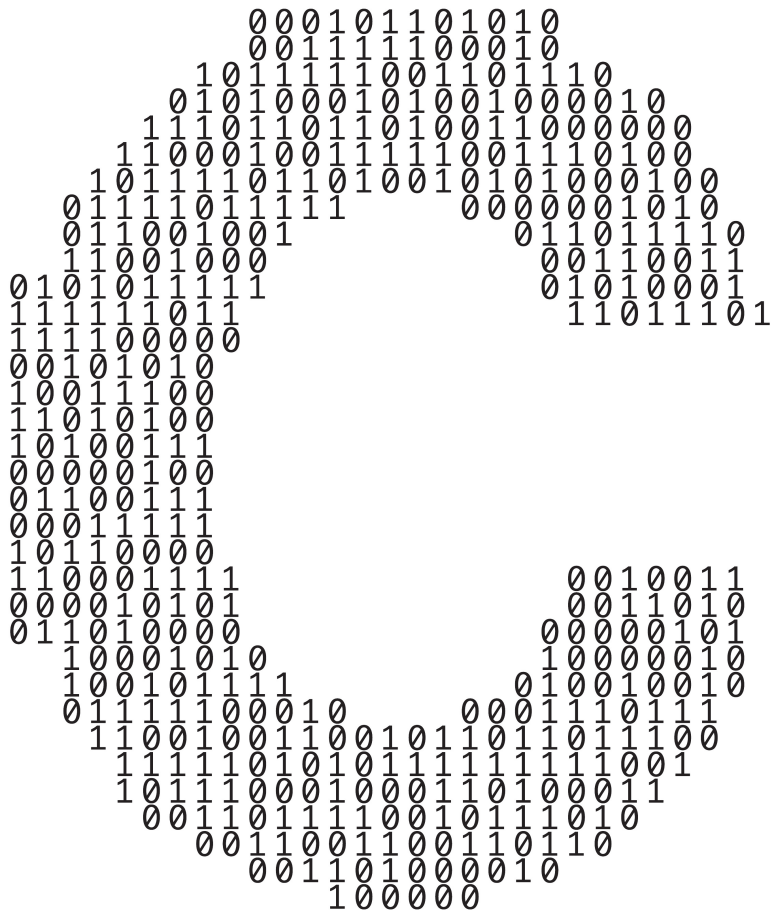
11

File Processing



Objectives

In this chapter, you'll:

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

Outline

11.1 Introduction
11.2 Files and Streams
11.3 Creating a Sequential-Access File
- 11.3.1 Pointer to a `FILE`
- 11.3.2 Using `fopen` to Open a File
- 11.3.3 Using `feof` to Check for the End-of-File Indicator
- 11.3.4 Using `fprintf` to Write to a File
- 11.3.5 Using `fclose` to Close a File
- 11.3.6 File-Open Modes

11.4 Reading Data from a Sequential-Access File
- 11.4.1 Resetting the File Position Pointer
- 11.4.2 Credit Inquiry Program

11.5 Random-Access Files
11.6 Creating a Random-Access File
11.7 Writing Data Randomly to a Random-Access File
- 11.7.1 Positioning the File Position Pointer with `fseek`
- 11.7.2 Error Checking

11.8 Reading Data from a Random-Access File
11.9 Case Study: Transaction-Processing System
11.10 Secure C Programming

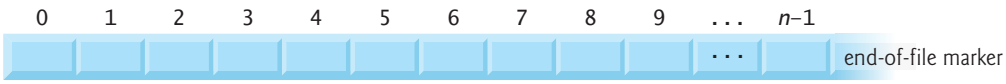
Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises* | *AI Case Study: Intro to NLP—Who Wrote Shakespeare's Works?* | *AI/Data-Science Case Study—Machine Learning with GNU Scientific Library* | *AI/Data-Science Case Study: Time Series and Simple Linear Regression* | *Web Services and the Cloud Case Study—libcurl and OpenWeatherMap*

11.1 Introduction

You studied the *data hierarchy* in Chapter 1. Data in variables is *temporary*—it's *lost* when a program terminates. **Files** enable long-term data retention. Computers store files on secondary storage devices, such as solid-state drives, flash drives and hard drives. This chapter explains how to create, update and process data files. We consider both sequential-access and random-access file processing.

11.2 Files and Streams

C views each file as a sequential stream of bytes, as shown in the following diagram:



Each file ends with an **end-of-file marker** or at a specific byte number recorded in a system-maintained, administrative data structure. This is platform-dependent and hidden from you.

Standard Streams in Every Program

When you open a file, C associates a **stream** with it. When program execution begins, C opens three streams automatically:

- The **standard input** stream receives input from the keyboard.
- The **standard output** stream displays output on the screen.
- The **standard error** stream displays error messages on the screen.

FILE Structure

Opening a file returns a pointer to a **FILE structure** (defined in `<stdio.h>`) containing information the program needs to process the file. In some operating systems, this structure includes a **file descriptor**—an integer index into an operating-system array called the **open file table**. Each array element contains a **file control block (FCB)**—information that the operating system uses to administer a particular file. You manipulate the standard input, standard output and standard error streams using the `FILE` pointers **`stdin`**, **`stdout`** and **`stderr`**.

File-Processing Function fgetc

The standard library provides many functions for reading data from and writing data to files. Function **`fgetc`**, like `getchar`, reads one character from the file specified by its `FILE` pointer argument. For example, the call `fgetc(stdin)` reads one character from the standard input stream. This call is equivalent to the call `getchar()`.

File-Processing Function fputc

Function **`fputc`**, like `putchar`, writes the character in its first argument to the file specified by the `FILE` pointer in its second argument. For example, the function call `fputc('a', stdout)` writes a character to the standard output stream and is equivalent to `putchar('a')`.

Other File-Processing Functions

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, read a line of text from a file and write a line of text to a file, respectively. The next few sections introduce the file-processing equivalents of functions `scanf` and `printf`—**`fscanf`** and **`fprintf`**. Later in the chapter, we discuss functions **`fread`** and **`fwrite`**.

Self Check

1 *(Fill-In)* When program execution begins, C opens three streams automatically: the _____, _____ and _____ streams.
Answer: standard input, standard output, standard error.

2 *(Fill-In)* C views each file as a sequential stream of bytes. Each file ends either with a(n) _____ or at a specific byte number recorded in a system-maintained, administrative data structure.
Answer: end-of-file marker.

11.3 Creating a Sequential-Access File

C imposes no structure on a file. Thus, notions such as a record of a file are not part of the C language. The following example shows how you can impose your own record structure on a file.

Figure 11.1 creates a simple sequential-access file that might be used in an accounts-receivable system to track the amounts owed by a company's credit clients. For each client, the program obtains the client's account number, name and balance—the amount the client owes the company for goods and services received in the past. The data for each client constitutes a "record" for that client. The account number is this application's record key. This program assumes the user enters the records in account-number order. In a comprehensive accounts-receivable system, a sorting capability would enable the user to enter records in any order. The program would then sort the records and write them to the file. Figures 11.2–11.3 use the data file created in Fig. 11.1, so you must run the program in Fig. 11.1 before the programs in Figs. 11.2–11.3.

```
1  // fig11_01.c
2  // Creating a sequential file
3  #include <stdio.h>
4
5  int main(void){
6     FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer
7
8     // fopen opens the file. Exit the program if unable to create the file
9     if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
10       puts("File could not be opened");
11    }
12    else {
13       puts("Enter the account, name, and balance.");
14       puts("Enter EOF to end input.");
15       printf("%s", "? ");
16
17       int account = 0; // account number
18       char name[30] = ""; // account name
19       double balance = 0.0; // account balance
20
21       scanf("%d%29s%lf", &account, name, &balance);
22
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("%s", "? ");
27          scanf("%d%29s%lf", &account, name, &balance);
28       }
29
30       fclose(cfPtr); // fclose closes file
31    }
32 }
```

Fig. 11.1 | Creating a sequential file. (Part 1 of 2.)

```
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. 11.1 | Creating a sequential file. (Part 2 of 2.)

11.3.1 Pointer to a FILE

Line 6 defines `cfPtr` as a pointer to a `FILE` structure. A program refers to each open file with a separate `FILE` pointer. You need not know the `FILE` structure's specifics to use files. If you're interested, you can study its declaration in `stdio.h`.

11.3.2 Using fopen to Open a File

Line 9 calls **fopen** to create the file `"clients.txt"` and establish a "line of communication" with it. The file pointer that `fopen` returns is assigned to `cfPtr`.

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 `fopen` should open the file for writing. If the file does not exist and the file open mode is `"w"`, `fopen` creates the file. If you open an existing file, `fopen` discards the file's contents *without warning*. This is a logic error, ⊗ ERR if your program is not supposed to replace the existing file.

The `if` statement determines whether the file pointer `cfPtr` is `NULL`. If it's `NULL`, the file could not be opened, possibly because the program does not have permission to create a file in the specified folder. In this program, the file gets created in the same folder as the program. If `cfPtr` is `NULL`, the program prints an error message and terminates. Otherwise, the program processes the user's inputs and writes them to the file.

11.3.3 Using feof to Check for the End-of-File Indicator

The program prompts the user to enter the various fields for each record or to enter *end-of-file* when data entry is complete. The key combinations for end-of-file are platform-dependent:

- Windows: <*Ctrl*> + *z*, then press *Enter*
- macOS/Linux: <*Ctrl*> + *d*

Line 24 calls `feof` to determine whether the end-of-file indicator is set for `stdin`. The end-of-file indicator informs the program that there's no more data to process. When the user enters the *end-of-file key combination*, the operating system sets the end-of-file indicator for the standard input stream. The `feof` function's argument is a `FILE` pointer to the file to test for the end-of-file indicator—`stdin` in this case. The function returns a nonzero (*true*) value when the end-of-file indicator has been set; otherwise, the function returns zero (*false*). This program's `while` statement continues executing until the user enters the end-of-file indicator.

11.3.4 Using `fprintf` to Write to a File

Line 25 writes a record as a line of text to the file `clients.txt`. You can retrieve the data later using a program designed to read the file (Section 11.4). The `fprintf` function is equivalent to `printf`, but `fprintf` also receives a `FILE` pointer argument specifying 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 argument.

11.3.5 Using `fclose` to Close a File

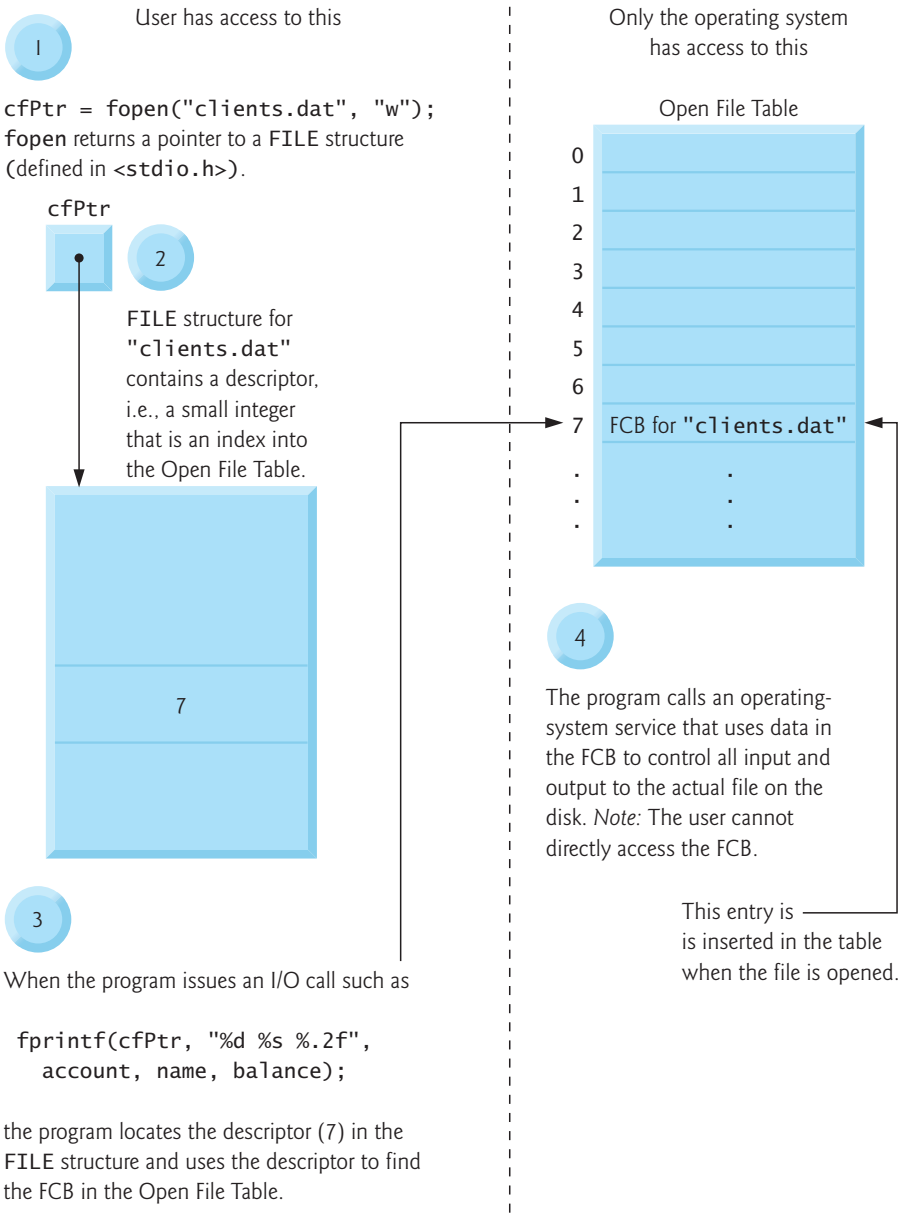
After the user enters end-of-file, the program closes the `clients.txt` file by calling `fclose` (line 30), then terminates. Function `fclose` receives the `FILE` pointer as an argument. If you do not call `fclose` explicitly, the operating system normally will close the file when program execution terminates. This is an example of operating-system "housekeeping." You should close each file as soon as it's no longer needed. This frees resources for which other users or programs may be waiting.

PERF

In Fig. 11.1's sample execution, the user enters information for five accounts, then enters end-of-file to complete data entry. The sample execution does not show how the data records actually appear in the file. The next section presents a program that reads the file and displays its contents to verify that the program created the file successfully.

Relationship Between `FILE` Pointers, `FILE` Structures and FCBs

The following diagram illustrates the relationship between `FILE` pointers, `FILE` structures and FCBs. When a program opens `"clients.txt"`, the operating system copies an FCB for the file into memory. 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 has 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.



11.3.6 File-Open Modes

The following table summarizes the file-open modes. The ones containing the letter "b" are for manipulating binary files, which we discuss in Sections 11.5–11.9 when we introduce random-access files.

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 a file—this is for write operations that append data to a 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 where all writing is done at the end of the file—that is, write operations append data to the file.
`rb`	Open an existing binary file for reading.
`wb`	Create a binary file for writing. If the file already exists, discard the current contents.
`ab`	Open or create a binary file for writing at the end of the file (appending).
`rb+`	Open an existing binary file for update (reading and writing).
`wb+`	Create a binary file for update. If the file already exists, discard the current contents.
`ab+`	Open or create a binary file for update. Writing is done at the end of the file.

C11 Exclusive Write Mode

C11 added the exclusive write mode,[1] indicated with `"wx"`, `"w+x"`, `"wbx"` or `"wb+x"`. In exclusive write mode, `fopen` fails if the file already exists or cannot be created. If your program successfully opens a file in exclusive write mode and the underlying system supports exclusive file access, then *only* your program can access the file while it's open. If an error occurs while opening a file in any mode, `fopen` returns `NULL`.

ERR

Common File-Processing Errors

Some common file-processing logic errors you might encounter include:

- Opening a nonexistent file for reading.
- Opening a file for reading or writing without having been granted the appropriate access rights to the file (this is operating-system dependent).
- Opening a file for writing when no space is available.
- Opening a file in write mode (`"w"`) when it should be opened in update mode (`"r+"`)—`"w"` discards the file's contents.

Self Check

1 *(Code)* Where will the following statement write its output?

```
fprintf(stdout, "%d %s %.2f\n", account, name, balance);
```

Answer: The standard output device, which is normally the screen.

1. Some compilers and platforms do not support exclusive write mode.

2 ***(True/False)*** The notion of a record of a file is built into C.
Answer: *False*. Actually, C imposes no structure on a file, so notions such as a record of a file are not part of the C language. You can impose your own record structure on a file.

3 ***(Fill-In)*** A C program administers each file with a separate _____ structure.
Answer: `FILE`.

4 ***(True/False)*** When you open a file for writing, `fopen` warns you if the file already exists.
Answer: *False*. `fopen` discards the file's contents without warning.

5 ***(Multiple Choice)*** Which file-open mode corresponds to the description, "open an existing file for update (reading and writing)"?

a) u.
b) `rw`.
c) `r+`.
d) `w+`.

Answer: c.

11.4 Reading Data from a Sequential-Access File

Data is stored in files so that it 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. If a file's contents should not be modified, open the file only for reading. This prevents unintentional modification of the file's contents and is another example of the principle of least privilege.

Figure 11.2 reads and displays records from the file `"clients.txt"` created in Fig. 11.1. Line 6 defines the `FILE` pointer `cfPtr`. Line 9 attempts to open the file for reading (`"r"`) and determines whether it opened successfully—that is, `fopen` did not return `NULL`. Line 18 reads a "record" from the file. Function **`fscanf`** is equivalent to `scanf` but receives as its first argument a `FILE` pointer for the file from which to read. The first time this statement executes, `account` will have the value `100`, `name` will have the value `"Jones"` and `balance` will have the value `24.98`. Each subsequent call to `fscanf` (line 23) reads another record from the file and gives new values to `account`, `name` and `balance`. When there is no more data to read, line 26 closes the file, and the program terminates. Function `feof` returns *true* only after the program attempts to read past the file's last line.

```
1  // fig11_02.c
2  // Reading and printing a sequential file
3  #include <stdio.h>
4
5  int main(void) {
6     FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer
```

Fig. 11.2 | Reading and printing a sequential file. (Part 1 of 2.)

```
7
8     // fopen opens file; exits program if file cannot be opened
9     if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
10       puts("File could not be opened");
11    }
12    else { // read account, name and balance from file
13       int account = 0; // account number
14       char name[30] = ""; // account name
15       double balance = 0.0; // account balance
16
17       printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
18       fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
19
20       // while not end of file
21       while (!feof(cfPtr)) {
22          printf("%-10d%-13s%7.2f\n", account, name, balance);
23          fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
24       }
25
26       fclose(cfPtr); // fclose closes the file
27    }
28 }
```

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

Fig. 11.2 | Reading and printing a sequential file. (Part 2 of 2.)

11.4.1 Resetting the File Position Pointer

When retrieving data sequentially from a file, a program normally reads from the beginning of the file until the desired data is found. In some cases, a program must process a file sequentially several times from the beginning. The statement

```
rewind(cfPtr);
```

repositions the **file position pointer** to the beginning (byte 0) of the file pointed to by `cfPtr`. The file position pointer is not really a pointer. It's an integer indicating the byte number of the next byte to read or write. This is sometimes referred to as the **file offset**. The file position pointer is a member of the `FILE` structure associated with each file.

11.4.2 Credit Inquiry Program

The program of Fig. 11.3 allows a credit manager to obtain lists of customers with:

- zero balances—customers who do not owe any money,
- credit balances—customers to whom the company owes money, or
- debit balances—customers who owe money for goods and services received.

A credit balance is a negative amount, and a debit balance is a positive amount. The program displays a menu and allows the credit manager to enter one of four options:

- 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.

```
1  // fig11_03.c
2  // Credit inquiry program
3  #include <stdbool.h>
4  #include <stdio.h>
5
6  enum Options {ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END};
7
8  // determine whether to display a record
9  bool shouldDisplay(enum Options option, double balance) {
10    if ((option == ZERO_BALANCE) && (balance == 0)) {
11       return true;
12    }
13
14    if ((option == CREDIT_BALANCE) && (balance < 0)) {
15       return true;
16    }
17
18    if ((option == DEBIT_BALANCE) && (balance > 0)) {
19       return true;
20    }
21
22    return false;
23 }
24
25 int main(void) {
26    FILE *cfPtr = NULL; // clients.txt file pointer
27
28    // fopen opens the file; exits program if file cannot be opened
29    if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
30       puts("File could not be opened");
31    }
32    else {
33       // display request options
34       printf("%s", "Enter request\n"
35          " 1 - List accounts with zero balances\n"
36          " 2 - List accounts with credit balances\n"
37          " 3 - List accounts with debit balances\n"
38          " 4 - End of run\n? ");
39       int request = 0;
40       scanf("%d", &request);
41
```

Fig. 11.3 | Credit inquiry program. (Part 1 of 3.)

```
42         // display records
43         while (request != END) {
44            switch (request) {
45               case ZERO_BALANCE:
46                  puts("\nAccounts with zero balances:");
47                  break;
48               case CREDIT_BALANCE:
49                  puts("\nAccounts with credit balances:");
50                  break;
51               case DEBIT_BALANCE:
52                  puts("\nAccounts with debit balances:");
53                  break;
54            }
55
56            int account = 0;
57            char name[30] = "";
58            double balance = 0.0;
59
60            // read account, name and balance from file
61            fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
62
63            // read file contents (until eof)
64            while (!feof(cfPtr)) {
65               // output only if balance is 0
66               if (shouldDisplay(request, balance)) {
67                  printf("%-10d%-13s%7.2f\n", account, name, balance);
68               }
69
70               // read account, name and balance from file
71               fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
72            }
73
74            rewind(cfPtr); // return cfPtr to beginning of file
75
76            printf("%s", "\n? ");
77            scanf("%d", &request);
78         }
79
80         puts("End of run.");
81         fclose(cfPtr); // close the file
82      }
83   }
```

```
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
```

Fig. 11.3 | Credit inquiry program. (Part 2 of 3.)

```
? 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. 11.3 | Credit inquiry program. (Part 3 of 3.)

Updating a Sequential File

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

```
300 White 0.00
```

If you were to rewrite the record beginning at the same location in the file using the new name, the record would be

```
300 Worthington 0.00
```

The new record 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 records can vary in size. For example, the values 7, 14, –117, 2074 and 27383 are all `int`s stored internally in the same number of bytes, but they're different-sized fields when displayed on the screen or written to a file as text.

So, sequential access with `fprintf` and `fscanf` typically is not used to update records in place. Instead, the entire file is rewritten. In a sequential-access file, we'd make the preceding name change by

- copying the records before `300 White 0.00` to a new file,
- writing the new record,
- copying the records after `300 White 0.00` to the new file, then
- replacing the old file with the new one.

This requires processing every record in the file to update one record.

Self Check

1 ***(Fill-In)*** Function `fscanf` is equivalent to function `scanf`, but `fscanf` receives as an argument a(n) _____.
Answer: file pointer for the file from which to read data.

2 ***(True/False)*** Function `feof` returns true only after the program attempts to read the nonexistent data following the last line.
Answer: *True*.

3 ***(Fill-In)*** The following statement repositions a file's _____ to the file's byte `0`.

```
rewind(cfPtr);
```

Answer: file position pointer.

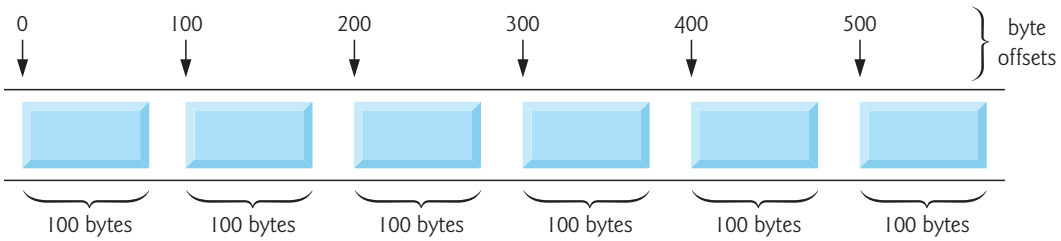
4 ***(True/False)*** In the formatted input/output model using `fprintf` and `fscanf`, fields—and hence records—are fixed in size.
Answer: *False*. Actually, in this model, fields—and hence records—can vary in size.

11.5 Random-Access Files

Records that you create with the formatted output function `fprintf` may vary in length. A **random-access file**, on the other hand, uses *fixed-length* records that may be accessed directly (and thus quickly) without searching through other records. This makes random-access files appropriate for **transaction-processing systems** that require rapid access to specific data, such as airline reservation systems, banking systems and point-of-sale systems. There are other ways to implement 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, each record's exact location 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.

The following diagram 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. Each car in the train has the same length.



Fixed-length records enable a program to insert data in a random-access file *without destroying other data in the file*. Data stored previously also can be updated or deleted without rewriting the entire file. In the sections that follow, we explain how to

- create a random-access file,
- enter data,
- read the data both sequentially and randomly,
- update the data, and
- delete data that's no longer needed.

Self Check

1 *(True/False)* Individual records that you write to and read from a random-access file may be accessed directly without searching through other records. This makes random-access files appropriate for systems that require rapid access to specific data.
Answer: *True*.

2 *(Fill-In)* A random-access file uses fixed-length records, so the exact location of a record relative to the beginning of the file can be calculated based on the _____.
Answer: record key.

11.6 Creating a Random-Access File

Function `fwrite` writes a specified number of bytes from a specified location in memory to a file. The data is written at the file position pointer's current location. Function `fread` reads a specified number of bytes from the file position pointer's current location to a specified area in memory. Writing a four-byte integer with

```
fprintf(fPtr, "%d", number);
```

could output as many as 11 digits—10 digits plus a sign, each of which requires at least one byte of storage, based on the character set for the locale. With random-access files, the statement

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

always writes four bytes (on a system with four-byte integers) from the `int` variable `number` to the file represented by `fPtr`. We'll explain the argument `1` in a moment. Later, we can use `fread` to read those four bytes into an `int` variable. Although `fread` and `fwrite` read and write data in fixed-size rather than variable-size format, they process data as "raw" bytes, rather than in `printf`'s and `scanf`'s human-readable text format. The "raw" data representation is system-dependent, so "raw" data may not be readable on other systems, or by programs produced by other compilers or with different compilation options.

`fwrite` and `fread` Can Write and Read Arrays

Functions `fwrite` and `fread` can write and read arrays. The third argument of both `fwrite` and `fread` is the number of elements to write or read. The preceding `fwrite` function call writes a single integer to a file, so the third argument is `1`—as if we were writing one array element. 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.

Problem Statement

Consider the following problem statement:

> *Create a transaction-processing system capable of storing up to 100 fixed-length records. Each record should have an account number (the record key), a last name, a first name and a balance. The program should use a random-access file and should be able to update an account, insert a new account, delete an account and list all the records in a formatted text file for printing.*

The next several sections introduce the techniques we'll use to create the transaction-processing program. Figure 11.4 shows how to open a random-access file, define a record format using a `struct`, write data to the file and close the file. This program initializes all 100 records of the file `"accounts.dat"` with empty `structs` using the function `fwrite`. Each empty `struct` contains the account number `0`, empty strings (`""`) for the last and first names, and the balance `0.0`. We initialize all the records to create the space in which the file will be stored and to make it possible to determine whether a record contains data.

```
1  // fig11_04.c
2  // Creating a random-access file sequentially
3  #include <stdio.h>
4
5  // clientData structure definition
6  struct clientData {
7     int account;
8     char lastName[15];
9     char firstName[10];
10    double balance;
11 };
12
13 int main(void) {
14    FILE *cfPtr = NULL; // accounts.dat file pointer
15
16    // fopen opens the file; exits if file cannot be opened
17    if ((cfPtr = fopen("accounts.dat", "wb")) == NULL) {
18       puts("File could not be opened.");
19    }
20    else {
21       // create clientData with default information
22       struct clientData blankClient = {0, "", "", 0.0};
23
24       // output 100 blank records to file
25       for (int i = 1; i <= 100; ++i) {
26          fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
27       }
28
29       fclose (cfPtr); // fclose closes the file
30    }
31 }
```

Fig. 11.4 | Creating a random-access file sequentially.

Line 17 opens the file "accounts.dat" for writing in binary mode ("wb"). Function fwrite (line 26) writes a block of bytes to a file. The arguments are:

- &blankClient—the address of the object to write,
- sizeof(struct clientData)—the size in bytes of the object to write,
- 1—the number of objects of that size to write, and
- cfPtr—a FILE * representing the file in which the bytes will be stored.

Recall that sizeof returns the size in bytes of its operand, struct clientData.

Writing an Array of Objects

In line 26, fwrite writes one object that's not an array element. To write an array, pass it to fwrite as the first argument and specify as the third argument the number of elements to output.

Self Check

1 *(True/False)* For a four-byte int variable number, the following statement always writes four bytes, even if the number's text representation could be as many as 11 digits:

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

Answer: *True*.

2 *(Fill-In)* Functions fread and fwrite read and write data in "raw data" format—that is, as _____ of data.
Answer: bytes.

3 *(Fill-In)* Function fwrite can write several array elements. In the call to fwrite, specify a pointer to an array and _____ as the first and third arguments.
Answer: the number of elements to write.

11.7 Writing Data Randomly to a Random-Access File

[*Note:* Figures 11.5, 11.6 and 11.7 use the data file created in Fig. 11.4, so you must run Fig. 11.4 before Figs. 11.5, 11.6 and 11.7.]

Figure 11.5 writes data to the file "accounts.dat". It uses fseek and fwrite to store data at specific locations in the file. Function fseek sets the file position pointer to a specific byte position, then fwrite writes the data there.

```
1  // fig11_05.c
2  // Writing data randomly to a random-access file
3  #include <stdio.h>
4
```

Fig. 11.5 | Writing data randomly to a random-access file. (Part 1 of 3.)

```
// clientData structure definition
struct clientData {
   int account;
   char lastName[15];
   char firstName[10];
   double balance;
}; // end structure clientData

int main(void) {
   FILE *cfPtr = NULL; // 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};

      // require user to specify account number
      printf("%s", "Enter account number (1 to 100, 0 to end input): ");
      scanf("%d", &client.account);

      // user enters information, which is copied into file
      while (client.account != 0) {
         // user enters last name, first name and balance
         printf("%s", "Enter lastname, firstname, balance: ");

         // set record lastName, firstName and balance value
         fscanf(stdin, "%14s%9s%lf", client.lastName,
            client.firstName, &client.balance);

         // seek position in file to user-specified record
         fseek(cfPtr, (client.account - 1) *
            sizeof(struct clientData), SEEK_SET);

         // write user-specified information in file
         fwrite(&client, sizeof(struct clientData), 1, cfPtr);

         // enable user to input another account number
         printf("%s", "\nEnter account number: ");
         scanf("%d", &client.account);
      }

      fclose(cfPtr); // fclose closes the file
   }
}
```

```
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
```

Fig. 11.5 | Writing data randomly to a random-access file. (Part 2 of 3.)

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

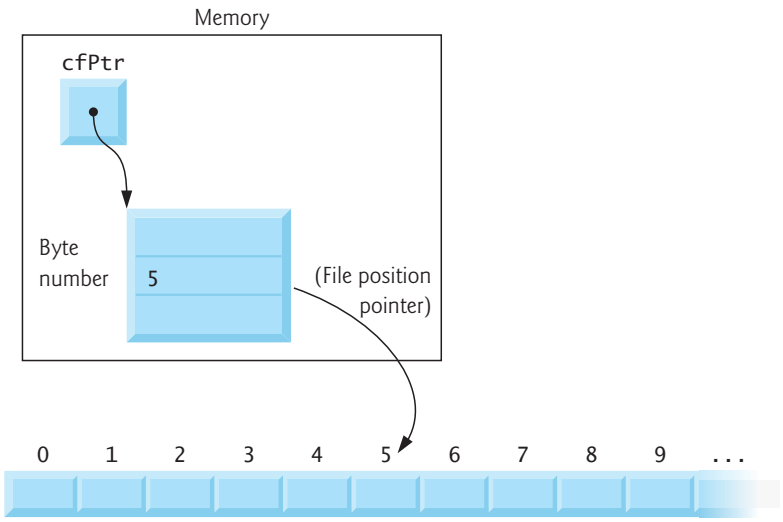
11.7.1 Positioning the File Position Pointer with `fseek`

Lines 38–39 position the file position pointer for the file referenced by `cfPtr` to the byte location calculated by

```
(client.account - 1) * sizeof(struct clientData)
```

This expression's value is the **offset** or **displacement**. In this example, the account number is 1–100. The file starts with byte `0`, so we subtract 1 from the account number when calculating the record's byte location. For record `1`, lines 38–39 set the file position pointer to byte `0` of the file. The symbolic constant **`SEEK_SET`** indicates that `fseek` should move the file position pointer relative to the beginning of the file.

The following diagram illustrates the `FILE` pointer referring to a `FILE` structure in memory. The file position pointer in this diagram indicates that the next byte to be read or written is byte number `5`.



`fseek` Function Prototype

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`. Positive `offsets` seek forward, and negative `offsets` seek backward. The argument `whence` can be `SEEK_SET`, `SEEK_CUR` or `SEEK_END` (all defined in `<stdio.h>`), which indicate the location from which the seek begins:

- `SEEK_SET` indicates that the seek is measured from the *beginning* of the file.
- `SEEK_CUR` indicates that the seek is measured from the *current location* in the file.
- `SEEK_END` indicates that the seek is measured from the *end* of the file.

You should use only positive offsets with `SEEK_SET` and only negative ones with `SEEK_END`.

11.7.2 Error Checking

For simplicity, the programs in this chapter do *not* perform error checking. Industrial-strength programs should determine whether functions such as `fscanf` (Fig. 11.5, lines 34–35), `fseek` (lines 38–39) and `fwrite` (line 42) operate correctly by checking their return values. 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 (e.g., attempting to seek to a position before the start of the file). 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.

Self Check

1 *(Fill-In)* Function _____ sets the file position pointer to a specific byte position in the file.
Answer: `fseek`.

2 *(Fill-In)* The symbolic constant _____ indicates that the file position pointer should be positioned relative to the beginning of the file.
Answer: `SEEK_SET`.

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. It reads bytes from the location specified by the file position pointer.

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. 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. If this number is less than the function call's third argument, a read error occurred.

Figure 11.6 sequentially reads each record in the "accounts.dat" file, determines whether it contains data and, if so, displays the formatted data. Function feof determines when the end of the file is reached, and the fread function (lines 28–29) transfers data from the file to the clientData structure client.

```
1  // fig11_06.c
2  // Reading a random-access file sequentially
3  #include <stdio.h>
4
5  // clientData structure definition
6  struct clientData {
7     int account;
8     char lastName[15];
9     char firstName[10];
10    double balance;
11 };
12
13 int main(void){
14    FILE *cfPtr = NULL; // accounts.dat file pointer
15
16    // fopen opens the file; exits if file cannot be opened
17    if ((cfPtr = fopen("accounts.dat", "rb")) == NULL) {
18       puts("File could not be opened.");
19    }
20    else {
21       printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
22          "First Name", "Balance");
23
24       // read all records from file (until eof)
25       while (!feof(cfPtr)) {
26          // read a record
27          struct clientData client = {0, "", "", 0.0};
28          size_t result =
29             fread(&client, sizeof(struct clientData), 1, cfPtr);
30
31          // display record
32          if (result != 0 && client.account != 0) {
33             printf("%-6d%-16s%-11s%10.2f\n", client.account,
34                client.lastName, client.firstName, client.balance);
35          }
36       }
37
38       fclose(cfPtr); // fclose closes the file
39    }
40 }
```

Fig. 11.6 | Reading a random-access file sequentially. (Part 1 of 2.)

```
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. 11.6 | Reading a random-access file sequentially. (Part 2 of 2.)

Self Check

1 *(True/False)* Function `fread` returns the number of bytes it successfully input.
Answer: *False*. Function `fread` returns the number of *items* it successfully input. Each item can be many bytes. If the number of items is fewer than `fread`'s third argument, then the read operation did not complete successfully.

2 *(True/False)* 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.
Answer: *True*.

11.9 Case Study: Transaction-Processing System

Let's create a transaction-processing program (Fig. 11.7) 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 current accounts in a text file for printing. We assume that the program of Fig. 11.4 created the file `accounts.dat`.

Option 1: Create a Formatted List of Accounts

The program has five options—option 5 terminates the program. Option 1 calls function `textFile` (lines 58–86) to store a formatted account report in a text file called `accounts.txt`, which can be printed later. The function uses `fread` and the sequential file-access techniques shown in Fig. 11.6. After option 1, `accounts.txt` contains:

```
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
```

Option 2: Update an Account

Option 2 calls the function `updateRecord` (lines 89–125) to update an account. The function updates only a record that already exists, so the function first checks whether the record specified by the user is empty. First, we read the record into structure `client` with `fread`. If the member `account` is 0, the record contains no information. So,

the program displays a message that the record is empty, then redisplays the menu choices. If the record contains information, function `updateRecord` inputs the transaction amount, calculates the new balance and rewrites the record to the file. A typical output for option 2 is

```
Enter account to update (1 - 100): 37
37  Barker          Doug             0.00

Enter charge (+) or payment (-): +87.99
37  Barker          Doug            87.99
```

Option 3: Create a New Account

Option 3 calls the function `newRecord` (lines 128–161) 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. 11.5. A typical output for option 3 is

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Option 4: Delete an Account

Option 4 calls function `deleteRecord` (lines 164–190) 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.

Code for the Transaction-Processing Program

The program is shown in Fig. 11.7. The file `"accounts.dat"` is opened for update (reading and writing) using `"rb+"` mode.

```
1  // fig11_07.c
2  // Transaction-processing program reads a random-access file sequentially,
3  // updates data already written to the file, creates new data to
4  // be placed in the file, and deletes data previously stored in the file.
5  #include <stdio.h>
6
7  // clientData structure definition
8  struct clientData {
9     int account;
10    char lastName[15];
11    char firstName[10];
12    double balance;
13 };
```

Fig. 11.7 | Transaction-processing program. (Part 1 of 5.)

```
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
22 int main(void) {
23    FILE *cfPtr = NULL; // accounts.dat file pointer
24
25    // fopen opens the file; exits if file cannot be opened
26    if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL) {
27       puts("File could not be opened.");
28    }
29    else {
30       int choice = 0; // user
31
32       // enable user to specify action
33       while ((choice = enterChoice()) != 5) {
34          switch (choice) {
35             case 1: // create text file from record file
36                textFile(cfPtr);
37                break;
38             case 2: // update record
39                updateRecord(cfPtr);
40                break;
41             case 3: // create record
42                newRecord(cfPtr);
43                break;
44             case 4: // delete existing record
45                deleteRecord(cfPtr);
46                break;
47             default: // display message for invalid choice
48                puts("Incorrect choice");
49                break;
50          }
51       }
52
53       fclose(cfPtr); // fclose closes the file
54    }
55 }
56
57 // create formatted text file for printing
58 void textFile(FILE *readPtr) {
59    FILE *writePtr = NULL; // accounts.txt file pointer
60
61    // fopen opens the file; exits if file cannot be opened
62    if ((writePtr = fopen("accounts.txt", "w")) == NULL) {
63       puts("File could not be opened.");
64    }
```

Fig. 11.7 | Transaction-processing program. (Part 2 of 5.)

```
65      else {
66         rewind(readPtr); // sets pointer to beginning of file
67         fprintf(writePtr, "%-6s%-16s%-11s%10s\n",
68            "Acct", "Last Name", "First Name","Balance");
69
70         // copy all records from random-access file into text file
71         while (!feof(readPtr)) {
72            // create clientData with default information
73            struct clientData client = {0, "", "", 0.0};
74            size_t result =
75               fread(&client, sizeof(struct clientData), 1, readPtr);
76
77            // write single record to text file
78            if (result != 0 && client.account != 0) {
79               fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n", client.account,
80                  client.lastName, client.firstName, client.balance);
81            }
82         }
83
84         fclose(writePtr); // fclose closes the file
85      }
86   }
87
88   // update balance in record
89   void updateRecord(FILE *fPtr) {
90      // obtain number of account to update
91      printf("%s", "Enter account to update (1 - 100): ");
92      int account = 0; // account number
93      scanf("%d", &account);
94
95      // move file pointer to correct record in file
96      fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
97
98      // read record from file
99      struct clientData client = {0, "", "", 0.0};
100     fread(&client, sizeof(struct clientData), 1, fPtr);
101
102     // display error if account does not exist
103     if (client.account == 0) {
104        printf("Account #%d has no information.\n", account);
105     }
106     else { // update record
107        printf("%-6d%-16s%-11s%10.2f\n\n", client.account, client.lastName,
108           client.firstName, client.balance);
109
110        // request transaction amount from user
111        printf("%s", "Enter charge (+) or payment (-): ");
112        double transaction = 0.0; // transaction amount
113        scanf("%lf", &transaction);
114        client.balance += transaction; // update record balance
115
```

Fig. 11.7 | Transaction-processing program. (Part 3 of 5.)

```
116         printf("%-6d%-16s%-11s%10.2f\n", client.account, client.lastName,
117            client.firstName, client.balance);
118
119         // move file pointer to correct record in file
120         fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
121
122         // write updated record over old record in file
123         fwrite(&client, sizeof(struct clientData), 1, fPtr);
124      }
125   }
126
127   // create and insert record
128   void newRecord(FILE *fPtr) {
129      // obtain number of account to create
130      printf("%s", "Enter new account number (1 - 100): ");
131      int account = 0; // account number
132      scanf("%d", &account);
133
134      // move file pointer to correct record in file
135      fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
136
137      // read record from file
138      struct clientData client = {0, "", "", 0.0};
139      fread(&client, sizeof(struct clientData), 1, fPtr);
140
141      // display error if account already exists
142      if (client.account != 0) {
143         printf("Account #%d already contains information.\n",
144            client.account);
145      }
146      else { // create record
147         // user enters last name, first name and balance
148         printf("%s", "Enter lastname, firstname, balance\n? ");
149         scanf("%14s%9s%lf", &client.lastName, &client.firstName,
150            &client.balance);
151
152         client.account = account;
153
154         // move file pointer to correct record in file
155         fseek(fPtr, (client.account - 1) * sizeof(struct clientData),
156            SEEK_SET);
157
158         // insert record in file
159         fwrite(&client, sizeof(struct clientData), 1, fPtr);
160      }
161   }
162
163   // delete an existing record
164   void deleteRecord(FILE *fPtr) {
165      // obtain number of account to delete
166      printf("%s", "Enter account number to delete (1 - 100): ");
167      int account = 0; // account number
168      scanf("%d", &account);
```

Fig. 11.7 | Transaction-processing program. (Part 4 of 5.)

```
169
170    // move file pointer to correct record in file
171    fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
172
173    // read record from file
174    struct clientData client = {0, "", "", 0.0};
175    fread(&client, sizeof(struct clientData), 1, fPtr);
176
177    // display error if record does not exist
178    if (client.account == 0) {
179       printf("Account %d does not exist.\n", account);
180    }
181    else { // delete record
182       // move file pointer to correct record in file
183       fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
184
185       struct clientData blankClient = {0, "", "", 0}; // blank client
186
187       // replace existing record with blank record
188       fwrite(&blankClient, sizeof(struct clientData), 1, fPtr);
189    }
190 }
191
192 // enable user to input menu choice
193 int enterChoice(void) {
194    // display available options
195    printf("%s", "\nEnter your choice\n"
196       "1 - store a formatted text file of accounts called\n"
197       "    \"accounts.txt\" for printing\n"
198       "2 - update an account\n"
199       "3 - add a new account\n"
200       "4 - delete an account\n"
201       "5 - end program\n? ");
202
203    int menuChoice = 0; // variable to store user
204    scanf("%d", &menuChoice); // receive choice from user
205    return menuChoice;
206 }
```

Fig. 11.7 | Transaction-processing program. (Part 5 of 5.)

Related Exercises

This Transaction-Processing System case study is supported by Exercise 11.11 (Hardware Inventory) and Exercise 11.17 (Modified Transaction-Processing System).

✓ Self Check

1 *(Discussion)* In the following code, what does the `if` statement's condition test?

```
if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL)
```

Answer: The condition tests whether the file accounts.dat was opened successfully in binary mode for reading and writing.

2 *(Discussion)* What does the following statement do in the program of Fig. 11.7?

```
fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
```

Answer: This statement moves the file position pointer for the file that `fPtr` represents to the position for the `clientData` record `account`.

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 C standard's Annex K provides 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`. As with `scanf_s` and `printf_s`, Microsoft's versions of `fprintf_s` and `fscanf_s` differ from those in Annex K.

Chapter 9 of the *SEI CERT C Coding Standard*

Chapter 9 of the *SEI CERT 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 `https://wiki.sei.cmu.edu/`:

- FIO03-C: When opening a file for writing using the nonexclusive file-open modes discussed in this chapter, 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 *exclusive write mode* (discussed in Section 11.3), which allows `fopen` to open the file *only* if it does *not* already exist.
- 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 nonzero value if it fails.
- 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.
- 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.

Self Check

1 *(Fill-In)* When you open a file for writing, you can ensure that an existing file is not truncated by using _____, which allows `fopen` to open the file only if it does not exist.
Answer: exclusive write mode.

2 *(True/False)* Function `rewind` does not return a value, so you cannot test whether the operation was successful. Instead, use function `fseek` because it returns a nonzero value if it fails.
Answer: *True*.

3 *(True/False)* Many platforms allow only a limited number of files to be open at once. So, you should always close a file as soon as it's no longer needed.
Answer: *True*.

Summary

Section 11.1 Introduction

- **Files** (p. 594) are used for permanent retention of large amounts of data.
- Computers store files on **secondary storage devices**, such as solid-state drives, flash drives and hard drives.

Section 11.2 Files and Streams

- C views each file as a sequential **stream of bytes** (p. 594). When a file is opened, a stream is associated with the file.
- Three streams are automatically opened when program execution begins—the **standard input** (p. 595), the **standard output** (p. 595) and the **standard error** (p. 595).
- Streams provide **communication channels** between files and programs.
- 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>`; p. 595) that contains information used to process the file. This structure includes a **file descriptor** (p. 595)—an index into an operating-system array called the **open file table** (p. 595). Each array element contains a **file control block** (**FCB**; p. 595) that the operating system uses to administer a particular file.
- The standard input, standard output and standard error are manipulated using the predefined file pointers **`stdin`**, **`stdout`** and **`stderr`**.
- Function **`fgetc`** (p. 595) **reads one character** from a file. It receives as an argument a `FILE` pointer for the file from which a character will be read.
- Function **`fputc`** (p. 595) **writes one character** to a file. It receives as arguments a character to be written and a `FILE` pointer for the file to which the character will be written.
- Functions **`fgets`** and **`fputs`** (p. 595) **read a line from a file** or **write a line to a file**, respectively.

Section 11.3 Creating a Sequential-Access File

- C imposes no structure on a file. You must provide a file structure to meet the requirements of a particular application.
- A C program administers each file with a separate `FILE` structure.
- Each open file must have a separately declared **`FILE` pointer** that's used to refer to the file.
- Function **`fopen`** (p. 597) takes as arguments a filename and a **file-open mode** (p. 597) and returns a pointer to the `FILE` structure for the file opened or `NULL` if the file could not be opened.
- The **file-open mode `"w"`** is used to open a file for writing. If the file does not exist, `fopen` creates it. If the file exists, the contents are discarded without warning.
- Function **`feof`** (p. 598) receives a pointer to a `FILE` and returns a nonzero (true) value when the end-of-file indicator has been set; otherwise, the function returns zero. Any attempt to read from a file for which `feof` returns true will fail.
- Function **`fprintf`** (p. 598) is equivalent to `printf` but also receives as an argument a file pointer for the file to which the data will be written.
- Function **`fclose`** (p. 598) receives a file pointer as an argument and closes the specified file.
- When a file is opened, the file control block (FCB) for the file is copied into memory. The FCB is used by the operating system to administer the file.
- 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 for reading and writing, use an **update mode**—**`"r+"`**, **`"w+"`** or **`"a+"`**. Mode `"r+"` opens a file for reading and writing. Mode `"w+"` creates a file for reading and writing, but an existing file's contents are discarded. 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** (**`b`**) for manipulating **binary files**.
- **Exclusive write mode** ensures that an existing file is not overwritten. If your program successfully opens a file in exclusive write mode and the underlying system supports exclusive file access, then only your program can access the file while it's open.

Section 11.4 Reading Data from a Sequential-Access File

- Function **`fscanf`** (p. 601) is equivalent to function `scanf` but receives as an argument a file pointer for the file from which the data is read.
- Function **`rewind`** repositions a program's **file position pointer** (p. 602) to the beginning of the file (i.e., byte 0) pointed to by its argument.
- The file position pointer is 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** (p. 602). The file position pointer is a member of the `FILE` structure associated with each file.
- The data in a sequential text file typically cannot be modified without the risk of destroying other data in the file.

Section 11.5 Random-Access Files

- **Random-access files** (p. 606) use **fixed-length** records that may be accessed directly without searching through other records.
- Every record in a random-access file normally has the same length, so the exact location of a record relative to the beginning of the file can be calculated as a function of the **record key**.
- Fixed-length records enable data to be inserted in a random-access file without destroying other data. Data stored previously can also be updated or deleted without rewriting the entire file.

Section 11.6 Creating a Random-Access File

- Function **`fwrite`** (p. 606) transfers a specified number of bytes beginning at a specified location in memory to a file. The data is written beginning at the file position pointer's location.
- Function **`fread`** (p. 606) 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.
- Functions `fwrite` and `fread` are capable of **reading and writing arrays of data** from and to files. The third argument of both `fread` and `fwrite` is the number of elements to process.
- File-processing programs normally write one `struct` at a time.

Section 11.7 Writing Data Randomly to a Random-Access File

- Function **`fseek`** (p. 609) repositions a file's file position pointer to a specific byte position. Its second argument indicates the number of bytes to seek, and its third argument indicates the location from which to seek. The third argument can be—`SEEK_SET`, `SEEK_CUR` or `SEEK_END`. **`SEEK_SET`** (p. 611) indicates that the seek starts at the beginning of the file; **`SEEK_CUR`** (p. 612) indicates that the seek starts at the current location in the file; and **`SEEK_END`** (p. 612) indicates that the seek is measured from the end of the file.
- Industrial-strength programs should determine whether functions such as `fscanf`, `fseek` and `fwrite` operate correctly by checking their return values.
- Function `fscanf` returns the number of fields 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.

Section 11.8 Reading Data from a Random-Access File

- Function `fread` reads a specified number of bytes from a file into memory.
- 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.
- Function `fread` returns the number of items it successfully input. If this number is less than the third argument in the function call, then a read error occurred.

Self-Review Exercises

11.1 Fill-In the blanks in each of the following:

a) Function _____ closes a file.
b) The _____ function reads data from a file in a manner similar to how `scanf` reads from `stdin`.
c) Function _____ reads a character from a specified file.
d) Function _____ reads a line from a specified file.
e) Function _____ opens a file.
f) Function _____ is normally used when reading data from a file in random-access applications.
g) Function _____ repositions the file position pointer to a specific location in the file.

11.2 State which of the following are *true* and which are *false*. If *false*, explain why.

a) Function `fscanf` cannot be used to read data from the standard input.

b) You must explicitly use `fopen` to open the standard input, standard output and standard error streams.
c) A program must explicitly call function `fclose` to close a file.
d) If the file position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
e) Function `fprintf` can write to the standard output.
f) Data in sequential-access files can be updated without overwriting other data.
g) It's not necessary to search through all the records in a random-access file to find a specific record.
h) Records in random-access files are not of uniform length.
i) Function `fseek` may seek only relative to the beginning of a file.

11.3 Write a single statement to accomplish each of the following. Assume that each of these statements applies to the same program.

a) Open the file `"oldmast.dat"` for reading and assign the returned file pointer to `ofPtr`.
b) Open the file `"trans.dat"` for reading and assign the returned file pointer to `tfPtr`.
c) Open the file `"newmast.dat"` for writing (and creation) and assign the returned file pointer to `nfPtr`.
d) Read a record from the file `"oldmast.dat"`. The record consists of integer `account`, string `name` and floating-point `currentBalance`.
e) Read a record from the file `"trans.dat"`. The record consists of the integer `account` and floating-point `dollarAmount`.
f) Write a record to the file `"newmast.dat"`. The record consists of the integer `account`, string `name` and floating-point `currentBalance`.

11.4 Find the error in each of the following and explain how to correct it.

a) The file referred to by `fPtr` (`"payables.dat"`) has not been opened.
```
printf(fPtr, "%d%s%d\n", account, company, amount);
```
b) `open("receive.dat", "r+");`
c) The following should read a record from `"payables.dat"`. File pointer `payPtr` refers to this file, and file pointer `recPtr` refers to the file `"receive.dat"`:
```
scanf(recPtr, "%d%s%d\n", &account, company, &amount);
```
d) The file `"tools.dat"` should be opened to add data to the file without discarding the current data.
```
if ((tfPtr = fopen("tools.dat", "w")) != NULL)
```
e) The file `"courses.dat"` should be opened for appending without modifying the current contents of the file.
```
if ((cfPtr = fopen("courses.dat", "w+")) != NULL)
```

Answers to Self-Review Exercises

11.1 a) `fclose`. b) `fscanf`. c) `fgetc`. d) `fgets`. e) `fopen`. f) `fread`. g) `fseek`.

11.2 See the answers below:

a) *False*. Function `fscanf` can be used to read from the standard input by including the pointer to the standard input stream, `stdin`, in the call to `fscanf`.

b) *False*. These three streams are opened automatically by C when program execution begins.

c) *False*. The files will be closed when program execution terminates, but all files should be explicitly closed with `fclose`.

d) *False*. Function `rewind` can be used to reposition the file position pointer to the beginning of the file.

e) *True*.

f) *False*. In most cases, sequential file records are not of uniform length. Therefore, it's possible that updating a record will cause other data to be overwritten.

g) *True*.

h) *False*. Records in a random-access file are normally of uniform length.

i) *False*. It's possible to seek from the beginning of the file, from the end of the file and from the current location in the file.

11.3 a)
```
ofPtr = fopen("oldmast.dat", "r");
```
b)
```
tfPtr = fopen("trans.dat", "r");
```
c)
```
nfPtr = fopen("newmast.dat", "w");
```
d)
```
fscanf(ofPtr, "%d%s%f", &account, name, &currentBalance);
```
e)
```
fscanf(tfPtr, "%d%f", &account, &dollarAmount);
```
f)
```
fprintf(nfPtr, "%d %s %.2f", account, name, currentBalance);
```

11.4 See the answers below:

a) Error: `"payables.dat"` has not been opened before using `fPtr`.
Correction: Use `fopen` to open `"payables.dat"` for writing, appending or updating.

b) Error: Function `open` is not a Standard C function.
Correction: Use function `fopen`.

c) Error: The function `scanf` should be `fscanf`. Function `fscanf` uses the incorrect file pointer to refer to file `"payables.dat"`.
Correction: Use `payPtr` to refer to `"payables.dat"` and use `fscanf`.

d) Error: The contents of the file are discarded because the file is opened for writing (`"w"`).
Correction: To add data to the file, either open the file for updating (`"r+"`) or open the file for appending (`"a"` or `"a+"`).

e) Error: File `"courses.dat"` is opened for updating in `"w+"` mode, which discards the current contents of the file.
Correction: Open the file in `"a"` or `"a+"` mode.

Exercises

11.5 Fill-In the blanks in each of the following:

a) Large amounts of data are stored on secondary storage devices as _____.

b) A(n) _____ is composed of several fields.

c) To facilitate the retrieval of specific records from a file, one field in each record is chosen as a(n) _____.
d) A group of related characters that conveys meaning is called a(n) _____.
e) The file pointers for the three streams that are opened automatically when program execution begins are named _____, _____ and _____.
f) Function _____ writes a character to a specified file.
g) Function _____ writes a line to a specified file.
h) Function _____ is generally used to write data to a random-access file.
i) Function _____ repositions the file-position pointer to the beginning of the file.

11.6 ***(Creating Data for a File-Matching Program)*** Write a simple program to create some test data for checking out the program of Exercise 11.7. Use the following sample account data:

Master File: Account number	Name	Balance
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Transaction File: Account number	Dollar amount
100	27.14
300	62.11
400	100.56
900	82.17

11.7 ***(File Matching)*** Exercise 11.3 asked you to write a series of single statements. These statements form the core of an important type of file-processing program, namely, a file-matching program. In commercial data processing, it's common to have several files in each system. In an accounts-receivable system, for example, there's generally a master file containing detailed information about each customer such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

As transactions occur, such as sales and payments, they're entered into a file. At the end of each business period (i.e., a month for some companies, a week for others and a day in some cases), the file of transactions (called `"trans.dat"` in Exercise 11.3) is applied to the master file (called `"oldmast.dat"` in Exercise 11.3), to update each account's purchase and payment record. During an update, the master file is rewritten as a new file (`"newmast.dat"`), which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period, and therefore no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments might

have just moved to this community, and the company may not have had a chance to create a master record for this customer.

Use the statements in Exercise 11.3 as the basis for a complete file-matching accounts-receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential file with records stored in increasing account-number order.

When a match occurs (i.e., records with the same account number appear on both the master file and the transaction file), add the dollar amount on the transaction file to the current balance on the master file and write the `"newmast.dat"` record. (Assume that purchases are indicated by positive amounts on the transaction file and that payments are indicated by negative amounts.) When there's a master record for a particular account but no corresponding transaction record, merely write the master record to `"newmast.dat"`. When there's a transaction record but no corresponding master record, print the message `"Unmatched transaction record for account number ..."` (fill in the account number from the transaction record).

11.8 ***(Testing the File-Matching Exercises)*** Run the program of Exercise 11.7 using the files of test data created in Exercise 11.6. Check the results carefully.

11.9 ***(File Matching with Multiple Transactions)*** It's possible (actually common) to have several transaction records with the same record key. This occurs because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts-receivable file-matching program of Exercise 11.7 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of Exercise 11.6 to include the following additional transaction records:

Account number	Dollar amount
300	83.89
700	80.78
700	1.53

11.10 ***(Write Statements to Accomplish a Task)*** Write statements that accomplish each of the following. Assume that the structure

```
struct person {
   char lastName[15];
   char firstName[15];
   char age[4];
};
```

has been defined and that the file is already open for writing.

a) Initialize the file `"nameage.dat"` so that there are 100 records with `lastName = "unassigned"`, `firstname = ""` and `age = "0"`.

b) Input `10` last names, first names and ages, and write them to the file.

c) Update a record; if there's no information in the record, tell the user `"No info"`.

d) Delete a record that has information by reinitializing that particular record.

11.11 You are a math teacher in an elementary school, and you need to keep a record of the students' examination scores. The fields to keep are "Student ID," "Student name," and "Examination score" (a number between 0 to 100). Write a program for users to add and view records, storing it in "`math_exam_scores.dat`":

Use the following information to start your record:

Student ID	Student Name	Examination Score
17	Steven	85
19	John	70
25	Jane	95

11.12 ***(Telephone-Number Word Generator)*** Standard telephone keypads contain the digits 0–9. The numbers 2–9 each have three letters associated with them, as is indicated by the following table:

Digit	Letter	Digit	Letter
2	A B C	6	M N O
3	D E F	7	P R S
4	G H I	8	T U V
5	J K L	9	W X Y

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the above table to develop the seven-letter word `"NUMBERS"`.

Businesses frequently attempt to get telephone numbers that are easy for their clients to remember. If a business can advertise a simple word for its customers to dial, then, no doubt, the business will receive a few more calls.

Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its take-home business could surely do so with the number 825-3688 (i.e., `"TAKEOUT"`).

Each seven-digit phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of letters. It's possible, however, that the owner of a barbershop would be pleased to know that the shop's telephone number, 424-7288, corresponds to `"HAIRCUT"`. A florist would, no doubt, be delighted to find that the store's telephone number, 356-9377, corresponds to `"FLOWERS"`. A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters `"PETCARE"`.

Write a C program that, given a seven-digit number, writes to a file every possible seven-letter word corresponding to that number. There are 2187 (3 to the seventh power) such words. Avoid phone numbers with the digits 0 and 1.

11.13 ***(Project: Telephone-Number Word Generator Modification)*** If you have a computerized dictionary available, modify the program you wrote in Exercise 11.12 to look up the words in the dictionary. Some seven-letter combinations created by this program consist of two or more words (e.g., the phone number 843-2677 produces `"THEBOSS"`).

11.14 ***(Using File-Processing Functions with Standard Input/Output Streams)*** Modify the example of Fig. 8.8 to use functions `fgetc` and `fputs` rather than `getchar` and `puts`. The program should give the user the option to read from the standard input and write to the standard output or to read from a specified file and write to a specified file. If the user chooses the second option, have the user enter the filenames for the input and output files.

11.15 Write a program that simulates a fair die throw (numbered from 1 to 6), lets the user enter the number of throws, and summarizes the result in a frequency table. Write the results into a file "`die_throw.dat`".

Example of a frequency table:

Number	Frequency
1	10
2	9
3	43
4	32
5	2
6	42

11.16 ***(Simpletron with File Processing)*** In Exercise 7.29, you wrote a software simulation of a computer that used a special machine language called Simpletron Machine Language (SML). In the simulation, each time you wanted to run an SML program, you entered the program into the simulator from the keyboard. If you made a mistake while typing the SML program, the simulator was restarted, and the SML code was reentered. It would be nice to be able to read the SML program from a file

rather than type it each time. This would reduce time and mistakes in preparing to run SML programs.

a) Modify the simulator you wrote in Exercise 7.29 to read SML programs from a file specified by the user at the keyboard.
b) After the Simpletron executes, it outputs the contents of its registers and memory on the screen. It would be nice to capture the output in a file, so modify the simulator to write its output to a file in addition to displaying it on the screen.

11.17 ***(Modified Transaction-Processing System)*** Modify the program of Section 11.9 to include an option that displays the list of accounts on the screen. Consider modifying function `textFile` to use either the standard output or a text file based on an additional function parameter that specifies where the output should be written.

11.18 ***(Project: Phishing Scanner)*** Phishing is a form of identity theft in which, in an e-mail, a sender posing as a trustworthy source attempts to acquire private information, such as your user names, passwords, credit-card numbers and Social Security number. Phishing e-mails claiming to be from popular banks, credit-card companies, auction sites, social networks and online payment services may look quite legitimate. These fraudulent messages often provide links to spoofed (fake) websites where you're asked to enter sensitive information.

Visit `https://snopes.com` and other websites to find lists of the top phishing scams. Also, check out the Anti-Phishing Working Group (`https://apwg.org/`), and the FBI's Cyber Investigations website (`https://www.fbi.gov/investigate/cyber`), where you'll find information about the latest scams and how to protect yourself.

Create a list of 30 words, phrases and company names commonly found in phishing messages. Assign a point value to each based on your estimate of its likeliness to be in a phishing message (e.g., one point if it's somewhat likely, two points if moderately likely, or three points if highly likely). Write a program that scans a file of text for these terms and phrases. For each occurrence of a keyword or phrase within the text file, add the assigned point value to the total points for that word or phrase. For each keyword or phrase found, output one line with the word or phrase, the number of occurrences and the point total. Then show the point total for the entire message. Does your program assign a high point total to some actual phishing e-mails you've received? Does it assign a high point total to some legitimate e-mails you've received?

AI Case Study: Intro to NLP—Who Wrote Shakespeare's Works?

11.19 ***(Intro to Natural Language Processing and Similarity Detection)*** Every day, we use **natural language** in various forms of communication, including:

- You read your text messages and check the latest news clips.
- You speak to family, friends and colleagues and listen to their responses.

- You have a hearing-impaired friend with whom you communicate via sign language and who enjoys close-captioned video programs.
- You have a blind colleague who reads braille, listens to audiobooks and listens to a screen reader speak about what's on the computer screen.
- You read e-mails, distinguishing junk from important communications.
- You receive a client e-mail in Spanish and run it through a free translation program, then respond in English, knowing that your client can easily translate your e-mail back to Spanish.
- You drive, observing road signs like "Stop," "Speed Limit 35" and "Road Under Construction."
- You give your car verbal commands, like "call home" or "play classical music," or ask questions like, "Where's the nearest gas station?"
- You teach a child how to speak and read.
- You learn a foreign language.

Natural Language Processing (NLP) helps computers understand, analyze and process human text and speech. Natural language processing is performed on text collections composed of Tweets, Facebook posts, conversations, movie reviews, Shakespeare's plays, historical documents, news items, meeting logs, and so much more. A text collection is known as a **corpus**, the plural of which is **corpora**.

Some key NLP applications include:

- **Natural language understanding**—understanding text content or spoken language.
- **Sentiment analysis**—determining whether text has positive, neutral or negative sentiment. For example, companies analyze the sentiment of tweets about their products.
- **Readability assessment**—determining how readable text is, based on the vocabulary used, word lengths, sentence lengths, sentence structure, topics covered and more. While writing this book, we used the paid NLP tool Grammarly[2] to help us tune the writing to ensure the text's readability for a wide audience.
- **Intelligent virtual assistants**—software that helps you perform everyday tasks. Popular intelligent virtual assistants include Amazon Alexa, Apple Siri, Microsoft Cortana and Google Assistant.
- **Text summarization**—summarizing the key points of a large text. This can save valuable time for busy people.
- **Speech recognition**—converting speech to text.
- **Speech synthesis**—converting text to speech.

2. Grammarly also has a free version (`https://www.grammarly.com`).

- **Language identification**—receiving a text when you don't know its language in advance then automatically determining the language.
- **Interlanguage translation**—converting text to other spoken languages.
- **Named-entity recognition**—locating and categorizing items like dates, times, quantities, places, people, things, organizations and more.
- **Chatbots**—AI-based software that humans interact with via natural language. One popular chatbot application is automated customer support.
- **Similarity detection**—examining documents to determine how alike they are. Basic similarity metrics include average sentence length, frequency distribution of sentence lengths, average word length, frequency distribution of word lengths, frequency distribution of word usage, and more.

Many lower-level NLP tasks support the applications above as they perform their tasks, including:

- **Tokenization**—splitting text into **tokens**, which are meaningful units, such as words and numbers.
- **Parts-of-speech (POS) tagging**—identifying each word's part of speech, such as noun, verb, adjective, etc.
- **Noun phrase extraction**—locating groups of words representing nouns, such as "red brick factory."[3]
- **Spell checking** and **spelling correction**.
- **Stemming**—reducing words to their stems by removing prefixes or suffixes. For example, the stem of "varieties" is "varieti."
- **Lemmatization**—like stemming, but produces real words based on the original words' context. For example, the lemmatized form of "varieties" is "variety."
- **Word frequency counting**—determining how often each word appears in a corpus.
- **Stop-word elimination**—removing common words, such as *a*, *an*, *the*, *I*, *we*, *you* and more to analyze the important words in a corpus.
- **n-grams**—producing sets of consecutive words in a corpus for use in identifying words that frequently appear adjacent to one another. n-grams are commonly used for predictive text input, such as when your smartphone suggests possible next words as you type a text message.

This case study exercise serves two purposes:

- First, it introduces the crucial AI subtopic of natural language processing, which will play a key role in the future of anyone learning programming today.

3. The phrase "red brick factory" illustrates why natural language is such a difficult subject. Is a "red brick factory" a factory that makes red bricks? Is it a red factory that makes bricks of any color? Is it a factory built of red bricks that makes products of any type? In today's music world, it could even be the name of a rock band or the name of a game on your smartphone.

- Second, it introduces the NLP subtopic of similarity detection, which you'll perform using straightforward array-, string- and file-processing techniques.

Project Gutenberg

A great source of text for analysis is the massive collection of free e-books at **Project Gutenberg**:

```
https://www.gutenberg.org
```

The site contains over 60,000 e-books in various formats, including plain-text files. These are out of copyright in the United States. For information about Project Gutenberg's Terms of Use and copyright in other countries, see:

```
https://www.gutenberg.org/policy/terms_of_use.html
```

For this case-study exercise, you'll use the plain-text e-book files for William Shakespeare's *Romeo and Juliet*:

```
https://www.gutenberg.org/ebooks/1513
```

and Christopher Marlowe's *Edward the Second*:

```
https://www.gutenberg.org/ebooks/20288
```

Each of these is available free for download at Project Gutenberg.

Downloading E-Books from Project Gutenberg

Project Gutenberg does not allow programmatic access to its e-books. You must download the books to your own system before analyzing them.[4] To download *Romeo and Juliet* as a plain-text file, right-click the **Plain Text UTF-8** link on the play's web page, then select

- **Save Link As...** (Chrome/Firefox/Microsoft Edge),
- **Download Linked File As...** (Safari), or

to save the play to the folder in which you'll place your solution to this exercise. Save the files with the names `RomeoAndJuliet.txt` and `EdwardTheSecond.txt`.

Who Wrote Shakespeare's Works?

Some people believe that William Shakespeare's works might have been penned by Christopher Marlowe, Sir Francis Bacon or others. You can learn more about this controversy at

```
https://en.wikipedia.org/wiki/Shakespeare_authorship_question
```

With some simple similarity-detection techniques, you can begin to compare Shakespeare's works with those of other authors. In this case-study exercise, your ultimate goal is to perform similarity detection between *Romeo and Juliet* and Christopher Marlowe's *Edward the Second* to determine whether Christopher Marlowe might have

4. "Information About Robot Access to our Pages." Accessed January 1, 2021. `https://www.gutenberg.org/policy/robot_access.html`.

authored Shakespeare's works. If you really get into this issue, you can explore more sophisticated similarity-detection techniques.

Analyzing *Romeo and Juliet* to Prepare for Simple Similarity Detection

You'll now perform some simple statistical analysis as a basis for determining document similarity. You'll begin by focusing on Shakespeare's *Romeo and Juliet*. Later, you'll perform the same tasks on *Edward the Second*, then compare your analyses' results. As a control, you also might want to analyze a play from a third author who is not involved in this controversy. You'll track the following items as you read and process *Romeo and Juliet*, then use them to display various statistics:

- The total number of sentences.
- The total number of words.
- The total number of characters.
- The number of sentences of each length.
- The number of words of each length.
- The unique words' frequencies.

Cleaning *Romeo and Juliet* Before Analyzing It

Data does not always come ready for analysis. It could, for example, be in the wrong format. Data scientists spend a large portion of their time preparing data before performing analyses. Preparing data for analysis is called **data munging** or **data wrangling**.

Each e-book you download from Project Gutenberg contains information and legal paragraphs that you will not want to include in your analyses. You should open *Romeo and Juliet* in a text editor and "clean" it by removing the Project Gutenberg text. In particular, remove everything from the beginning of the document up to and including the title "THE TRAGEDY OF ROMEO AND JULIET," then remove everything from the following text through the end of the file:

```
End of the Project Gutenberg EBook of Romeo and Juliet,
by William Shakespeare
```

You should do some additional text cleaning with a text editor before running your analytics on the play:

- **Each character's name is mentioned each time that character speaks**—this is standard in plays. You don't need the characters' names for the particular analytics you're going to run in this case study. In fact, they'll "get in the way." For more sophisticated similarity detection, you might want to keep them.
- **Plays also include many staging directions indicating when characters should enter and leave the stage, duel one another, fall down and die when poisoned, and the like**. These directions also should be removed.

You could write a program to handle these cleaning chores. Be careful, though—scrutinizing the manuscript may reveal many special cases that your code would need to handle. Programming for them could be time-consuming and error-prone.

Arrays You'll Need to Perform Your Analysis

You are now ready to analyze *Romeo and Juliet* to create the statistics you'll use for simple similarity detection. Use three arrays to record various counts that characterize the text of the play.

- The **`sentenceLengths` array** will keep counts of how many sentences consist of one word, two words, three words, etc.
- The **`wordLengths` array** will keep counts of how many words consist of one character, two characters, three characters, etc.
- The **`wordFrequencies` array** contains a `struct` for each distinct word in the play. The `struct`'s members are the word and a count of how many times that word appears in the play. The word should be stored as a string in a fixed-length `char` array, which must be large enough to store the longest word in the play and its terminating null character.

Analyzing a Sentence

Consider the sentence:

"O Romeo, Romeo, wherefore art thou Romeo."

- The sentence has seven words, so your program would add `1` to `sentenceLengths[7]`.
- The first word ("O") has one letter, so your program would add `1` to `wordLengths[1]`.
- The second word ("Romeo") has five letters, so your program would add `1` to `wordLengths[5]`, and so on.

To perform word-frequency counting, convert the words to lowercase letters, so that all occurrences of the same word will compare as equal. As you process each word, search the `wordFrequencies` array to determine whether the word already is in the array. If so, add one to that word's count. Otherwise, place the word in the next empty `wordFrequencies` array element and set its count to `1`.

Implementing Your Analysis Code

Use the array-, string- and file-processing techniques you've learned to read the contents of *Romeo and Juliet* and perform the following tasks:

- Every sentence ends with a **sentence terminator**—a **period** (.), a **question mark** (?) or an **exclamation point** (!). Define a **`processSentence` function** that reads words until it encounters a sentence terminator. This function updates the sentence, word and character counters as you process each word. When you hit the end of a sentence, increment the appropriate counter in the `sentenceLengths` array and reset the word counter to zero.
- For every word, `processSentence` should call the **`processWord` function** to increment the appropriate counter in the `wordLengths` array, and either incre-

ment the word's counter in the `wordFrequencies` array or add the word to the `wordFrequencies` array with a count of `1`.

Remember to keep track of the total number of sentences, words and characters.

Analysis Report

Next, display the following statistics for *Romeo and Juliet*:

- The total number of sentences.
- The total number of words.
- The total number of characters.
- The mean (average) sentence length.
- The mean word length.
- The median sentence length.
- The median word length.
- A table of sentence lengths and their percentages among all sentence lengths.
- A table of word lengths and their percentages among all word lengths.
- A frequency-distribution table containing the play's unique words, their frequencies and their percentages among all words in the play—display these in descending order by frequency.

Your program also should output these statistics to a file to make it easier to study the results you produce and compare them between plays.

Analyzing Christopher Marlowe's Play *Edward the Second*

Now that you've analyzed *Romeo and Juliet*, use a text editor to clean Christopher Marlowe's play *Edward the Second*. As part of any data-science study, it's important to get to know your data. The conventions used in *Edward the Second* for specifying who's speaking and the play's staging directions are different from those you saw in *Romeo and Juliet*. So, be careful to observe these differences when cleaning *Edward the Second*. After you clean the text, run your analytics program on *Edward the Second*. Compare the analytics with those you produced for *Romeo and Juliet*. Comment on the similarities you find between these plays.

AI/Data-Science Case Study—Machine Learning with GNU Scientific Library

11.20 ***(Machine Learning with Simple Linear Regression)*** Machine learning is one of the most exciting and promising subfields of artificial intelligence. Our goal here is to give you a friendly, hands-on introduction to one of the simpler machine-learning techniques.

Prediction

Machine learning is typically used to make predictions, based on existing data—and often lots of it. Wouldn't it be fantastic if you could improve weather forecasting to

save lives, minimize injuries and property damage? What if we could improve cancer diagnoses and treatment regimens to save lives, or improve business forecasts to maximize profits and secure people's jobs? What about detecting fraudulent credit-card purchases and insurance claims? How about predicting customer "churn," what prices houses are likely to sell for, ticket sales of new movies, and more generally, anticipated revenue of new products and services? How about predicting the best strategies for coaches and players to use to win more games and championships? All of these kinds of predictions are happening today with machine learning.

GNU Scientific Library and gnuplot

In this case study, you'll **examine a completely coded program** that demonstrates the machine-learning technique called **simple linear regression**, performed with a function from the open-source **GNU Scientific Library**:

```
https://www.gnu.org/software/gsl/
```

This library defines many commonly used algorithms from engineering, science and mathematics. The program you'll study then passes commands to the 2D and 3D plotting application **gnuplot** to create several plot images. As you'll see, gnuplot uses its own plotting language different from C, so in our code, we provide extensive comments that explain the gnuplot commands.

Descriptive Statistics

In data science, you'll often use statistics to describe and summarize your data. Some basic **descriptive statistics** are:

- **minimum**—the smallest value in a collection of values.
- **maximum**—the largest value in a collection of values.
- **range**—the difference between the maximum and minimum values.
- **count**—the number of values in a collection.
- **sum**—the total of the values in a collection.

Measures of dispersion (also called **measures of variability**), such as *range*, determine how spread out values are. Other measures of dispersion include *variance* and *standard deviation*.[5,6,7]

Additional descriptive statistics include mean, median and mode, which we discussed in Section 6.9. These are **measures of central tendency**—each is a way of producing a single value that represents a "central" value in a set of values, i.e., one which is in some sense typical of the others.

5. "Understanding Descriptive Statistics." Accessed January 1, 2021. `https://towardsdatascience.com/understanding-descriptive-statistics-c9c2b0641291`.
6. "Standard deviation." Accessed January 1, 2021. `https://en.wikipedia.org/wiki/Standard_deviation`.
7. "Variance." Accessed January 1, 2021. `https://en.wikipedia.org/wiki/Variance`.

Anscombe's Quartet

An important step in data analytics is "getting to know your data." The *basic descriptive statistics* above certainly help you know more about your data. One caution, though, is that dramatically different datasets actually can have identical or nearly identical descriptive statistics. For an example of this phenomenon, we'll consider *Anscombe's Quartet*:

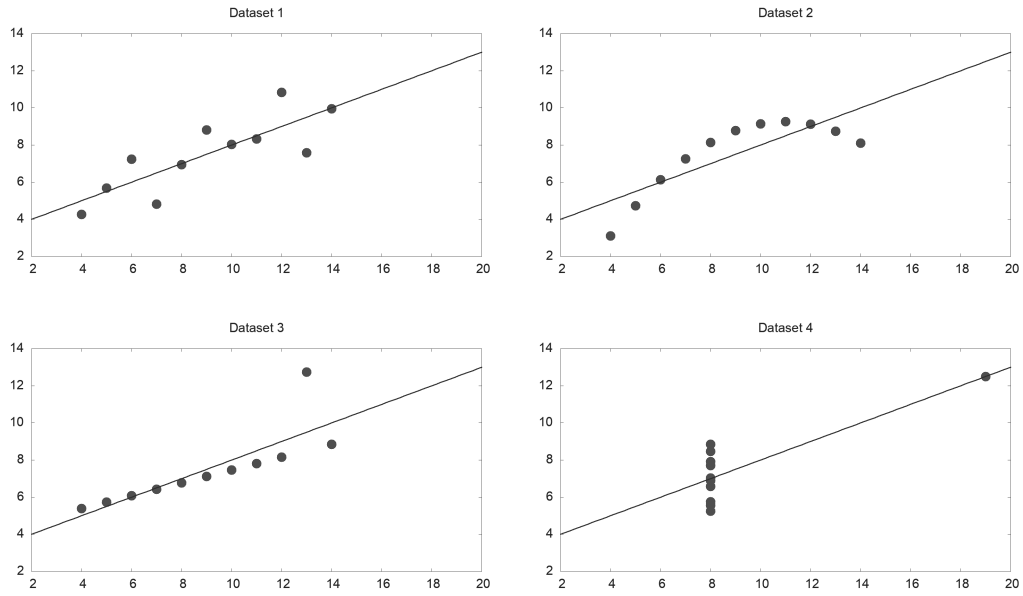
```
https://en.wikipedia.org/wiki/Anscombe%27s_quartet
```

Anscombe's Quartet consists of the following four sets of *x*-*y* coordinate pairs with 11 data samples each:

x1	y1	x2	y2	x3	y3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Interestingly, these datasets have *nearly identical* descriptive statistics. For instance, in all four datasets, the *x*- and *y*-coordinates' mean values are 9 and 7.5, respectively.

The following diagrams—which our **fully coded case study** example creates—plot the *x1*-*y1*, *x2*-*y2*, *x3*-*y3* and *x4*-*y4* data, respectively:



We'll discuss the lines (known as **regression lines**) shortly. As you can see in the **visualizations**—but not necessarily by simply looking at the data in the preceding table—these Anscombe's Quartet datasets are *significantly different*. Yet, like their descriptive statistics, their regression lines appear to be identical. This shows that you cannot just draw conclusions from descriptive statistics and regressions. You must use additional tools, like the visualizations above, to **get to know your data**.

Simple Linear Regression

Given a collection of points (*x*-*y* coordinate pairs) representing an **independent variable** (*x*) and a **dependent variable** (*y*), **simple linear regression** describes the **linear relationship** between the dependent and independent variables with a straight line, known as the **regression line**. The lines in the preceding diagrams are the regression lines for each of the four datasets in Anscombe's Quartet.

Consider the linear relationship between Celsius and Fahrenheit temperatures. Given a Celsius temperature, we can calculate the corresponding Fahrenheit temperature using the following formula:

```
fahrenheit = 9 / 5 * celsius + 32
```

In this formula, `celsius` is the *independent variable*, and `fahrenheit` is the *dependent variable*. Each `fahrenheit` temperature *depends on* the `celsius` temperature used in the calculation. If we were to plot the Fahrenheit temperature for each Celsius temperature, all of the points would appear along the same straight line, revealing a *linear relationship* between the two temperature scales.

Components of the Simple-Linear-Regression Equation

The points along any straight line (in two dimensions) like the regression lines shown in the preceding diagrams can be calculated with the equation:

$$y = mx + b$$

where

- m is the line's **slope**,
- b is the line's **intercept** with the y-axis (at $x = 0$), or simply the ***y*-intercept**,
- x is the independent variable, and
- y is the dependent variable.

In the formula for converting Celsius temperatures to Fahrenheit temperatures:

- m is `9 / 5`,
- b is `32`,
- x is `celsius`—the independent Celsius temperature, and
- y is `fahrenheit`—dependent Fahrenheit temperature produced by the calculation.

In simple linear regression, y is the *predicted value* for a given x. Of course, a line has an infinite number of points. If you can determine with simple linear regression the

equation for a straight line from a modest finite number of sample points, you then have the means to make an infinite number of predictions, even for independent variable values you've never seen before.

How Simple Linear Regression Works

Simple linear regression is a **machine-learning** technique that determines the slope (m) and y intercept (b) of a straight line that "best fits" your data. The simple linear regression algorithm iteratively adjusts the slope and intercept and, for each adjustment, calculates the square of each point's distance from the line. The "best fit" occurs when the slope and intercept values minimize the sum of those squared distances. This is known as an **ordinary least squares** calculation.[8]

Performing Simple Linear Regression with the GNU Scientific Library

The **GNU Scientific Library's** **`gsl_fit_linear` function** encapsulates simple linear regression's calculations, giving you as results the slope and y-intercept for the straight line that best fits the data. After calling `gsl_fit_linear`, you can plug into the $y = mx + b$ equation the slope (m) and intercept (b), then predict dependent y values, based on independent x values. We also use these values with **gnuplot** to display the regression line for the data along with the data points.

Comma-Separated Values (CSV) Files

We provided the Anscombe's Quartet data for you in the file `anscombe.csv`. This file and this case-study exercise's source code are located in the `AnscombesQuartet` subfolder of this chapter's examples folder. The **`.csv` filename extension** indicates that the file is in **CSV (comma-separated values)** format—a particularly popular file format for distributing datasets. CSV files are simply text files in which each line is one record of information with its items separated by commas. The following are the first two rows of `anscombe.csv`:

```
x1,y1,x2,y2,x3,y3,x4,y4
10,8.04,10,9.14,10,7.46,8,6.58
```

A CSV file's first row typically contains column names for the data in subsequent rows. In `anscombe.csv`, the remaining rows are the numeric values for the columns. In our code, the function `readAnscombesQuartetData` loads the data into arrays.

Installing the GNU Scientific Library on macOS

On macOS, you can install the GNU Scientific Library using the **Homebrew package manager**[9] as follows:

```
brew install gsl
```

Installing the GNU Scientific Library on Windows

In Visual Studio, you add the GNU Scientific Library to each project in which you wish to use it. With your project open in Visual Studio, perform the following steps:

---

8. `https://en.wikipedia.org/wiki/Ordinary_least_squares`.
9. If the `brew` command is not found, visit `https://brew.sh/` for install instructions.

1. Select **Tools > NuGet Package Manager > Manage NuGet Packages for Solution...**.
2. In the **Browse** tab, search for `"gsl-msvc-"`, then select `gsl-msvc-x64`.
3. In the right side of the NuGet package manager, click the checkbox next to your project's name, then click **Install** to add the library to your project.

Installing gnuplot on macOS

Install the gnuplot using the **Homebrew package manager** as follows:

```
brew install gnuplot
```

Installing gnuplot on Windows

Download and run the gnuplot Windows installer (`gp541-win64-mingw.exe`) from:

```
https://sourceforge.net/projects/gnuplot/files/gnuplot/5.4.1/
```

Click **Next >** until you reach the **Select Additional Tasks** step, then:

- Under **Select gnuplot's default terminal**, select the **windows** radio button.
- Scroll to the bottom of the settings and check the **Add application directory to your PATH environment variable** checkbox.

Click **Next >**, then **Install**. Once the installation completes, reboot your computer.

Compiling and Running the Program on macOS

On macOS, compile `anscome_macos.c` by performing the following steps:

1. Open a Terminal window.
2. Use the `cd` command to change to `AnscombesQuartet` subfolder of this chapter's examples folder.
3. Compile the program with the following command:

   ```
   clang -std=c18 anscombe_macos.c -lgsl -o anscombe_macos
   ```

4. Run the program:

   ```
   ./anscombe_macos
   ```

The program will create four PNG image files in the same folder as `anscome_macos.c` on macOS. You can open these image files to view the four plots.

Compiling and Running the Program on Windows

In the Visual Studio solution where you added the GNU Scientific library:

- Add to your project the file `anscome_windows.c` from the `AnscombesQuartet` subfolder of this chapter's examples folder.
- Modify line 72 to specify the location of `anscombe.csv` on your system
- Build and run your project.

When you run the program it will create four PNG image files in your project's folder. Use **File Explorer** to navigate to that folder, then open the image files to view the four plots.

Extensively Commented Code

Next, study the code to learn how to use the `gsl_fit_linear` function of the GNU Scientific Library to perform simple linear regression, and how to send gnuplot commands from a C program to the gnuplot application. Consider tweaking the gnuplot commands to see how your changes affect the plots our program produces. For example, you can change the plot's `pointtype`, `linewidth` and `linecolor` values.

AI/Data-Science Case Study: Time Series and Simple Linear Regression

Now that you've carefully studied the code for Anscombe's Quartet, you can adapt the program to other simple-linear-regression problems. Simple linear regression is commonly used to analyze **time series**—sequences of values (called **observations**) associated with points in time. Some examples are daily closing stock prices, hourly temperature readings, the changing positions of a plane in flight, annual crop yields and quarterly company profits. Perhaps the ultimate time series is the stream of time-stamped tweets coming from Twitter users worldwide.

Time Series

For this exercise, you'll use simple linear regression to analyze a time series containing New York City's average January temperatures *ordered* by year for the years 1895–2020. This is a **univariate time series**—it contains *one* observation per time. A **multivariate time series** has *two or more* observations per time, such as hourly temperature, humidity and barometric-pressure readings in a weather application. Your goal in this exercise is to determine whether the regression line has:

- a **negative slope**, indicating a **declining average temperature trend** over that time,
- a **zero slope**, indicating a **stable average temperature trend** over that time, or
- a **positive slope**, indicating an **increasing average temperature trend** over that time.

Getting Weather Data from NOAA

The National Oceanic and Atmospheric Administration (NOAA)

```
http://www.noaa.gov
```

provides extensive public historical weather data, including time series for average temperatures in specific cities over various time intervals.

We obtained the New York City January average temperatures for 1895–2020 (the maximum date range available at the time of this writing) from NOAA's "Climate at a Glance" time series at:

```
https://www.ncdc.noaa.gov/cag/city/time-series
```

You can select weather data for the entire U.S., regions within the U.S., states, cities and more. After selecting the data you need and the time frame to analyze, click **Plot** to display a diagram and view a table of the selected data. At the top of that table are icons you can click to download the data in several formats, including CSV.

For your convenience, we provided the file `nyc_ave_january_temps.csv` containing the data you'll use in this exercise. The file is located in the `nycdata` subfolder of this chapter's examples folder. We also "cleaned" the data, so the file contains the following two columns per observation:

- `Date`—A value of the form `YYYY` (such as `2020`). The downloaded data is in the form `YYYYMM` (such as `202001`), where `01` represents January. We removed `01` from each data item in this column, leaving only the year.
- `Temperature`—A floating-point Fahrenheit temperature. We renamed this column from `Value` in the downloaded data.

We deleted a third column called `Anomaly` that's not required for this exercise.

Performing the Regression

Modify the Anscombe's Quartet code to perform simple linear regression using the New York City average January temperatures data and to plot the data with a regression line. What trend do you see over the last 126 years?

Web Services and the Cloud Case Study—libcurl and OpenWeatherMap

11.21 ***(Getting a City's Weather Report with OpenWeatherMap)*** This is another of our challenge case-study exercises. Section 1.11 introduced the **Internet**, the **web**, **the cloud**, **web services** and **mashups**. In this case-study exercise, you'll dive into the world of web services using the open-source **libcurl**[10] and **cJSON**[11] libraries to invoke a web service and process the results it returns. You'll study a **fully coded, heavily commented program** that interacts with an **OpenWeatherMap free web service** from

```
https://openweathermap.org/
```

to get the current weather report for a city of your choosing.

We'll then challenge you to create your first mashup using what you've learned from this **fully coded example**. If you're entrepreneurial, you can quickly prototype powerful new applications by weaving existing web services into mashups. Even if you're not going to do the related challenge project, just mastering this case study's code will open up the vast world of web services to you.

Web Services

The machine on which a web service resides is referred to as a **web-service host**. A **client application** (in our case, a C program) sends a **request** over a network to the web-service host, which processes the request and returns a **response** over the network to the client. This kind of distributed computing benefits systems in various ways. For example, an application without direct access to data on another system might be

10. "libcurl — the multiprotocol file transfer library." Accessed January 4, 2021. `https://curl.se/libcurl/`.
11. "cJSON." Accessed January 4, 2021. `https://github.com/DaveGamble/cJSON`.

able to retrieve the data via a web service. Similarly, an application lacking the processing power to perform specific computations could use a web service to take advantage of another system's superior resources.

Representational State Transfer (REST)

Most web services today use an architectural style known as **Representational State Transfer (REST)** and are often called **RESTful web services**. In a RESTful web service, each function you can call is identified by a unique URL. **URLs (Uniform Resource Locators)** identify the locations on the Internet of resources, such as web sites and web services. When a web server receives a request to a RESTful web service, it immediately knows what function to call on that server. RESTful web services can be called from programs, as you'll do here, or directly from a web browser's address bar by entering the appropriate URL.

OpenWeatherMap

OpenWeatherMap provides a free tier to many of its weather web services. Before you can use them, you must sign up for a free account at `https://openweathermap.org/`. They will send you an email to verify your account. Once you do, they'll reply with an email that contains your free API key. You also can find this under the **API Keys** tab in your account when you log into the site.

You can view the variety of OpenWeatherMap APIs and their documentation at

```
https://openweathermap.org/api
```

Some are free and some are available only to subscribers. With a free API key, you can access:

- the **Current Weather Data** for a specified location, which we use in this case study,
- the **One Call API**, which returns a combination of current and future weather data for a specified location, and
- the **5 Day / 3 Hour Forecast** for a specified location.

JavaScript Object Notation (JSON)

Many cloud-based services like OpenWeatherMap communicate with your applications via JSON objects. **JSON (JavaScript Object Notation)** is a text-based, human-and-computer-readable, data-interchange format used to represent data as collections of name/value pairs. JSON has become the preferred data format for transmitting data over the Internet between applications. This is especially true for invoking cloud-based web services.

Each JSON object contains a comma-separated list of *property names* and *values* in curly braces. For example, the following key–value pairs might represent a client record:

```
{"account": 100, "name": "Jones", "balance": 24.98}
```

JSON also supports arrays, which are comma-separated values in square brackets. For example, the following is an acceptable JSON array of numbers:

```
[100, 200, 300]
```

Values in JSON objects and arrays can be:

- strings in *double quotes* (like `"Jones"`),
- numbers (like `100` or `24.98`),
- JSON Boolean values (represented as `true` or `false`),
- `null` (to represent no value, like `NULL` in C),
- arrays (like `[100, 200, 300]`), and
- other JSON objects.

Figure 11.8 contains a sample JSON response from OpenWeatherMap's Current Weather Data web service, which we formatted for readability. Even though you may never have seen JSON-encoded data, you should find this to be well organized, readable and pretty understandable.

```
{
  "coord": {
    "lon": -71.06,
    "lat": 42.36
  },
  "weather": [
    {
      "id": 803,
      "main": "Clouds",
      "description": "broken clouds",
      "icon": "04n"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 0.03,
    "feels_like": -4.96,
    "temp_min": -1.11,
    "temp_max": 1.11,
    "pressure": 1014,
    "humidity": 93
  },
  "visibility": 10000,
  "wind": {
    "speed": 4.1,
    "deg": 360
  },
  "clouds": {
    "all": 75
  },
  "dt": 1609815037,
```

Fig. 11.8 | Sample OpenWeatherMap response for Boston, MA, USA. (Part 1 of 2.)

```
32    "sys": {
33      "type": 1,
34      "id": 3486,
35      "country": "US",
36      "sunrise": 1609762409,
37      "sunset": 1609795488
38    },
39    "timezone": -18000,
40    "id": 4930956,
41    "name": "Boston",
42    "cod": 200
43  }
```

Fig. 11.8 | Sample OpenWeatherMap response for Boston, MA, USA. (Part 2 of 2.)

Open-Source libcurl Library

To obtain the JSON response in Fig. 11.8, our application uses functions from the open-source **libcurl library**:

```
https://curl.se/libcurl/
```

The library supports many Internet and web protocols for transmitting data between applications and can be used to invoke web services and receive their responses. You can find the documentation for libcurl's C functions at:

```
https://curl.se/libcurl/c/
```

In our fully coded example, `weather.c`, located in the weather folder of this chapter's examples folder, we extensively comment the libcurl functions you need to invoke a web service and save its response to a file.

To install libcurl on macOS or Linux:

- For macOS, you can install the libcurl library using the **Homebrew package manager**[12] as follows:

  ```
  brew install libcurl4
  ```

- For Ubuntu Linux, execute the command

  ```
  sudo apt install libcurl4-openssl-dev
  ```

In Visual Studio, you add libcurl to each project in which you wish to use it. With your project open in Visual Studio, perform the following steps:

1. Select **Tools > NuGet Package Manager > Manage NuGet Packages for Solution...**.
2. In the **Browse** tab, search for `"curl"`, then select "curl by curl contributors."
3. In the right side of the **NuGet Package Manager**, click the checkbox next to your project's name, then click **Install** to add the library to your project.

Open Source cJSON Library

The libcurl part of our application writes the OpenWeatherMap JSON response to a file. To display the weather report, our app reads the file's contents into a string,

12. If the `brew` command is not found, visit `https://brew.sh/` for install instructions.

then uses the open-source **cJSON library** to extract items from the JSON. You can download cJSON from:

```
https://github.com/DaveGamble/cJSON
```

There is no installation procedure for this library. You simply include the library's `cJSON.h` and `cJSON.c` files in your project.

cJSON's functions enable you to access items in the JSON response so we can display a weather report like the following:

```
Boston Weather
Temperature: 0.0 C
 Feels like: -5.0 C
   Pressure: 1014 hPa
   Humidity: 93%
 Conditions: broken clouds
```

In `weather.c`, we extensively commented the cJSON functions you need to extract the data above for the city you specify when you run the application (discussed below).

Compiling and Running the Program on macOS and Linux

On macOS and Linux, compile the weather app by performing the following steps:

1. Open a Terminal window.
2. Use the `cd` command to change to `weather` subfolder of this chapter's examples folder.
3. Compile the program with one of the following commands—`clang` on macOS or `gcc` on Linux:

```
clang -std=c18 -Wall weather.c cJSON.c -lcurl -o weather
gcc -std=c18 -Wall weather.c cJSON.c -lcurl -o weather
```

This application receives two command-line arguments. Though we do not discuss the details of command-line arguments until Section 15.3, this completely coded simulation provides the statements you need to receive the command-line arguments. The first is the city for which you'd like to get the current weather, such as

```
Boston,MA,USA
```

If the city's name contains a space, enclose the location in quotes:

```
"Los Angeles,CA,USA"
```

The second command-line argument is your OpenWeatherMap API key. The following command would get the current weather data for Boston, MA, USA:

```
./weather Boston,MA,USA API_KEY
```

Be sure to replace *API_KEY* with the OpenWeatherMap API key you received when you signed up for your free account.

Compile the program and run it several times. Next, study this application's code (including the extensive comments).

Compiling and Running the Weather App in Visual Studio

In the Visual Studio solution where you added the libcurl library, add to your project the files `weather.c` and `cJSON.c` from the `weather` subfolder of this chapter's examples folder. Specify the command-line arguments as follows:

- Right-click the project name in the Solution Explorer and select **Properties**.
- Expand **Configuration Properties** and select **Debugging**.
- Enter the arguments in the textbox to the right of **Command Arguments**.
- Build and run your project.

Challenge: Create Your Own Mashup

We introduced mashups in Section 1.11.3. After studying our libcurl-and-web-services-based weather application's code, including the extensive comments, as a challenge exercise attempt your first mashup. Web mashups typically combine capabilities from two or more complementary web services. For many popular mashups one of those is a mapping service, such as Google Maps or Microsoft's Bing Maps, but there are many mashup possibilities that do not use mapping.

To build a web-services mashup, you typically need:

- two or more complementary web services, which when you mash them up will help you produce a valuable new application.
- to be able to send a request from your C program to a web service, as you learned how to do with libcurl in this case study.
- to be able to receive results back from that web service in a form (typically JSON) that your C program can understand.

The web-services directory ProgrammableWeb

```
https://programmableweb.com/
```

lists nearly **24,000 web services** and **8,000 mashups**. They also provide "how-to" guides and sample code for working with web services and creating your own mashups. According to their website, some of the most widely used web services are Google Maps and others from Facebook, Twitter and YouTube.

Familiarize yourself with ProgrammableWeb. Look at lots of the web services they describe, focusing on free ones—it's common for web-service providers to offer some free services and some paid. Read the ProgrammableWeb "how-to" guides on creating your own mashups. For inspiration, glance through some of the 8,000 mashups they list on their site. Try to find two complementary free web services from which you can create a valuable mashup, then build that mashup.