



C File Processing

C How to Program, 6/e



OBJECTIVES

In this chapter, you'll learn:

- To create, read, write and update files.
- Sequential access file processing.
- Random-access file processing.



- 11.1 Introduction
- 11.2 Data Hierarchy
- 11.3 Files and Streams
- 11.4 Creating a Sequential-Access File
- 11.5 Reading Data from a Sequential-Access File
- 11.6 Random-Access Files
- 11.7 Creating a Random-Access File
- 11.8 Writing Data Randomly to a Random-Access File
- 11.9 Reading Data from a Random-Access File
- 11.10 Case Study: Transaction-Processing Program



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, especially disk storage devices.
- ▶ In this chapter, we explain how data files are created, updated and processed by C programs.
- ▶ We consider sequential-access files and random-access files.



11.2 Data Hierarchy

- ▶ Ultimately, all data items processed by a computer are reduced to combinations of **zeros and ones**.
- ▶ This occurs because it's simple and economical to build electronic devices that can assume two stable states—one of the states represents 0 and the other represents 1.
- ▶ It's remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of 0s and 1s.
- ▶ The smallest data item in a computer can assume the value 0 or the value 1.



11.2 Data Hierarchy (Cont.)

- ▶ Such a data item is called a **bit** (short for “**binary digit**”—a digit that can assume one of two values).
- ▶ Computer circuitry performs various simple bit manipulations such as determining a bit’s value, setting a bit’s value and reversing a bit (from 1 to 0 or from 0 to 1).
- ▶ It’s cumbersome to work with data in the low-level form of bits.
- ▶ Instead, programmers prefer to work with data in the form of **decimal digits** (i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9), **letters** (i.e., A–Z, and a–z), and **special symbols** (i.e., \$, @, %, &, *, (,), -, +, ", :, ?, /, and others).



11.2 Data Hierarchy (Cont.)

- ▶ Digits, letters, and special symbols are referred to as **characters**.
- ▶ The set of all characters that may be used to write programs and represent data items on a particular computer is called that computer's **character set**.
- ▶ Since computers can process only 1s and 0s, every character in a computer's character set is represented as a pattern of 1s and 0s (called a **byte**).
- ▶ Today, bytes are most commonly composed of eight bits.
- ▶ You create programs and data items as characters; computers then manipulate and process these characters as patterns of bits.



11.2 Data Hierarchy (Cont.)

- ▶ Just as characters are composed of bits, **fields** are composed of characters.
- ▶ A field is a group of characters that conveys meaning.
- ▶ For example, a field consisting solely of uppercase and lowercase letters can be used to represent a person's name.
- ▶ Data items processed by computers form a **data hierarchy** in which data items become larger and more complex in structure as we progress from bits, to characters (bytes), to fields, and so on.
- ▶ A **record** (i.e., a **struct** in C) is composed of several fields.



11.2 Data Hierarchy (Cont.)

- ▶ In a payroll system, for example, a record for a particular employee might consist of the following fields:
 - Social Security number (alphanumeric field)
 - Name (alphabetic field)
 - Address (alphanumeric field)
 - Hourly salary rate (numeric field)
 - Number of exemptions claimed (numeric field)
 - Year-to-date earnings (numeric field)
 - Amount of federal taxes withheld (numeric field)



11.2 Data Hierarchy (Cont.)

- ▶ Thus, a record is a group of related fields.
- ▶ In the preceding example, each of the fields belongs to the same employee.
- ▶ Of course, a particular company may have many employees and will have a payroll record for each employee.
- ▶ A **file** is a group of related records.



11.2 Data Hierarchy (Cont.)

- ▶ A company's payroll file normally contains one record for each employee.
- ▶ Thus, a payroll file for a small company might contain only 22 records, whereas a payroll file for a large company might contain 100,000 records.
- ▶ It's not unusual for an organization to have hundreds or even thousands of files, with some containing billions or even trillions of characters of information.
- ▶ Figure 11.1 illustrates the data hierarchy.

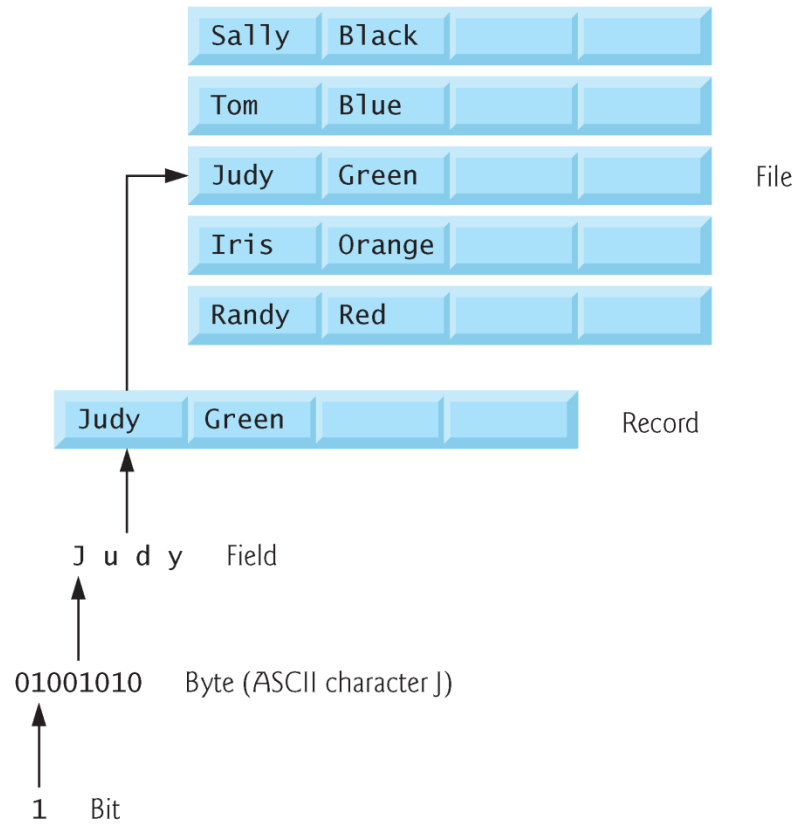


Fig. 11.1 | Data hierarchy.



11.2 Data Hierarchy (Cont.)

- ▶ To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a **record key**.
- ▶ A record key identifies a record as belonging to a particular person or entity.
- ▶ For example, in the payroll record described in this section, the Social Security number would normally be chosen as the record key.
- ▶ There are many ways of organizing records in a file.
- ▶ The most popular type of organization is called a **sequential file**, in which records are typically stored in order by the record key field.



11.2 Data Hierarchy (Cont.)

- ▶ In a payroll file, records are usually placed in order by Social Security Number.
- ▶ The first employee record in the file contains the lowest Social Security number, and subsequent records contain increasingly higher Social Security numbers.
- ▶ Companies may have payroll files, accounts receivable files (listing money due from clients), accounts payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types of files.
- ▶ A group of related files is sometimes called a **database**.
- ▶ A collection of programs designed to create and manage databases is called a **database management system (DBMS)**.



11.3 Files and Streams

- ▶ C views each file simply as a sequential stream of bytes (Fig. 11.2).
- ▶ 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 the file.
- ▶ 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.



11.3 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.
- ▶ 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. 11.2 | C's view of a file of n bytes.



11.3 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.



11.3 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`.



11.4 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.
- ▶ Therefore, you must provide a file structure to meet the requirements of a particular application.
- ▶ The following example shows how to impose a record structure on a file.
- ▶ Figure 11.3 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.



11.4 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.



11.4 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 11.7–11.8 use the data file created in Fig. 11.3, so you must run Fig. 11.3 before Figs. 11.7–11.8.]



```
1  /* Fig. 11.3: fig11_03.c
2     Create a sequential file */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int account; /* account number */
8     char name[ 30 ]; /* account name */
9     double balance; /* account balance */
10
11     FILE *cfPtr; /* cfPtr = clients.dat file pointer */
12
13     /* fopen opens file. Exit program if unable to create file */
14     if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
15         printf( "File could not be opened\n" );
16     } /* end if */
17     else {
18         printf( "Enter the account, name, and balance.\n" );
19         printf( "Enter EOF to end input.\n" );
20         printf( "? " );
21         scanf( "%d%s%lf", &account, name, &balance );
22     }
```

Fig. 11.3 | Creating a sequential file. (Part I of 2.)



```
23      /* write account, name and balance into file with fprintf */
24      while ( !feof( stdin ) ) {
25          fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
26          printf( "? " );
27          scanf( "%d%s%lf", &account, name, &balance );
28      } /* end while */
29
30      fclose( cfPtr ); /* fclose closes file */
31  } /* end else */
32
33      return 0; /* indicates successful termination */
34  } /* end main */
```

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. 11.3 | Creating a sequential file. (Part 2 of 2.)



11.4 Creating a Sequential-Access File (Cont.)

- ▶ Now let's examine this program.
- ▶ Line 11 states that `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, though the interested reader can study the declaration in `stdio.h`.
- ▶ 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 is used to refer to the file.



11.4 Creating a Sequential-Access File (Cont.)

- ▶ Line 14 names the file—"clients.dat"—to be 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 file name 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.



11.4 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.



11.4 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 11.4 lists the key combinations for entering end-of-file for various computer systems.
- ▶ Line 24 uses 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 is no more data to be processed.
- ▶ In Fig. 11.3, 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	<i><Ctrl> d</i>
Windows	<i><Ctrl> z</i>

Fig. 11.4 | End-of-file key combinations for various popular operating systems.



11.4 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.
- ▶ Line 25 writes data to the file `clients.dat`.
- ▶ The data may be retrieved later by a program designed to read the file (see Section 11.5).



11.4 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);`



Common Programming Error 11.3

Using the wrong file pointer to refer to a file is a logic error.



Error-Prevention Tip 11.1

Be sure that calls to file processing functions in a program contain the correct file pointers.



11.4 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 file name) 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.”



Good Programming Practice 11.1

Explicitly close each file as soon as it's no longer needed.



Performance Tip 11.1

Closing a file can free resources for which other users or programs may be waiting.



11.4 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 11.5 illustrates the relationship between **FILE** pointers, **FILE** structures and FCBs in memory.
- ▶ When the file "`clients.dat`" is opened, an FCB for the file is copied into memory.

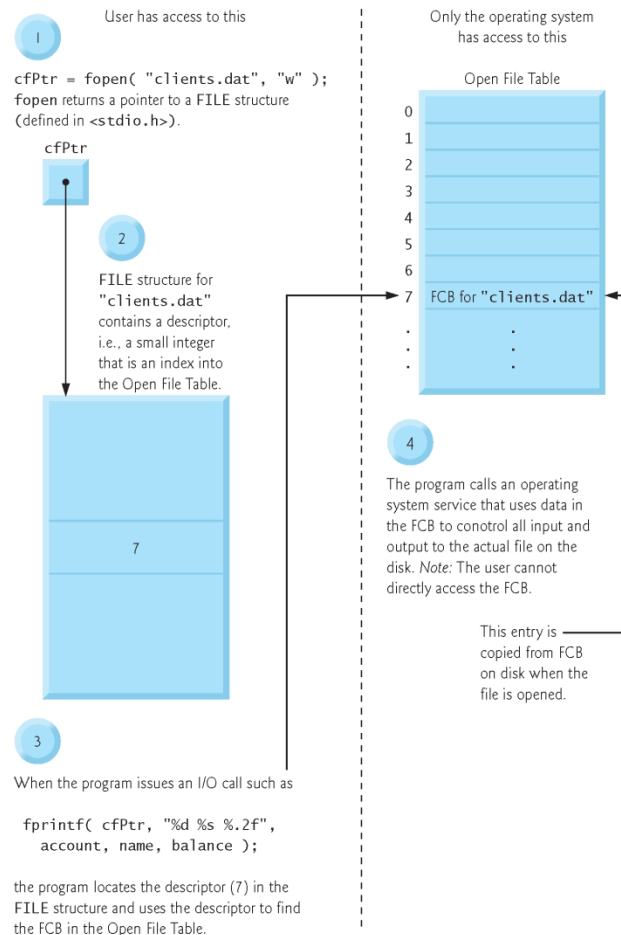


Fig. 11.5 | Relationship between FILE pointers, FILE structures and FCBs.



11.4 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 must have a unique name and 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. 11.6).
- ▶ To create a file, or to discard the contents of a file before writing data, open the file for writing ("w").



11.4 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 a file for reading and writing.
- ▶ Mode "w+" creates a file for reading and writing.
- ▶ If the file already exists, the file is opened and the current contents of the file are discarded.



11.4 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 11.6–11.10 when we introduce random-access files.
- ▶ 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	Append; open or create a file for writing at the end of the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append: open or create a file for update; writing is done at the end of the file.
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. 11.6 | File opening modes.



Common Programming Error 11.4

Opening a nonexistent file for reading is an error.



Common Programming Error 11.5

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 11.6

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



Common Programming Error 11.7

Opening a file with the incorrect file mode is a logic error. For example, 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 11.2

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



11.5 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 11.7 reads records from the file "clients.dat" created by the program of Fig. 11.3 and prints the contents of the records.
- ▶ Line 11 indicates that `cfPtr` is a pointer to a `FILE`.
- ▶ Line 14 attempts to open the file "clients.dat" for reading ("r") and determines whether the file is opened successfully (i.e., `fopen` does not return `NULL`).



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

- ▶ Line 19 reads 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 (line 24) 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 (line 27) and the program terminates.
- ▶ Function `feof` returns true only *after* the program attempts to read the nonexistent data following the last line.



```
1  /* Fig. 11.7: fig11_07.c
2     Reading and printing a sequential file */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int account;      /* account number */
8     char name[ 30 ]; /* account name */
9     double balance;   /* account balance */
10
11     FILE *cfPtr;      /* cfPtr = clients.dat file pointer */
12
13     /* fopen opens file; exits program if file cannot be opened */
14     if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
15         printf( "File could not be opened\n" );
16     } /* end if */
17     else { /* read account, name and balance from file */
18         printf( "%-10s%-13s%\n", "Account", "Name", "Balance" );
19         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
20     }
```

Fig. 11.7 | 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%s%lf", &account, name, &balance );
25      } /* end while */
26
27      fclose( cfPtr ); /* fclose closes the file */
28  } /* end else */
29
30      return 0; /* indicates successful termination */
31  } /* end main */
```

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

Fig. 11.7 | Reading and printing a sequential file. (Part 2 of 2.)



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

- ▶ 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.



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

- ▶ A statement such as
 - `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 location 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.



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

- ▶ The program of Fig. 11.8 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.



11.5 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. 11.9.



```
1  /* Fig. 11.8: fig11_08.c
2     Credit inquiry program */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int request;      /* request number */
9     int account;      /* account number */
10    double balance;   /* account balance */
11    char name[ 30 ];  /* account name */
12    FILE *cfPtr;      /* clients.dat file pointer */
13
14    /* fopen opens the file; exits program if file cannot be opened */
15    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
16        printf( "File could not be opened\n" );
17    } /* end if */
```

Fig. 11.8 | Credit inquiry program. (Part 1 of 6.)



```
18     else {
19
20         /* display request options */
21         printf( "Enter request\n"
22             " 1 - List accounts with zero balances\n"
23             " 2 - List accounts with credit balances\n"
24             " 3 - List accounts with debit balances\n"
25             " 4 - End of run\n? " );
26         scanf( "%d", &request );
27
28         /* process user's request */
29         while ( request != 4 ) {
30
31             /* read account, name and balance from file */
32             fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
33         }
```

Fig. 11.8 | Credit inquiry program. (Part 2 of 6.)



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

Fig. 11.8 | Credit inquiry program. (Part 3 of 6.)



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

Fig. 11.8 | Credit inquiry program. (Part 4 of 6.)



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

Fig. 11.8 | Credit inquiry program. (Part 5 of 6.)



```
93
94     printf( "End of run.\n" );
95     fclose( cfPtr ); /* fclose closes the file */
96 } /* end else */
97
98     return 0; /* indicates successful termination */
99 } /* end main */
```

Fig. 11.8 | Credit inquiry program. (Part 6 of 6.)



11.5 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” needed to be changed to “worthington,” the old name cannot simply be overwritten.
- ▶ The record for white was written to the file as



```
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. 11.9 | Sample output of the credit inquiry program of Fig. 11.8.



11.5 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 would 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” would 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.



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

- ▶ For example, the values 7, 14, -117, 2074 and 27383 are all `ints` stored in the same number of bytes internally, but they are 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.



11.5 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.



11.6 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.



11.6 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 11.10 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.

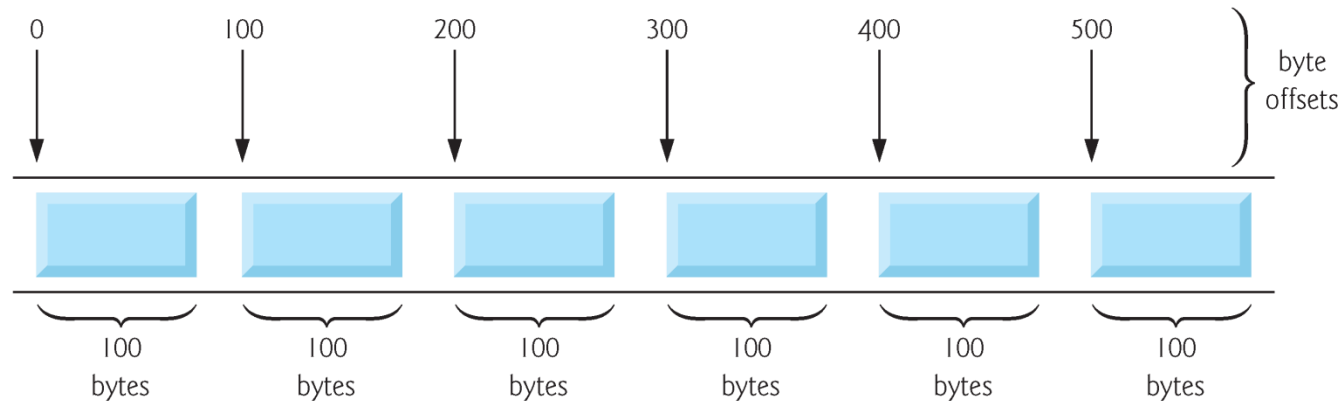


Fig. 11.10 | C's view of a random-access file.



11.6 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.



11.7 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.



11.7 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 4-byte integer, we can use

- `fwrite( &number, sizeof( int ), 1, fPtr );`

which always writes 4 bytes (or 2 bytes on a system with 2-byte integers) from a variable `number` to the file represented by `fPtr` (we'll explain the `1` argument shortly).



11.7 Creating a Random-Access File (Cont.)

- ▶ Later, `fread` can be used to read 4 of those 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.
- ▶ Since 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.



11.7 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 disk 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.



11.7 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 11.11 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 **structs** 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 if a record contains data.



```
1  /* Fig. 11.11: fig11_11.c
2     Creating a random-access file sequentially */
3  #include <stdio.h>
4
5  /* clientData structure definition */
6  struct clientData {
7     int acctNum;          /* account number */
8     char lastName[ 15 ]; /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance;       /* account balance */
11 }; /* end structure clientData */
12
13 int main( void )
14 {
15     int i; /* counter used to count from 1-100 */
16
17     /* create clientData with default information */
18     struct clientData blankClient = { 0, "", "", 0.0 };
19
20     FILE *cfPtr; /* credit.dat file pointer */
21
```

Fig. 11.11 | Creating a random access file sequentially. (Part 1 of 2.)



```
22  /* fopen opens the file; exits if file cannot be opened */
23  if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
24      printf( "File could not be opened.\n" );
25  } /* end if */
26  else {
27      /* output 100 blank records to file */
28      for ( i = 1; i <= 100; i++ ) {
29          fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
30      } /* end for */
31
32      fclose ( cfPtr ); /* fclose closes the file */
33  } /* end else */
34
35  return 0; /* indicates successful termination */
36  } /* end main */
```

Fig. 11.11 | Creating a random access file sequentially. (Part 2 of 2.)



11.7 Creating a Random-Access File (Cont.)

- ▶ Function `fwrite` writes a block (specific number of bytes) of data to a file.
- ▶ In our program, line 29 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`).
- ▶ The `sizeof` operator returns an unsigned integer and can be used to determine the size in bytes of any data type or expression.



11.7 Creating a Random-Access File (Cont.)

- ▶ For example, `sizeof(int)` can be used to determine whether an integer is stored in 2 or 4 bytes on a particular computer.
- ▶ Function `fwrite` can actually be used to write several elements of an array of objects.
- ▶ To write several array elements, 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.



11.7 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 11.12, 11.15 and 11.16 use the data file created in Fig. 11.11, so you must run Fig. 11.11 before Figs. 11.12, 11.15 and 11.16]



11.8 Writing Data Randomly to a Random-Access File

- ▶ Figure 11.12 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. 11.13.



```
1  /* Fig. 11.12: fig11_12.c
2     Writing to a random access file */
3  #include <stdio.h>
4
5  /* clientData structure definition */
6  struct clientData {
7     int acctNum;          /* account number */
8     char lastName[ 15 ]; /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance;        /* account balance */
11 }; /* end structure clientData */
12
13 int main( void )
14 {
15     FILE *cfPtr; /* credit.dat file pointer */
16
17     /* create clientData with default information */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen opens the file; exits if file cannot be opened */
21     if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
22         printf( "File could not be opened.\n" );
23     } /* end if */
```

Fig. 11.12 | Writing data randomly to a random-access file. (Part 1 of 3.)



```
24  else {
25      /* require user to specify account number */
26      printf( "Enter account number"
27             " ( 1 to 100, 0 to end input )\n? " );
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( "Enter lastname, firstname, balance\n? " );
34
35          /* set record lastName, firstName and balance value */
36          fscanf( stdin, "%s%s%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. 11.12 | Writing data randomly to a random-access file. (Part 2 of 3.)



```
46         /* enable user to input another account number */
47         printf( "Enter account number\n? " );
48         scanf( "%d", &client.acctNum );
49     } /* end while */
50
51     fclose( cfPtr ); /* fclose closes the file */
52 } /* end else */
53
54 return 0; /* indicates successful termination */
55 } /* end main */
```

Fig. 11.12 | Writing data randomly to a random-access file. (Part 3 of 3.)



11.8 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 $(\text{client.accountNum} - 1) * \text{sizeof}(\text{struct clientData})$.
- ▶ 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.



```
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. 11.13 | Sample execution of the program in Fig. 11.12.



11.8 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 11.14 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.

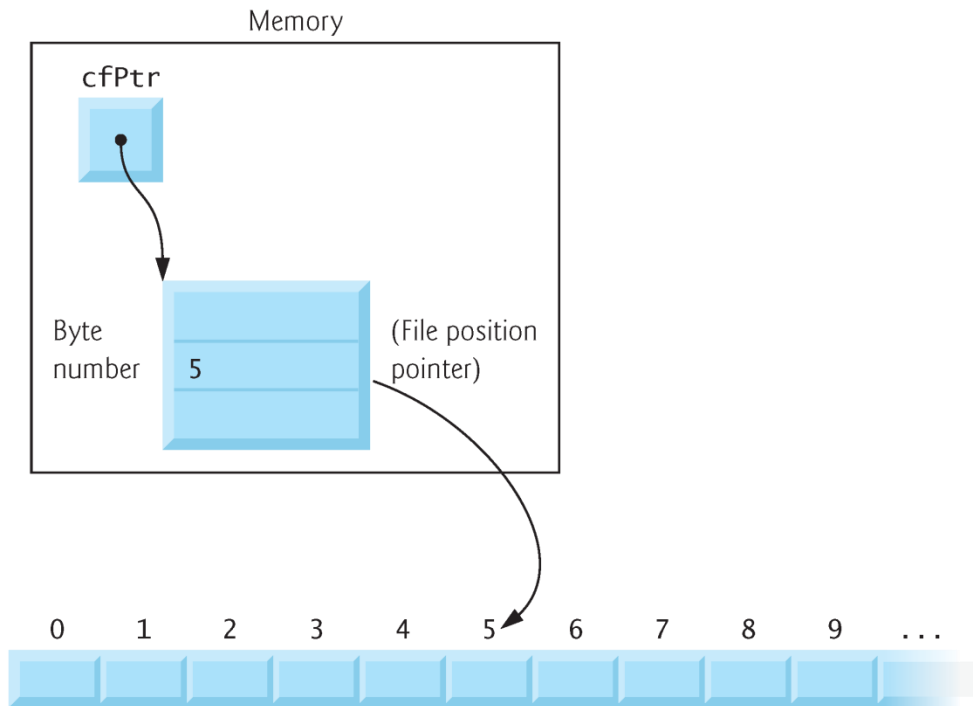


Fig. 11.14 | File position pointer indicating an offset of 5 bytes from the beginning of the file.



11.8 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 location `whence` in the file pointed to by `stream`.
- ▶ The argument `whence` can have one of three values—`SEEK_SET`, `SEEK_CUR` or `SEEK_END` (all defined in `<stdio.h>`)—indicating the location in the file from which the seek begins.



11.8 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.
- ▶ If you wish to determine whether functions like `fscanf` (lines 36–37), `fseek` (lines 40–41) and `fwrite` (line 44) operate correctly, you can check their return values.
- ▶ Function `fscanf` returns the number of data items successfully read or the value `EOF` if a problem occurs while reading data.



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

- ▶ 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.



11.9 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` and stores the data in the structure `client`.
- ▶ The bytes are read from the location in the file specified by the file position pointer.



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

- ▶ Function `fread` can be used to 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 specifies that one element should be read.
- ▶ To read more than one element, specify the number of elements in the third argument of the `fread` statement.
- ▶ Function `fread` returns the number of items it successfully input.



11.9 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 11.15 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 disk to the `clientData` structure `client`.



```
1  /* Fig. 11.15: fig11_15.c
2     Reading a random access file sequentially */
3  #include <stdio.h>
4
5  /* clientData structure definition */
6  struct clientData {
7     int acctNum; /* account number */
8     char lastName[ 15 ]; /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance; /* account balance */
11 }; /* end structure clientData */
12
13 int main( void )
14 {
15     FILE *cfPtr; /* credit.dat file pointer */
16
17     /* create clientData with default information */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen opens the file; exits if file cannot be opened */
21     if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
22         printf( "File could not be opened.\n" );
23     } /* end if */
```

Fig. 11.15 | Reading a random-access file sequentially. (Part 1 of 3.)



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

Fig. 11.15 | 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. 11.15 | Reading a random-access file sequentially. (Part 3 of 3.)



11.10 Case Study: Transaction-Processing Program

- ▶ We now present a substantial transaction-processing program using random-access files.
- ▶ The program maintains a bank's account information.
- ▶ The program updates existing accounts, adds new accounts, deletes accounts and stores a listing of all the current accounts in a text file for printing.
- ▶ We assume that the program of Fig. 11.11 has been executed to create the file `credit.dat`.



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

- ▶ The program has five options.
- ▶ Option 1 calls function `textFile` (lines 65–95) to store a formatted list of all the accounts 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. 11.15.



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

- ▶ Option 2 calls the function `updateRecord` (lines 98–142) to update an account.
- ▶ The function will only update a record that already exists, so the function first checks to see if 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.



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

- ▶ Option 3 calls the function **newRecord** (lines 179–218) to add a new account to the file.
- ▶ If the user enters an account number for an existing account, **newRecord** displays an error message 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. 11.12.



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

- ▶ Option 4 calls function `deleteRecord` (lines 145–176) 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 that the account does not exist.
- ▶ Option 5 terminates program execution.
- ▶ The program is shown in Fig. 11.16.
- ▶ The file "`credit.dat`" is opened for update (reading and writing) using "`rb+`" mode.



```
1  /* Fig. 11.16: fig11_16.c
2     This program reads a random access file sequentially, updates data
3     already written to the file, creates new data to be placed in the
4     file, and deletes data previously in the file. */
5  #include <stdio.h>
6
7  /* clientData structure definition */
8  struct clientData {
9     int acctNum; /* account number */
10    char lastName[ 15 ]; /* account last name */
11    char firstName[ 10 ]; /* account first name */
12    double balance; /* account balance */
13 }; /* end structure clientData */
14
15 /* prototypes */
16 int enterChoice( void );
17 void textFile( FILE *readPtr );
18 void updateRecord( FILE *fPtr );
19 void newRecord( FILE *fPtr );
20 void deleteRecord( FILE *fPtr );
21
```

Fig. 11.16 | Bank account program. (Part I of 12.)



```
22  int main( void )
23  {
24      FILE *cfPtr; /* credit.dat file pointer */
25      int choice; /* user's choice */
26
27      /* fopen opens the file; exits if file cannot be opened */
28      if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
29          printf( "File could not be opened.\n" );
30      } /* end if */
31      else {
32          /* enable user to specify action */
33          while ( ( choice = enterChoice() ) != 5 ) {
34              switch ( choice ) {
35                  /* create text file from record file */
36                  case 1:
37                      textFile( cfPtr );
38                      break;
39                  /* update record */
40                  case 2:
41                      updateRecord( cfPtr );
42                      break;
```

Fig. 11.16 | Bank account program. (Part 2 of 12.)



```
43         /* create record */
44         case 3:
45             newRecord( cfPtr );
46             break;
47         /* delete existing record */
48         case 4:
49             deleteRecord( cfPtr );
50             break;
51         /* display message if user does not select valid choice */
52         default:
53             printf( "Incorrect choice\n" );
54             break;
55     } /* end switch */
56 } /* end while */
57
58     fclose( cfPtr ); /* fclose closes the file */
59 } /* end else */
60
61     return 0; /* indicates successful termination */
62 } /* end main */
63
```

Fig. 11.16 | Bank account program. (Part 3 of 12.)



```
64  /* create formatted text file for printing */
65  void textFile( FILE *readPtr )
66  {
67      FILE *writePtr; /* accounts.txt file pointer */
68
69      /* create clientData with default information */
70      struct clientData client = { 0, "", "", 0.0 };
71
72      /* fopen opens the file; exits if file cannot be opened */
73      if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL ) {
74          printf( "File could not be opened.\n" );
75      } /* end if */
76      else {
77          rewind( readPtr ); /* sets pointer to beginning of file */
78          fprintf( writePtr, "%-6s%-16s%-11s%10s\n",
79                  "Acct", "Last Name", "First Name", "Balance" );
80
```

Fig. 11.16 | Bank account program. (Part 4 of 12.)



```
81      /* copy all records from random-access file into text file */
82      while ( !feof( readPtr ) ) {
83          fread( &client, sizeof( struct clientData ), 1, readPtr );
84
85          /* write single record to text file */
86          if ( client.acctNum != 0 ) {
87              fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n",
88                  client.acctNum, client.lastName,
89                  client.firstName, client.balance );
90          } /* end if */
91      } /* end while */
92
93      fclose( writePtr ); /* fclose closes the file */
94  } /* end else */
95 } /* end function textFile */
96
```

Fig. 11.16 | Bank account program. (Part 5 of 12.)



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

Fig. 11.16 | Bank account program. (Part 6 of 12.)



```
121 else { /* update record */
122     printf( "%-6d%-16s%-11s%10.2f\n\n",
123         client.acctNum, client.lastName,
124         client.firstName, client.balance );
125
126     /* request transaction amount from user */
127     printf( "Enter charge ( + ) or payment ( - ): " );
128     scanf( "%lf", &transaction );
129     client.balance += transaction; /* update record balance */
130
131     printf( "%-6d%-16s%-11s%10.2f\n",
132         client.acctNum, client.lastName,
133         client.firstName, client.balance );
134
135     /* move file pointer to correct record in file */
136     fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
137         SEEK_SET );
138
139     /* write updated record over old record in file */
140     fwrite( &client, sizeof( struct clientData ), 1, fPtr );
141 } /* end else */
142 } /* end function updateRecord */
143
```

Fig. 11.16 | Bank account program. (Part 7 of 12.)



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

Fig. 11.16 | Bank account program. (Part 8 of 12.)



```
167     else { /* delete record */
168         /* move file pointer to correct record in file */
169         fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),
170             SEEK_SET );
171
172         /* replace existing record with blank record */
173         fwrite( &blankClient,
174             sizeof( struct clientData ), 1, fPtr );
175     } /* end else */
176 } /* end function deleteRecord */
177
```

Fig. 11.16 | Bank account program. (Part 9 of 12.)



```
178 /* create and insert record */
179 void newRecord( FILE *fPtr )
180 {
181     /* create clientData with default information */
182     struct clientData client = { 0, "", "", 0.0 };
183
184     int accountNum; /* account number */
185
186     /* obtain number of account to create */
187     printf( "Enter new account number ( 1 - 100 ): " );
188     scanf( "%d", &accountNum );
189
190     /* move file pointer to correct record in file */
191     fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),
192           SEEK_SET );
193
194     /* read record from file */
195     fread( &client, sizeof( struct clientData ), 1, fPtr );
196
197     /* display error if account already exists */
198     if ( client.acctNum != 0 ) {
199         printf( "Account #%d already contains information.\n",
200               client.acctNum );
201     } /* end if */
```

Fig. 11.16 | Bank account program. (Part 10 of 12.)



```
202     else { /* create record */
203         /* user enters last name, first name and balance */
204         printf( "Enter lastname, firstname, balance\n? " );
205         scanf( "%s%s%lf", &client.lastName, &client.firstName,
206             &client.balance );
207
208         client.acctNum = accountNum;
209
210         /* move file pointer to correct record in file */
211         fseek( fPtr, ( client.acctNum - 1 ) *
212             sizeof( struct clientData ), SEEK_SET );
213
214         /* insert record in file */
215         fwrite( &client,
216             sizeof( struct clientData ), 1, fPtr );
217     } /* end else */
218 } /* end function newRecord */
219
```

Fig. 11.16 | Bank account program. (Part 11 of 12.)



```
220  /* enable user to input menu choice */
221  int enterChoice( void )
222  {
223      int menuChoice; /* variable to store user's choice */
224
225      /* display available options */
226      printf( "\nEnter your choice\n"
227             "1 - store a formatted text file of accounts called\n"
228             "    \"accounts.txt\" for printing\n"
229             "2 - update an account\n"
230             "3 - add a new account\n"
231             "4 - delete an account\n"
232             "5 - end program\n? " );
233
234      scanf( "%d", &menuChoice ); /* receive choice from user */
235      return menuChoice;
236  } /* end function enterChoice */
```

Fig. 11.16 | Bank account program. (Part 12 of 12.)