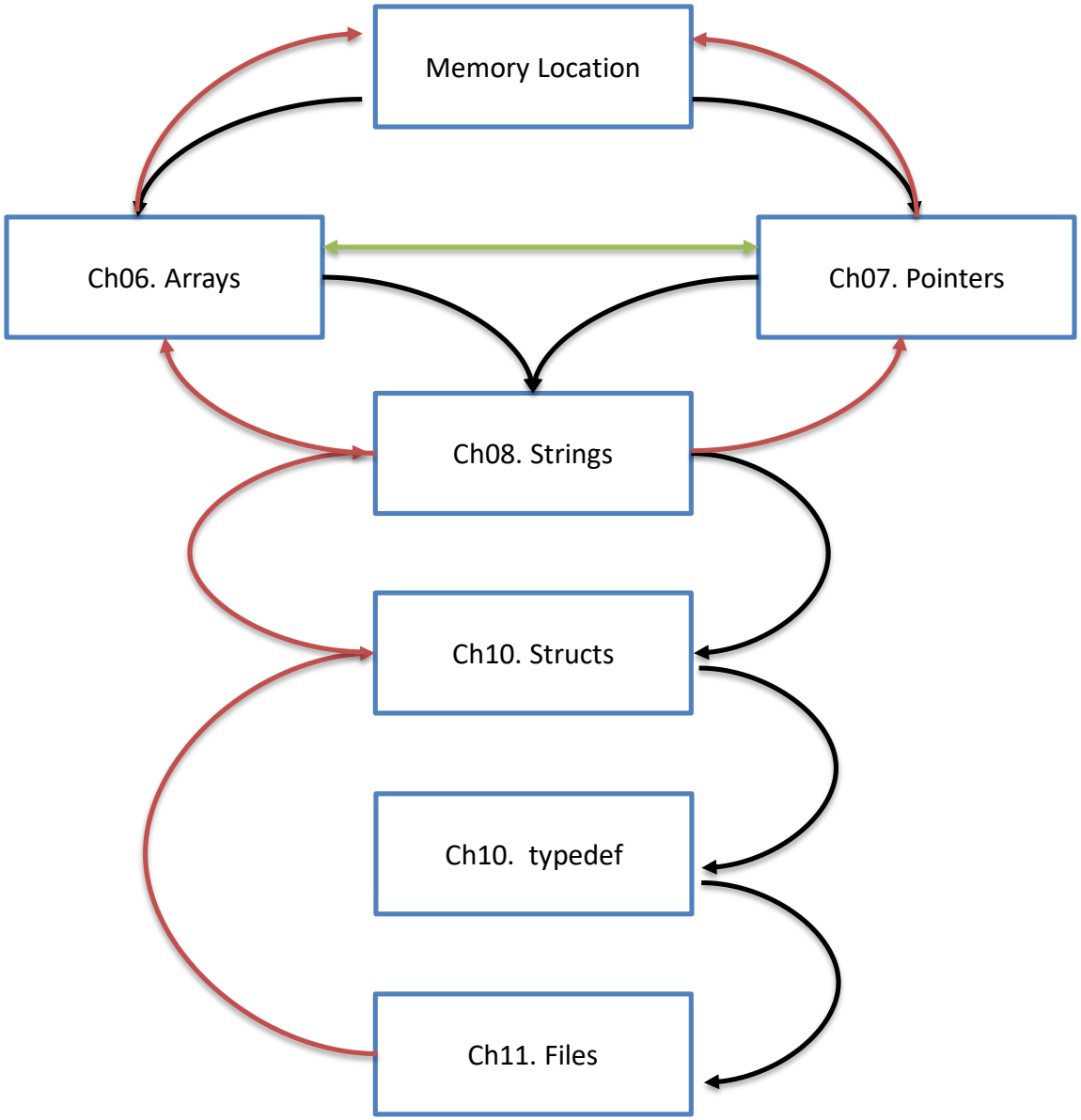
02_11_ C File Processing


Memory Location
Ch06. Arrays
Ch07. Pointers
Ch08. Strings
Ch10. Structs
Ch10. typedef
Ch11. Files

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.

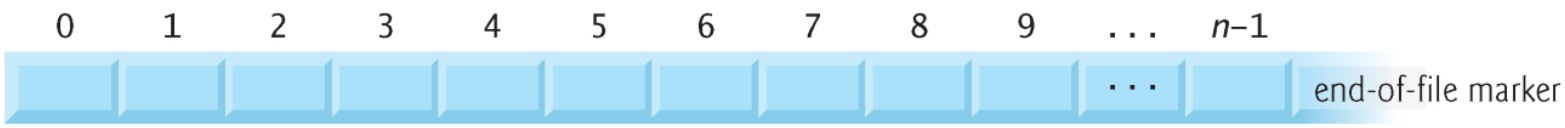


Fig. 11.1 | C's view of a file of *n* bytes.

```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.)

FILE Structure

```
typedef struct
{
 short level ;
 short token ;
 short bsize ;
 char fd ;
 unsigned flags ;
 unsigned char hold ;
 unsigned char *buffer ;
 unsigned char * curp ;
 unsigned istemp;
}FILE ;
```

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.

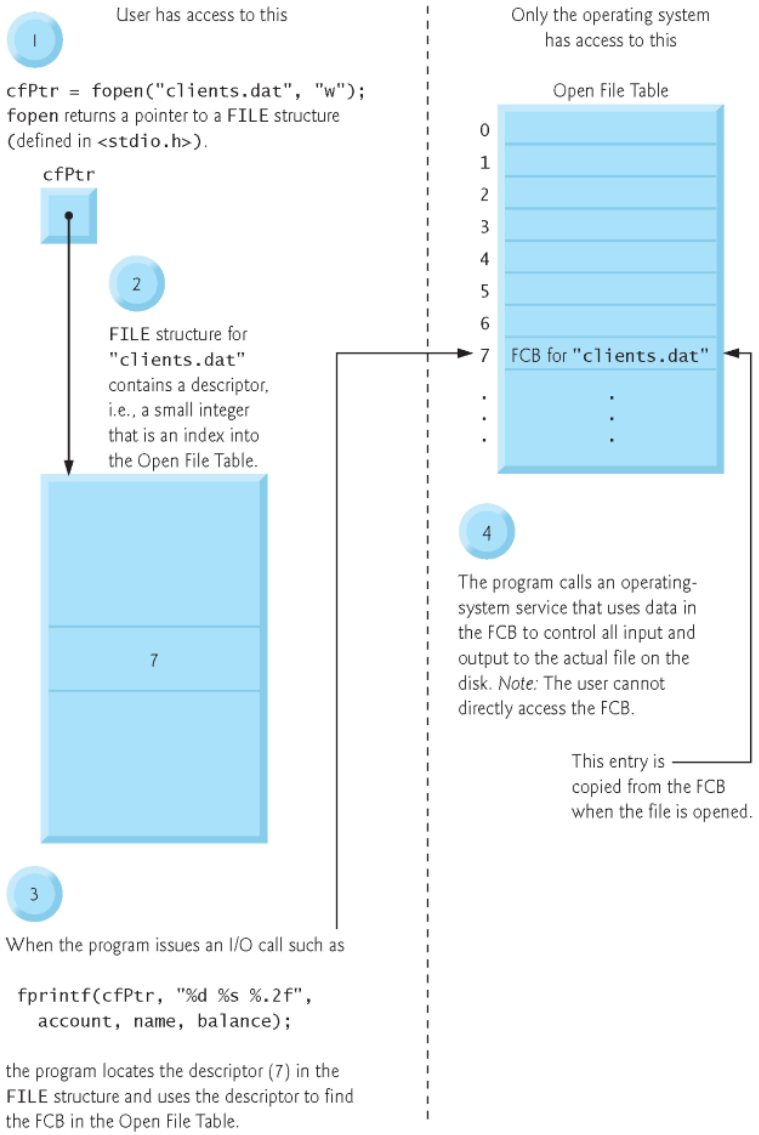


Fig. 11.4 | Relationship between `FILE` pointers, `FILE` structures and FCBs.

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.)

```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.)

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.

11.5 Random-Access Files

- 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.

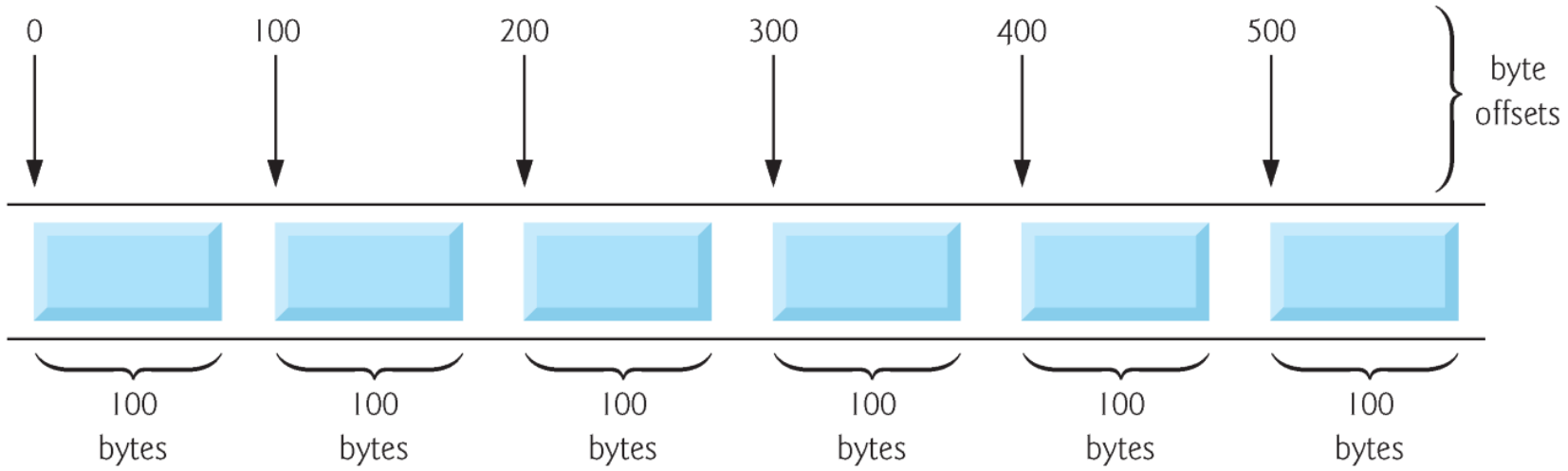


Fig. 11.9 | C's view of a random-access file.

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 (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.

```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.)

11.7 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.

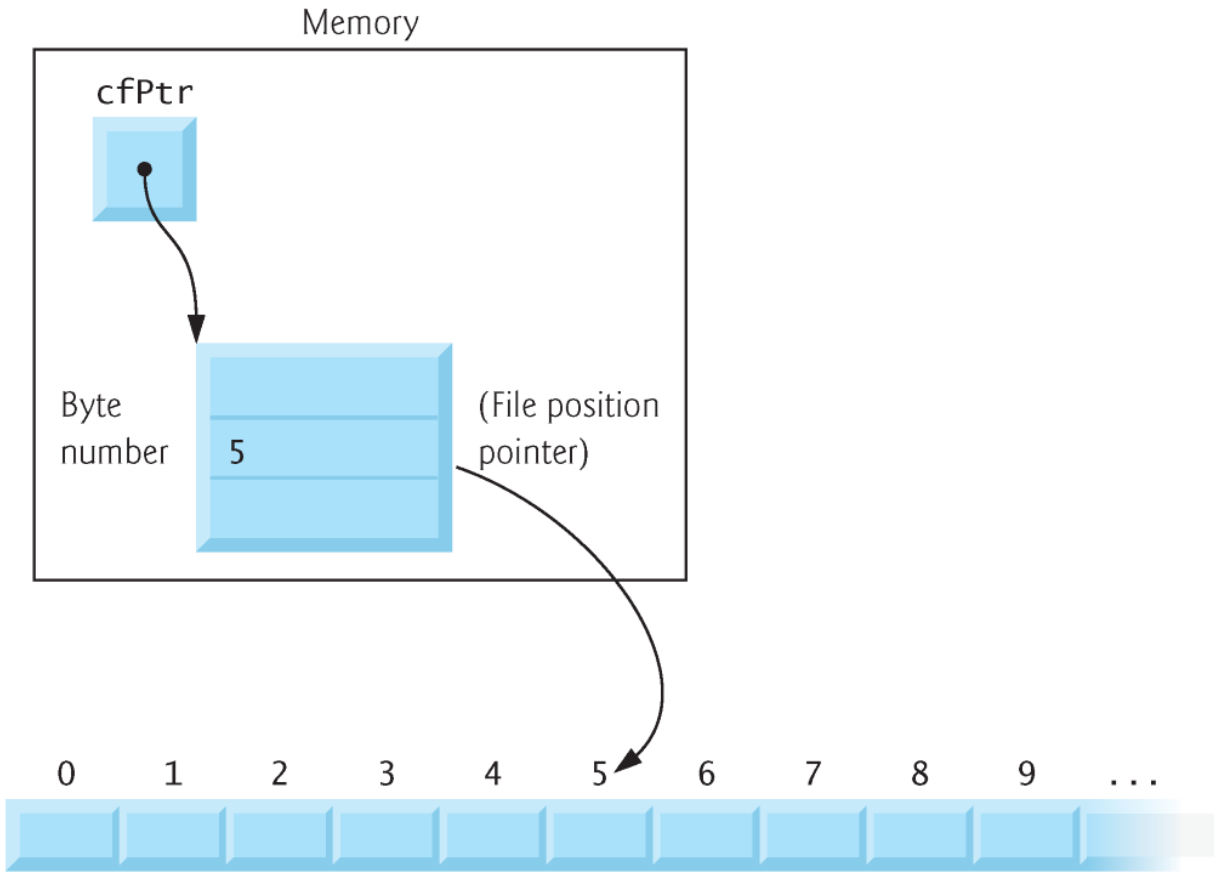


Fig. 11.13 | File position pointer indicating an offset of 5 bytes from the beginning of the file.

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.

11.8 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.)

11.10 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.