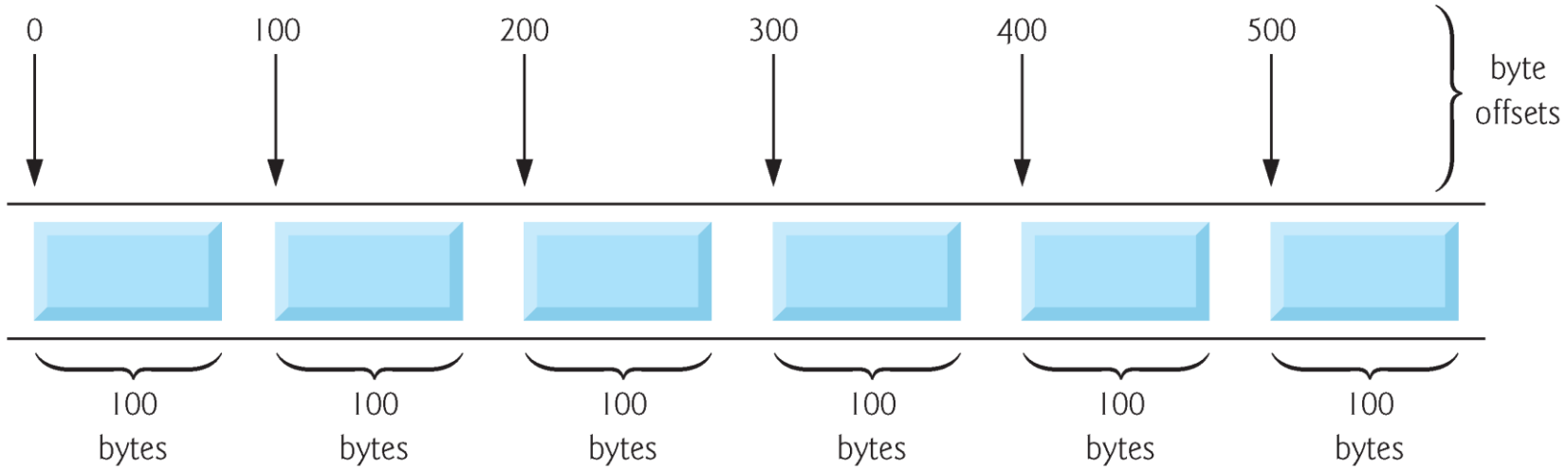

Fig. 11.9 | C's view of a random-access file.

11.5 Random-Access Files (Cont.)

- Fixed-length records enable data to be inserted in a random-access file *without destroying other data in the file*.
- Data stored previously can also be updated or deleted without rewriting the entire file.

11.6 Creating a Random-Access File

- Function `fwrite` transfers a specified number of bytes beginning at a specified location in memory to a file.
- The data is written beginning at the location in the file indicated by the file position pointer.
- Function `fread` transfers a specified number of bytes from the location in the file specified by the file position pointer to an area in memory beginning with a specified address.

11.6 Creating a Random-Access File (Cont.)

- Now, when writing an integer, instead of using
  - `fprintf(fPtr, "%d", number);`

  which could print a single digit or as many as 11 digits (10 digits plus a sign, each of which requires 1 byte of storage) for a four-byte integer, we can use
  - `fwrite(&number, sizeof(int), 1, fPtr);`

  which *always* writes four bytes on a system with four-byte integers from a variable `number` to the file represented by `fPtr` (we'll explain the `1` argument shortly).

11.6 Creating a Random-Access File (Cont.)

- Later, `fread` can be used to read those four bytes into an integer variable `number`.
- Although `fread` and `fwrite` read and write data, such as integers, in fixed-size rather than variable-size format, the data they handle are processed in computer "raw data" format (i.e., bytes of data) rather than in `printf`'s and `scanf`'s human-readable text format.
- Because the "raw" representation of data is system dependent, "raw data" may not be readable on other systems, or by programs produced by other compilers or with other compiler options.

11.6 Creating a Random-Access File (Cont.)

- Functions `fwrite` and `fread` are capable of reading and writing arrays of data to and from disk.
- The third argument of both `fread` and `fwrite` is the number of elements in the array that should be read from or written to disk.
- The preceding `fwrite` function call writes a single integer to disk, so the third argument is `1` (as if one element of an array is being written).
- File-processing programs rarely write a single field to a file.
- Normally, they write one `struct` at a time, as we show in the following examples.

11.6 Creating a Random-Access File (Cont.)

- Consider the following problem statement:
  - Create a credit-processing system capable of storing up to 100 fixed-length records. Each record should consist of an account number that will be used as the record key, a last name, a first name and a balance. The resulting program should be able to update an account, insert a new account record, delete an account and list all the account records in a formatted text file for printing. Use a random-access file.
- The next several sections introduce the techniques necessary to create the credit-processing program.

11.7 Creating a Random-Access File (Cont.)

- Figure 11.10 shows how to open a random-access file, define a record format using a `struct`, write data to the disk and close the file.
- This program initializes all 100 records of the file `"credit.dat"` with empty `struct`s using the function `fwrite`.
- Each empty `struct` contains `0` for the account number, `""` (the empty string) for the last name, `""` for the first name and `0.0` for the balance.
- The file is initialized in this manner to create space on the disk in which the file will be stored and to make it possible to determine whether a record contains data.

```c
// Fig. 11.10: fig11_10.c
// Creating a random-access file sequentially
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
};

int main(void)
{
   FILE *cfPtr; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("accounts.dat", "wb")) == NULL) {
      puts("File could not be opened.");
   }
```

Fig. 11.10 | Creating a random-access file sequentially. (Part 1 of 2.)

```
21      else {
22         // create clientData with default information
23         struct clientData blankClient = {0, "", "", 0.0};
24
25         // output 100 blank records to file
26         for (unsigned int i = 1; i <= 100; ++i) {
27            fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
28         }
29
30         fclose (cfPtr); // fclose closes the file
31      }
32   }
```

Fig. 11.10 | Creating a random-access file sequentially. (Part 2 of 2.)

11.6 Creating a Random-Access File (Cont.)

- Function `fwrite` writes a block bytes to a file.
- Line 27 causes the structure `blankClient` of size `sizeof(struct clientData)` to be written to the file pointed to by `cfPtr`.
- The operator `sizeof` returns the size in bytes of its operand in parentheses (in this case `struct clientData`).
- Function `fwrite` can actually be used to write several elements of an array of objects.

11.6 Creating a Random-Access File (Cont.)

- To do so, supply in the call to `fwrite` a pointer to an array as the first argument and the number of elements to be written as the third argument.
- In the preceding statement, `fwrite` was used to write a single object that was not an array element.

11.6 Creating a Random-Access File (Cont.)

- Writing a single object is equivalent to writing one element of an array, hence the `1` in the `fwrite` call.
- [*Note:* Figures 11.11, 11.14 and 11.15 use the data file created in Fig. 11.10, so you must run Fig. 11.10 before Figs. 11.11, 11.14 and 11.15]

11.7 Writing Data Randomly to a Random-Access File

- Figure 11.11 writes data to the file `"credit.dat"`.
- It uses the combination of `fseek` and `fwrite` to store data at specific locations in the file.
- Function `fseek` sets the file position pointer to a specific position in the file, then `fwrite` writes the data.
- A sample execution is shown in Fig. 11.12.

```c
// Fig. 11.11: fig11_11.c
// Writing data randomly to a random-access file
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
}; // end structure clientData

int main(void)
{
   FILE *cfPtr; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL) {
      puts("File could not be opened.");
   }
   else {
      // create clientData with default information
      struct clientData client = {0, "", "", 0.0};

```

Fig. 11.11 | Writing data randomly to a random-access file. (Part 1 of 3.)

```
25          // require user to specify account number
26          printf("%s", "Enter account number"
27             " (1 to 100, 0 to end input): ");
28          scanf("%d", &client.acctNum);
29
30          // user enters information, which is copied into file
31          while (client.acctNum != 0) {
32             // user enters last name, first name and balance
33             printf("%s", "\nEnter lastname, firstname, balance: ");
34
35             // set record lastName, firstName and balance value
36             fscanf(stdin, "%14s%9s%lf", client.lastName,
37                client.firstName, &client.balance);
38
39             // seek position in file to user-specified record
40             fseek(cfPtr, (client.acctNum - 1) *
41                sizeof(struct clientData), SEEK_SET);
42
43             // write user-specified information in file
44             fwrite(&client, sizeof(struct clientData), 1, cfPtr);
45
```

Fig. 11.11 | Writing data randomly to a random-access file. (Part 2 of 3.)

```
46            // enable user to input another account number
47            printf("%s", "\nEnter account number: ");
48            scanf("%d", &client.acctNum);
49         }
50
51         fclose(cfPtr); // fclose closes the file
52      }
53   }
```

Fig. 11.11 | Writing data randomly to a random-access file. (Part 3 of 3.)

```
Enter account number (1 to 100, 0 to end input): 37
Enter lastname, firstname, balance: Barker Doug 0.00
Enter account number: 29
Enter lastname, firstname, balance: Brown Nancy -24.54
Enter account number: 96
Enter lastname, firstname, balance: Stone Sam 34.98
Enter account number: 88
Enter lastname, firstname, balance: Smith Dave 258.34
Enter account number: 33
Enter lastname, firstname, balance: Dunn Stacey 314.33
Enter account number: 0
```

Fig. 11.12 | Sample execution of the program in Fig. 11.11.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- Lines 40–41 position the file position pointer for the file referenced by `cfPtr` to the byte location calculated by `(client.accountNum - 1) * sizeof(struct clientData)`.
- The value of this expression is called the offset or the displacement.
- Because the account number is between 1 and 100 but the byte positions in the file start with 0, 1 is subtracted from the account number when calculating the byte location of the record.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- Thus, for record 1, the file position pointer is set to byte 0 of the file.
- The symbolic constant `SEEK_SET` indicates that the file position pointer is positioned relative to the beginning of the file by the amount of the offset.
- As the above statement indicates, a seek for account number 1 in the file sets the file position pointer to the beginning of the file because the byte location calculated is 0.
- Figure 11.13 illustrates the file pointer referring to a `FILE` structure in memory.
- The file position pointer here indicates that the next byte to be read or written is 5 bytes from the beginning of the file.

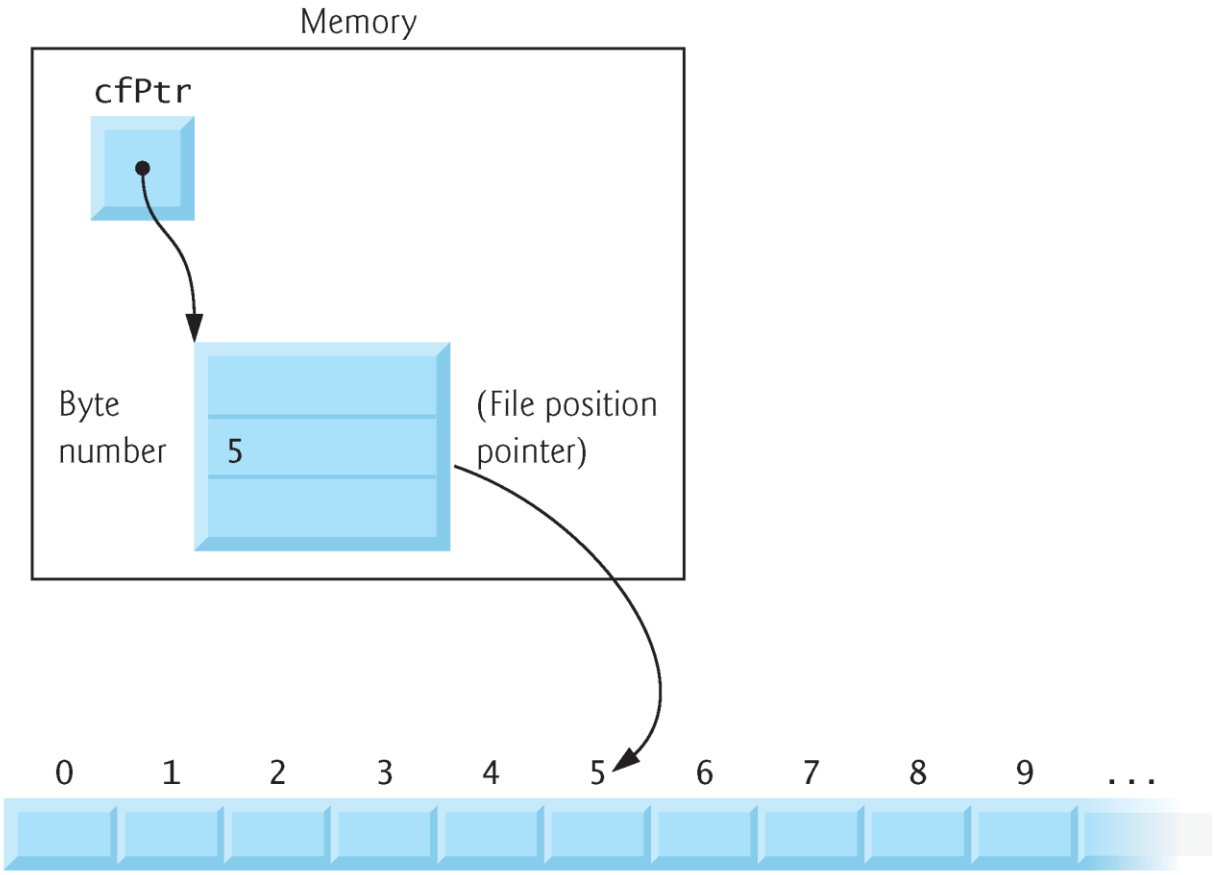

Fig. 11.13 | File position pointer indicating an offset of 5 bytes from the beginning of the file.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- The function prototype for `fseek` is
  - `int fseek(FILE *stream, long int offset, int whence);`
- where `offset` is the number of bytes to seek from `whence` in the file pointed to by `stream`—a positive `offset` seeks forward and a negative one seeks backward.
- Argument `whence` is one of the values `SEEK_SET`, `SEEK_CUR` or `SEEK_END` (all defined in `<stdio.h>`), which indicate the location from which the seek begins.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- `SEEK_SET` indicates that the seek starts at the *beginning* of the file; `SEEK_CUR` indicates that the seek starts at the *current location* in the file; and `SEEK_END` indicates that the seek starts at the *end* of the file.
- For simplicity, the programs in this chapter do not perform error checking.
- Industrial-strength programs should determine whether functions such as `fscanf`, `fseek` and `fwrite` operate correctly by checking their return values.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- Function `fscanf` returns the number of data items successfully read or the value `EOF` if a problem occurs while reading data.
- Function `fseek` returns a nonzero value if the seek operation cannot be performed.
- Function `fwrite` returns the number of items it successfully output.
- If this number is less than the *third argument* in the function call, then a write error occurred.

11.8 Reading Data from a Random-Access File

- Function `fread` reads a specified number of bytes from a file into memory.
- For example,
  - `fread(&client, sizeof(struct clientData), 1, cfPtr);`

  reads the number of bytes determined by `sizeof(struct clientData)` from the file referenced by `cfPtr`, stores the data in `client` and returns the number of bytes read.
- The bytes are read from the location specified by the file position pointer.

11.8 Reading Data from a Random-Access File (Cont.)

- Function `fread` can read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read.
- The preceding statement reads *one* element should be read.
- To read *more than one*, specify the number of elements as `fread`'s third argument.
- Function `fread` returns the number of items it successfully input.

11.8 Reading Data from a Random-Access File (Cont.)

- If this number is less than the third argument in the function call, then a read error occurred.
- Figure 11.14 reads sequentially every record in the `"credit.dat"` file, determines whether each record contains data and displays the formatted data for records containing data.
- Function `feof` determines when the end of the file is reached, and the `fread` function transfers data from the file to the `clientData` structure `client`.

```c
// Fig. 11.14: fig11_14.c
// Reading a random-access file sequentially
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
};

int main(void)
{
   FILE *cfPtr; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("credit.txt", "rb")) == NULL) {
      puts("File could not be opened.");
   }
```

Fig. 11.14 | Reading a random-access file sequentially. (Part 1 of 3.)

```
21      else {
22         printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
23            "First Name", "Balance");
24
25         // read all records from file (until eof)
26         while (!feof(cfPtr)) {
27            // create clientData with default information
28            struct clientData client = {0, "", "", 0.0};
29
30            int result = fread(&client, sizeof(struct clientData), 1, cfPtr);
31
32            // display record
33            if (result != 0 && client.acctNum != 0) {
34               printf("%-6d%-16s%-11s%10.2f\n",
35                  client.acctNum, client.lastName,
36                  client.firstName, client.balance);
37            }
38         }
39
40         fclose(cfPtr); // fclose closes the file
41      }
42   }
```

Fig. 11.14 | Reading a random-access file sequentially. (Part 2 of 3.)

```
Acct  Last Name       First Name    Balance
29    Brown           Nancy          -24.54
33    Dunn            Stacey         314.33
37    Barker          Doug             0.00
88    Smith           Dave           258.34
96    Stone           Sam             34.98
```

Fig. 11.14 | Reading a random-access file sequentially. (Part 3 of 3.)

11.9 Case Study: Transaction-Processing Program

- We now present a substantial transaction-processing program (Fig. 11.15) using random-access files.
- The program maintains a bank's account information—updating existing accounts, adding new accounts, deleting accounts and storing a listing of all the current accounts in a text file for printing.
- We assume that the program of Fig. 11.10 has been executed to create the file `credit.dat`.

11.9 Case Study: Transaction-Processing Program (Cont.)

- The program has five options.
- Option 1 calls function `textFile` to store a formatted list of all the accounts (typically called a report) in a text file called `accounts.txt` that may be printed later.
- The function uses `fread` and the sequential file access techniques used in the program of Fig. 11.14.

11.9 Case Study: Transaction-Processing Program (Cont.)

- Option 2 calls the function `updateRecord` to update an account.
- The function will update only a record that already exists, so the function first checks whether the record specified by the user is empty.
- The record is read into structure `client` with `fread`, then member `acctNum` is compared to 0.
- If it's 0, the record contains no information, and a message is printed stating that the record is empty.
- Then the menu choices are displayed.
- If the record contains information, function `updateRecord` inputs the transaction amount, calculates the new balance and rewrites the record to the file.

11.9 Case Study: Transaction-Processing Program (Cont.)

- Option 3 calls the function `newRecord` to add a new account to the file.
- If the user enters an account number for an existing account, `newRecord` displays an error message indicating that the record already contains information, and the menu choices are printed again.
- This function uses the same process to add a new account as does the program in Fig. 11.11.

11.9 Case Study: Transaction-Processing Program (Cont.)

- Option 4 calls function `deleteRecord` to delete a record from the file.
- Deletion is accomplished by asking the user for the account number and reinitializing the record.
- If the account contains no information, `deleteRecord` displays an error message indicating that the account does not exist.
- Option 5 terminates program execution.
- The program is shown in Fig. 11.15.
- The file `"credit.dat"` is opened for update (reading and writing) using `"rb+"` mode.

```c
// Fig. 11.15: fig11_15.c
// Transaction-processing program reads a random-access file sequentially,
// updates data already written to the file, creates new data to
// be placed in the file, and deletes data previously stored in the file.
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
};

// prototypes
unsigned int enterChoice(void);
void textFile(FILE *readPtr);
void updateRecord(FILE *fPtr);
void newRecord(FILE *fPtr);
void deleteRecord(FILE *fPtr);

```

Fig. 11.15 | Transaction-processing program. (Part 1 of 11.)

```
22  int main(void)
23  {
24     FILE *cfPtr; // accounts.dat file pointer
25
26     // fopen opens the file; exits if file cannot be opened
27     if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL) {
28        puts("File could not be opened.");
29     }
30     else {
31        unsigned int choice; // user's choice
32
33        // enable user to specify action
34        while ((choice = enterChoice()) != 5) {
35           switch (choice) {
36              // create text file from record file
37              case 1:
38                 textFile(cfPtr);
39                 break;
```

Fig. 11.15 | Transaction-processing program. (Part 2 of 11.)

```
40                // update record
41                case 2:
42                   updateRecord(cfPtr);
43                   break;
44                // create record
45                case 3:
46                   newRecord(cfPtr);
47                   break;
48                // delete existing record
49                case 4:
50                   deleteRecord(cfPtr);
51                   break;
52                // display message if user does not select valid choice
53                default:
54                   puts("Incorrect choice");
55                   break;
56             }
57          }
58
59          fclose(cfPtr); // fclose closes the file
60       }
61    }
62
```

Fig. 11.15 | Transaction-processing program. (Part 3 of 11.)

```
63  // create formatted text file for printing
64  void textFile(FILE *readPtr)
65  {
66     FILE *writePtr; // accounts.txt file pointer
67
68     // fopen opens the file; exits if file cannot be opened
69     if ((writePtr = fopen("accounts.txt", "w")  ) == NULL) {
70        puts("File could not be opened.");
71     }
72     else {
73        rewind(readPtr); // sets pointer to beginning of file
74        fprintf(writePtr, "%-6s%-16s%-11s%10s\n",
75           "Acct", "Last Name", "First Name","Balance");
76
```

Fig. 11.15 | Transaction-processing program. (Part 4 of 11.)

```
77          // copy all records from random-access file into text file
78          while (!feof(readPtr)) {
79             // create clientData with default information
80             struct clientData client = { 0, "", "", 0.0 };
81             int result =
82                fread(&client, sizeof(struct clientData), 1, readPtr);
83
84             // write single record to text file
85             if (result != 0 && client.acctNum != 0) {
86                fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n",
87                   client.acctNum, client.lastName,
88                   client.firstName, client.balance);
89             }
90          }
91
92          fclose(writePtr); // fclose closes the file
93       }
94    }
95
```

Fig. 11.15 | Transaction-processing program. (Part 5 of 11.)

```
96  // update balance in record
97  void updateRecord(FILE *fPtr)
98  {
99     // obtain number of account to update
100    printf("%s", "Enter account to update (1 - 100): ");
101    unsigned int account; // account number
102    scanf("%d", &account);
103
104    // move file pointer to correct record in file
105    fseek(fPtr, (account - 1) * sizeof(struct clientData),
106       SEEK_SET);
107
108    // create clientData with no information
109    struct clientData client = {0, "", "", 0.0};
110
111    // read record from file
112    fread(&client, sizeof(struct clientData), 1, fPtr);
113
114    // display error if account does not exist
115    if (client.acctNum == 0) {
116       printf("Account #%d has no information.\n", account);
117    }
```

Fig. 11.15 | Transaction-processing program. (Part 6 of 11.)

```
118     else { // update record
119        printf("%-6d%-16s%-11s%10.2f\n\n",
120           client.acctNum, client.lastName,
121           client.firstName, client.balance);
122
123        // request transaction amount from user
124        printf("%s", "Enter charge (+) or payment (-): ");
125        double transaction; // transaction amount
126        scanf("%lf", &transaction);
127        client.balance += transaction; // update record balance
128
129        printf("%-6d%-16s%-11s%10.2f\n",
130           client.acctNum, client.lastName,
131           client.firstName, client.balance);
132
133        // move file pointer to correct record in file
134        fseek(fPtr, (account - 1) * sizeof(struct clientData),
135           SEEK_SET);
136
137        // write updated record over old record in file
138        fwrite(&client, sizeof(struct clientData), 1, fPtr);
139     }
140  }
141
```

Fig. 11.15 | Transaction-processing program. (Part 7 of 11.)

```
142 // delete an existing record
143 void deleteRecord(FILE *fPtr)
144 {
145    // obtain number of account to delete
146    printf("%s", "Enter account number to delete (1 - 100): ");
147    unsigned int accountNum; // account number
148    scanf("%d", &accountNum);
149
150    // move file pointer to correct record in file
151    fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
152       SEEK_SET);
153
154    struct clientData client; // stores record read from file
155
156    // read record from file
157    fread(&client, sizeof(struct clientData), 1, fPtr);
158
159    // display error if record does not exist
160    if (client.acctNum == 0) {
161       printf("Account %d does not exist.\n", accountNum);
162    }
```

Fig. 11.15 | Transaction-processing program. (Part 8 of 11.)

```
163     else { // delete record
164        // move file pointer to correct record in file
165        fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
166           SEEK_SET);
167
168        struct clientData blankClient = {0, "", "", 0}; // blank client
169
170        // replace existing record with blank record
171        fwrite(&blankClient,
172           sizeof(struct clientData), 1, fPtr);
173     }
174  }
175
176  // create and insert record
177  void newRecord(FILE *fPtr)
178  {
179     // obtain number of account to create
180     printf("%s", "Enter new account number (1 - 100): ");
181     unsigned int accountNum; // account number
182     scanf("%d", &accountNum);
183
184     // move file pointer to correct record in file
185     fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
186        SEEK_SET);
```

Fig. 11.15 | Transaction-processing program. (Part 9 of 11.)

```
187
188     // create clientData with default information
189     struct clientData client = { 0, "", "", 0.0 };
190
191     // read record from file
192     fread(&client, sizeof(struct clientData), 1, fPtr);
193
194     // display error if account already exists
195     if (client.acctNum != 0) {
196        printf("Account #%d already contains information.\n",
197           client.acctNum);
198     }
199     else { // create record
200        // user enters last name, first name and balance
201        printf("%s", "Enter lastname, firstname, balance\n? ");
202        scanf("%14s%9s%lf", &client.lastName, &client.firstName,
203           &client.balance);
204
205        client.acctNum = accountNum;
206
207        // move file pointer to correct record in file
208        fseek(fPtr, (client.acctNum - 1) *
209           sizeof(struct clientData), SEEK_SET);
210
```

Fig. 11.15 | Transaction-processing program. (Part 10 of 11.)

```
211         // insert record in file
212         fwrite(&client,
213            sizeof(struct clientData), 1, fPtr);
214      }
215   }
216
217   // enable user to input menu choice
218   unsigned int enterChoice(void)
219   {
220      // display available options
221      printf("%s", "\nEnter your choice\n"
222         "1 - store a formatted text file of accounts called\n"
223         "    \"accounts.txt\" for printing\n"
224         "2 - update an account\n"
225         "3 - add a new account\n"
226         "4 - delete an account\n"
227         "5 - end program\n? ");
228
229      unsigned int menuChoice; // variable to store user's choice
230      scanf("%u", &menuChoice); // receive choice from user
231      return menuChoice;
232   }
```

Fig. 11.15 | Transaction-processing program. (Part 11 of 11.)

11.10 Secure C Programming

fprintf_s and fscanf_s

- The examples in Sections 11.3–11.4 used functions `fprintf` and `fscanf` to write text to and read text from files, respectively.
- The new standard's Annex K provides more secure versions of these functions named `fprintf_s` and `fscanf_s` that are identical to the `printf_s` and `scanf_s` functions we've previously introduced, except that you also specify a `FILE` pointer argument indicating the file to manipulate.
- If your C compiler's standard libraries include these functions, you should use them instead of `fprintf` and `fscanf`.

11.10 Secure C Programming (Cont.)

Chapter 9 of the CERT Secure C Coding Standard

- Chapter 9 of the CERT Secure C Coding Standard is dedicated to input/output recommendations and rules—many apply to file processing in general and several of these apply to the file-processing functions presented in this chapter.
- For more information on each, visit `www.securecoding.cert.org`.

11.10 Secure C Programming (Cont.)

FIO03-C:

- When opening a file for writing using the non-exclusive file-open modes (Fig. 11.5), if the file exists, function `fopen` opens it and truncates its contents, providing no indication of whether the file existed before the `fopen` call.
- To ensure that an existing file is not opened and truncated, you can use C11's new exclusive mode (discussed in Section 11.3), which allows `fopen` to open the file only if it does not already exist.

11.10 Secure C Programming (Cont.)

FIO04-C:

- In industrial-strength code, you should always check the return values of file-processing functions that return error indicators to ensure that the functions performed their tasks correctly.

FIO07-C.

- Function rewind does not return a value, so you cannot test whether the operation was successful.
- It's recommended instead that you use function `fseek`, because it returns a non-zero value if it fails.

11.10 Secure C Programming (Cont.)

FIO09-C:

- We demonstrated both text files and binary files in this chapter.
- Due to differences in binary data representations across platforms, files written in binary format often are not portable.
- For more portable file representations, consider using text files or a function library that can handle the differences in binary file representations across platforms.

11.10 Secure C Programming (Cont.)

FIO14-C.

- Some library functions do not operate identically on text files and binary files.
- In particular, function `fseek` is not guaranteed to work correctly with binary files if you seek from `SEEK_END`, so `SEEK_SET` should be used.

FIO42-C.

- On many platforms, you can have only a limited number of files open at once. For this reason, you should always close a file as soon as it's no longer needed by your program.