Chapter 12

Data Files

Many applications require that information be written to or read from an auxiliary memory device. Such information is stored on the memory device in the form of a *data file*. Thus, data files allow us to store information permanently, and to access and alter that information whenever necessary.

In C, an extensive set of library functions is available for creating and processing data files. Unlike other programming languages, C does not distinguish between sequential and direct access (random access) data files. However, there are two different types of data files, called *stream-oriented* (or *standard*) data files, and *system-oriented* (or *low-level*) data files. Stream-oriented data files are generally easier to work with and are therefore more commonly used.

Stream-oriented data files can be subdivided into two categories. In the first category are *text* files, consisting of consecutive characters. These characters can be interpreted as individual data items, or as components of strings or numbers. The manner in which these characters are interpreted is determined either by the particular library functions used to transfer the information, or by format specifications within the library functions, as in the `scanf` and `printf` functions discussed in Chap. 4.

The second category of stream-oriented data files, often referred to as *unformatted* data files, organizes data into blocks containing contiguous bytes of information. These blocks represent more complex data structures, such as arrays and structures. A separate set of library functions is available for processing stream-oriented data files of this type. These library functions provide single instructions that can transfer entire arrays or structures to or from data files.

System-oriented data files are more closely related to the computer's operating system than stream-oriented data files. They are somewhat more complicated to work with, though their use may be more efficient for certain kinds of applications. A separate set of procedures, with accompanying library functions, is required to process system-oriented data files.

This chapter is concerned only with stream-oriented data files. The overall approach is relatively standardized, though the details may vary from one version of C to another. Thus, the examples presented in this chapter may not apply to all versions of the language in exactly the manner shown. Nevertheless, readers should have little difficulty in relating this material to their particular version of C.

12.1 OPENING AND CLOSING A DATA FILE

When working with a stream-oriented data file, the first step is to establish a *buffer area*, where information is temporarily stored while being transferred between the computer's memory and the data file. This buffer area allows information to be read from or written to the data file more rapidly than would otherwise be possible. The buffer area is established by writing

```
FILE *ptvar;
```

where `FILE` (uppercase letters required) is a special structure type that establishes the buffer area, and *ptvar* is a pointer variable that indicates the beginning of the buffer area. The structure type `FILE` is defined within a system `include` file, typically `stdio.h`. The pointer *ptvar* is often referred to as a *stream pointer*, or simply a *stream*.

A data file must be *opened* before it can be created or processed. This associates the file name with the buffer area (i.e., with the stream). It also specifies how the data file will be utilized, i.e., as a read-only file, a write-only file, or a read/write file, in which both operations are permitted.

The library function `fopen` is used to open a file. This function is typically written as

```
ptvar = fopen(file-name, file-type);
```

where *file-name* and *file-type* are strings that represent the name of the data file and the manner in which the data file will be utilized. The name chosen for the *file-name* must be consistent with the rules for naming files, as determined by the computer's operating system. The *file-type* must be one of the strings shown in Table 12-1.

Table 12-1 File-Type Specifications

File-Type	*Meaning*
`"r"`	Open an existing file for reading only.
`"w"`	Open a new file for writing only. If a file with the specified *file-name* currently exists, it will be destroyed and a new file created in its place.
`"a"`	Open an existing file for appending (i.e., for adding new information at the end of the file). A new file will be created if the file with the specified *file-name* does not exist.
`"r+"`	Open an existing file for both reading and writing.
`"w+"`	Open a new file for both reading and writing. If a file with the specified *file-name* currently exists, it will be destroyed and a new file created in its place.
`"a+"`	Open an existing file for both reading and appending. A new file will be created if the file with the specified *file-name* does not exist.

The `fopen` function returns a pointer to the beginning of the buffer area associated with the file. A `NULL` value is returned if the file cannot be opened as, for example, when an existing data file cannot be found.

Finally, a data file must be *closed* at the end of the program. This can be accomplished with the library function `fclose`. The syntax is simply

```
fclose(ptvar);
```

It is good programming practice to close a data file explicitly using the `fclose` function, though most C compilers will automatically close a data file at the end of program execution if a call to `fclose` is not present.

EXAMPLE 12.1 A C program contains the following statements.

```
#include <stdio.h>
FILE *fpt;
fpt = fopen("sample.dat", "w");
. . . . .
fclose(fpt);
```

The first statement causes the header file `stdio.h` to be included in the program. The second statement defines a pointer called `fpt` which will point to a structure of type `FILE`, indicating the beginning of the data-file buffer area. Note that `FILE` is defined in `stdio.h`.

The third statement opens a new data file called `sample.dat` as a write-only file. Moreover, the `fopen` function returns a pointer to the beginning of the buffer area and assigns it to the pointer variable `fpt`. Thus, `fpt` points to the buffer area associated with the data file `sample.dat`. All subsequent file processing statements (which are not shown explicitly in this example) will access the data file via the pointer variable `fpt` rather than by the file name.

Finally, the last statement closes the data file. Note that the argument is the pointer variable `fpt`, not the file name `sample.dat`.

The value returned by the `fopen` function can be used to generate an error message if a data file cannot be opened, as illustrated in the next example.

EXAMPLE 12.2 A C program contains the following statements.

```
#include <stdio.h>
#define  NULL  0

main()
{
    FILE *fpt;

    fpt = fopen("sample.dat", "r+");

    if (fpt == NULL)
       printf("\nERROR - Cannot open the designated file\n");
    else  {

       . . . . .

       fclose (fpt);
    }
}
```

This program attempts to open an existing data file called `sample.dat` for both reading and writing. An error message will be generated if this data file cannot be found. Otherwise the data file will be opened and processed, as indicated.

The `fopen` and the `if` statments are often combined, as follows.

```
if ((fpt = fopen("sample.dat", "r+")) == NULL)
   printf("\nERROR - Cannot open the designated file\n");
```

Either method is acceptable.

12.2 CREATING A DATA FILE

A data file must be created before it can be processed. A *stream-oriented* data file can be created in two ways. One is to create the file directly, using a text editor or a word processor. The other is to write a program that enters information into the computer and then writes it out to the data file. *Unformatted* data files can only be created with such specially written programs.

When creating a new data file with a specially written program, the usual approach is to enter the information from the keyboard and then write it out to the data file. If the data file consists of individual characters, the library functions `getchar` and `putc` can be used to enter the data from the keyboard and to write it out to the data file. We have already discussed the use of `getchar` in Sec. 4.2. The `putc` function is new, though its use is analogous to `putchar`, which we discussed in Sec. 4.3.

EXAMPLE 12.3 Creating a Data File (Lowercase to Uppercase Text Conversion) Here is a variation of several earlier programs, which read a line of lowercase text into the computer and write it out in uppercase (see Examples 4.4, 6.9, 6.12, 6.16 and 9.2). In this example we will read the text into the computer on a character-by-character basis using the `getchar` function, and then write it out to a data file using `putc`. The lowercase to uppercase conversion will be carried out by the library function `toupper`, as before.

The program begins by defining the stream pointer `fpt`, indicating the beginning of the data-file buffer area. A new data file, called `sample.dat`, is then opened for writing only. Next, a `do - while` loop reads a series of characters from the keyboard and writes their uppercase equivalents to the data file. The `putc` function is used to write each character to the data file. Notice that `putc` requires specification of the stream pointer `fpt` as an argument.

The loop continues as long as a newline character ('\n') is not entered from the keyboard. Once a newline character is detected, the loop ceases and the data file is closed.

```
/* read a line of lowercase text and store in uppercase within a data file */

#include <stdio.h>
#include <ctype.h>

main()
{
    FILE *fpt;    /* define a pointer to predefined structure type FILE */

    char c;

    /* open a new data file for writing only */
    fpt = fopen("sample.dat", "w");

    /* read each character and write its uppercase equivalent to the data file */
    do
        putc(toupper(c = getchar()), fpt);
    while (c != '\n');

    /* close the data file */
    fclose(fpt);
}
```

After the program has been executed, the data file `sample.dat` will contain an uppercase equivalent of the line of text entered into the computer from the keyboard. For example, if the original line of text had been

```
We, the people of the United States
```

the data file would contain the text

```
WE, THE PEOPLE OF THE UNITED STATES
```

A data file that has been created in this manner can be viewed in several different ways. For example, the data file can be viewed directly, using an operating system command such as `print` or `type`. The data file can also be examined using a text editor or a word processor.

Another approach is to write a program that will read the data file and display its contents. Such a program will, in a sense, be a mirror image of the one described above; i.e., the library function `getc` will read the individual characters from the data file, and `putchar` will display them on the screen. This is a more complicated way to display a data file but it offers a great deal of flexibility, since the individual data items can be processed as they are read.

EXAMPLE 12.4 Reading a Data File The following program will read a line of text from a data file on a character-by-character basis, and display the text on the screen. The program makes use of the library functions `getc` and `putchar` (see Sec. 4.3) to read and display the data. It complements the program presented in Example 12.3.

The logic is directly analogous to that of the program shown in Example 12.3. However, this program opens the data file `sample.dat` as a read-only file, whereas the previous program opened `sample.dat` as a write-only file. An error message is generated if `sample.dat` cannot be opened. Also, notice that `getc` requires the stream pointer `fpt` to be specified as an argument.

```
/* read a line of text from a data file and display it on the screen */

#include <stdio.h>

#define  NULL  0
```

```
main()

{
    FILE *fpt;    /* define a pointer to predefined structure type FILE */

    char c;

    if ((fpt = fopen("sample.dat", "r")) == NULL)
       /* open the data file for reading only */
       printf("\nERROR - Cannot open the designated file\n");

    else    /* read and display each character from the data file */
       do
            putchar(c = getc(fpt));
       while (c != '\n');

    /* close the data file */
    fclose(fpt);
}
```

Data files consisting entirely of strings can often be created and read more easily with programs that utilize special string-oriented library functions. Some commonly used functions of this type are `gets`, `puts`, `fgets` and `fputs`. The functions `gets` and `puts` read or write strings to or from the standard output devices, whereas `fgets` and `fputs` exchange strings with data files. Since the use of these functions is straightforward, we will not pursue this topic further. You may wish to experiment with these functions, however, by reprogramming some of the character-oriented read/write programs presented earlier.

Many data files consist of complex data structures, such as structures that contain various combinations of numeric and character information. Such data files can be processed using the library functions `fscanf` and `fprintf`, which are analogous to the functions `scanf` and `printf` discussed in Chap. 4 (see Secs. 4.4 and 4.6). Thus, the `fscanf` function permits formatted data to be read from a data file associated with a particular stream, and `fprintf` permits formatted data to be written to the data file. The actual format specifications are the same as those used with the `scanf` and `printf` functions.

EXAMPLE 12.5 Creating a File Containing Customer Records The last chapter presented three programs that supposedly were used to create and update customer records (see Examples 11.14 and 11.28). When describing the programs we remarked that the examples were unrealistic, because data files should be used for applications of this type. We now turn our attention to a program that creates such a data file for a series of customer records whose composition is as follows.

```
typedef struct  {
    int month;
    int day;
    int year;
}  date;

typedef struct  {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;
    char acct_type;
    float oldbalance;
    float newbalance;
    float payment;
    struct date lastpayment;
}  record;
```

The overall strategy will be to provide the current date, and then enter a loop that will process a series of customer records. For each customer, the customer's name, street, city, account number (acct_no) and initial balance (oldbalance) will be read into the computer. An initial value of 0 will then be assigned to the structure members newbalance and payment, the character 'C' will be assigned to acct_type (indicating a current status), and the current date assigned to lastpayment. Each customer record will then be written to a write-only data file called records.dat.

The procedure will continue until a customer name is encountered whose first three characters are END (in either upper- or lowercase). When END is encountered, it will be written to the data file, indicating an end-of-file condition.

Here is the complete C program.

```
/* create a data file containing customer records */

#include <stdio.h>
#include <string.h>

#define  TRUE  1

typedef struct  {
    int month;
    int day;
    int year;
}  date;

typedef struct  {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;                    /* (positive integer) */
    char acct_type;                 /* C (current), O (overdue), or D (delinquent) */
    float oldbalance;               /* (nonnegative quantity) */
    float newbalance;               /* (nonnegative quantity) */
    float payment;                  /* (nonnegative quantity) */
    date lastpayment;
}  record;

record readscreen(record customer);     /* function prototype */
void writefile(record customer);        /* function prototype */

FILE *fpt;                              /* pointer to predefined structure FILE */

main()
{
    int flag = TRUE;                    /* variable declaration */
    record customer;                    /* structure variable declaration */

    /* open a new data file for writing only */
    fpt = fopen("records.dat", "w");

    /* enter date and assign initial values */
    printf("CUSTOMER BILLING SYSTEM - INITIALIZATION\n\n");
    printf("Please enter today\'s date (mm/dd/yyyy): ");
    scanf("%d/%d/%d", &customer.lastpayment.month,
                      &customer.lastpayment.day,
                      &customer.lastpayment.year);
    customer.newbalance = 0;
    customer.payment = 0;
    customer.acct_type = 'C';
```

```
    /* main loop */
    while (flag)  {
                        /* enter customer's name and write to data file */
                        printf("\nName (enter \'END\' when finished): ");
                        scanf(" %[^\n]", customer.name);
                        fprintf(fpt, "\n%s\n", customer.name);

                        /* test for stopping condition */
                        if (strcmp(customer.name, "END") == 0)
                           break;

                        customer = readscreen(customer);
                        writefile(customer);
    }

    fclose(fpt);
}

record readscreen(record customer)     /* read remaining data */

{
    printf("Street: ");
    scanf(" %[^\n]", customer.street);
    printf("City: ");
    scanf(" %[^\n]", customer.city);
    printf("Account number: ");
    scanf("%d", &customer.acct_no);
    printf("Current balance: ");
    scanf("%f", &customer.oldbalance);
    return(customer);
}

void writefile(record customer)     /* write remaining data to a data file */

{
    fprintf(fpt, "%s\n", customer.street);
    fprintf(fpt, "%s\n", customer.city);
    fprintf(fpt, "%d\n", customer.acct_no);
    fprintf(fpt, "%c\n", customer.acct_type);
    fprintf(fpt, "%.2f\n", customer.oldbalance);
    fprintf(fpt, "%.2f\n", customer.newbalance);
    fprintf(fpt, "%.2f\n", customer.payment);
    fprintf(fpt, "%d/%d/%d\n", customer.lastpayment.month,
                               customer.lastpayment.day,
                               customer.lastpayment.year);
    return;
}
```

The program begins by defining the composition of each customer record and the stream pointer `fpt`. Within `main`, a new data file, called `records.dat`, is then opened for writing only. Next, the program prompts for the current date, and initial values are assigned to the structure members `newbalance`, `payment` and `acct_type`.

The program then enters a `while` loop, which prompts for a customer name and writes the name to the data file. Next, the program tests to see if the name that has been entered is `END` (upper- or lowercase). If so, the program breaks out of the loop, the data file is closed, and the computation terminates. Otherwise, the remaining information for the current customer is entered via function `readscreen`, and then written to the data file via function `writefile`.

Within `main` and `readscreen` we see that the various data items are entered interactively, using the familiar formatted `printf` and `scanf` functions. On the other hand, within `main` and `writefile` the data are written to the data file via the `fprintf` function. The syntax governing the use of this function is the same as the syntax used with `printf`, except that a stream pointer must be included as an additional argument. Notice that the control string makes use of the same character groups (i.e., the same formatting features) as the `printf` function described in Chap. 4.

When the program is executed, the information for each customer record is entered interactively, as shown below for four fictitious customers. As usual, the user's responses are underlined.

```
CUSTOMER BILLING SYSTEM - INITIALIZATION

Please enter today's date (mm/dd/yyyy): 5/24/1998

Name (enter 'END' when finished): Steve Johnson
Street: 123 Mountainview Drive
City: Denver, CO
Account number: 4208
Current Balance: 247.88

Name (enter 'END' when finished): Susan Richards
Street: 4383 Alligator Blvd
City: Fort Lauderdale, FL
Account number: 2219
Current Balance: 135.00

Name (enter 'END' when finished): Martin Peterson
Street: 1787 Pacific Parkway
City: San Diego, CA
Account number: 8452
Current Balance: 387.42

Name (enter 'END' when finished): Phyllis Smith
Street: 1000 Great White Way
City: New York, NY
Account number: 711
Current Balance: 260.00

Name (enter 'END' when finished): END
```

After the program has been executed, the data file `records.dat` will have been created, containing the following information.

```
Steve Johnson
123 Mountainview Drive
Denver, CO
4208
C
247.88
0.00
0.00
5/24/1998
```

```
Susan Richards
4383 Alligator Blvd
Beechview, OH
2219
C
135.00
0.00
0.00
5/24/1998

Martin Peterson
1787 Pacific Parkway
San Diego, CA
8452
C
387.42
0.00
0.00
5/24/1998

Phyllis Smith
1000 Great White Way
New York, NY
711
C
260.00
0.00
0.00
5/24/1998

END
```

In the next section we will see a program that updates the information contained in this file.

12.3 PROCESSING A DATA FILE

Most data file applications require that a data file be altered as it is being processed. For example, in an application involving the processing of customer records, it may be desirable to add new records to the file (either at the end of the file or interspersed among the existing records), to delete existing records, to modify the contents of existing records, or to rearrange the records. These requirements in turn suggest several different computational strategies.

Consider, for example, the problem of updating the records within a data file. There are several approaches to this problem. Perhaps the most obvious approach is to read each record from a data file, update the record as required, and then write the updated record to the same data file. However, there are some problems with this strategy. In particular, it is difficult to read and write formatted data to the same data file without disrupting the arrangement of the data items within the file. Moreover, the original set of records may become inaccessible if something goes wrong during the program execution.

Another approach is to work with two different data files — an old file (a *source*) and a new file. Each record is read from the old file, updated as necessary, and then written to the new file. When all of the records have been updated, the old file is deleted or placed into archival storage and the new file renamed. Hence, the new file will become the source for the next round of updates.

Historically, the origin of this method goes back to the early days of computing, when data files were maintained on magnetic tapes. The method is still used, however, because it provides a series of old source

files that can be used to generate a customer history. The most recent source file can also be used to recreate the current file if the current file is damaged or destroyed.

EXAMPLE 12.6 Updating a File Containing Customer Records Example 12.5 presents a program to create a data file called `records.dat` that contains customer records. We now present a program to update the records within this data file. The program will make use of the two-file update procedure described above. Hence, we will assume that the previously created data file `records.dat` has been renamed `records.old`. This will be the source file.

Our overall strategy will be similar to that described in Example 12.5. That is, we will first provide the current date, and then enter a loop that will read a series of customer records from `records.old`, and write the corresponding updated records to a new data file called `records.new`. Each pass through the loop will read one record, update it if necessary, and then write the record to `records.new`. By following this procedure, all of the records will be written to `records.new`, whether updated or not.

The procedure will continue until the customer name `END` has been read from the source file (in either upper- or lowercase). Once this happens, `END` will be written to the new data file, indicating an end-of-file condition.

The complete program is given below. The program begins by defining the composition of each customer record, using the same definitions presented in Example 12.5. These definitions are followed by definitions of the stream pointers `ptold` and `ptnew`.

Within the `main` function, `records.old` is opened as an existing read-only file, and `records.new` is opened as a new write-only file. An error message is generated if `records.old` cannot be opened. Otherwise, the program enters a `while` loop that reads successive customer records from `records.old` (actually, from stream `ptold`), updates each record as required, and writes each record to `records.new` (to stream `ptnew`).

```
/* update a data file containing customer records */

#include <stdio.h>
#include <string.h>

#define  NULL  0
#define  TRUE  1

typedef struct  {
    int month;
    int day;
    int year;
}  date;

typedef struct  {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;                    /* (positive integer) */
    char acct_type;                 /* C (current), O (overdue), or D (delinquent) */
    float oldbalance;               /* (nonnegative quantity) */
    float newbalance;               /* (nonnegative quantity) */
    float payment;                  /* (nonnegative quantity) */
    date lastpayment;
}  record;

record readfile(record customer);   /* function prototype */
record update(record customer);     /* function prototype */
void writefile(record customer);    /* function prototype */

FILE *ptold, *ptnew;                /* pointers to predefined structure FILE */
int month, day, year;               /* global variable declarations */
```

```
main()

{
int flag = TRUE;                    /* local variable declaration */
    record customer;                /* structure variable declaration */

    /* open data files */
    if ((ptold = fopen("records.old", "r")) == NULL)
       printf("\nERROR - Cannot open the designated read file\n");
    else   {
       ptnew = fopen("records.new", "w");

       /* enter current date */
       printf("CUSTOMER BILLING SYSTEM - UPDATE\n\n");
       printf("Please enter today\'s date (mm/dd/yyyy): ");
       scanf("%d/%d/%d", &month, &day, &year);

       /* main loop */
       while (flag)  {
           /* read a name from old data file and write to new data file */
           fscanf(ptold, " %[^\n]", customer.name);
           fprintf(ptnew, "\n%s\n", customer.name);

           /* test for stopping condition */
           if (strcmp(customer.name, "END") == 0)
              break;

           /* read remaining data from old data file */
           customer = readfile(customer);

           /* prompt for updated information */
           customer = update(customer);

           /* write updated information to new data file */
           writefile(customer);
       }
       fclose(ptold);
       fclose(ptnew);
    }  /* end else */
}


record readfile(record customer)   /* read remaining data from the old data file */

{
    fscanf(ptold, " %[^\n]", customer.street);
    fscanf(ptold, " %[^\n]", customer.city);
    fscanf(ptold, " %d", &customer.acct_no);
    fscanf(ptold, " %c", &customer.acct_type);
    fscanf(ptold, " %f", &customer.oldbalance);
    fscanf(ptold, " %f", &customer.newbalance);
    fscanf(ptold, " %f", &customer.payment);
    fscanf(ptold, " %d/%d/%d", &customer.lastpayment.month,
                               &customer.lastpayment.day,
                               &customer.lastpayment.year);
    return(customer);
}
```

```
record update(record customer)   /* prompt for new information, update records and
                                                                display summary data */
{
    printf("\n\nName:   %s", customer.name);
    printf("     Account number: %d\n", customer.acct_no);
    printf("\nOld balance: %7.2f", customer.oldbalance);
    printf("    Current payment: ");
    scanf("%f", &customer.payment);

    if (customer.payment > 0)   {
       customer.lastpayment.month = month;
       customer.lastpayment.day = day;
       customer.lastpayment.year = year;
       customer.acct_type = (customer.payment < 0.1 * customer.oldbalance) ? 'O' : 'C';
    }
    else
       customer.acct_type = (customer.oldbalance > 0) ? 'D' : 'C';

    customer.newbalance = customer.oldbalance - customer.payment;
    printf("New balance: %7.2f", customer.newbalance);

    printf("    Account status: ");
    switch (customer.acct_type)   {
       case 'C':
                 printf("CURRENT\n");
                 break;
       case 'O':
                 printf("OVERDUE\n");
                 break;
       case 'D':
                 printf("DELINQUENT\n");
                 break;
       default:
                 printf("ERROR\n");
    }

    return(customer);
}

void writefile(record customer)   /* write updated information to the new data file */

{
    fprintf(ptnew, "%s\n", customer.street);
    fprintf(ptnew, "%s\n", customer.city);
    fprintf(ptnew, "%d\n", customer.acct_no);
    fprintf(ptnew, "%c\n", customer.acct_type);
    fprintf(ptnew, "%.2f\n", customer.oldbalance);
    fprintf(ptnew, "%.2f\n", customer.newbalance);
    fprintf(ptnew, "%.2f\n", customer.payment);
    fprintf(ptnew, "%d/%d/%d\n", customer.lastpayment.month,
                                 customer.lastpayment.day,
                                 customer.lastpayment.year);
    return;
}
```

Each customer name is read from the source file and then written to the new file within `main`. The remaining information for each record is then read from the source file, updated, and written to the new file within the functions `readfile`, `update`, and `writefile`, respectively. This process continues until a record is encountered containing the customer name `END`, as discussed above. Both data files are then closed, and the computation terminates.

The function `readfile` reads additional information for each customer record from the source file. The various data items are represented as members of the structure variable `customer`. This structure variable is passed to the function as an argument. The library function `fscanf` is used to read each data item, using a syntax that is essentially identical to that used with the `scanf` function, as described in Chap. 4. With `fscanf`, however, the stream pointer `ptold` must be included as an additional argument within each function call. Once all of the information has been read from the source file, the customer record is returned to `main`.

The function `update` is similar, though it requires that a value for `customer.payment` be entered from the keyboard. Additional information is then assigned to `customer.lastpayment`, `customer.acct_type` and `customer.newbalance`. The values assigned depend on the value provided for `customer.payment`. The updated record is then returned to `main`.

The remaining function, `writefile`, simply accepts each customer record as an argument and writes it to the new data file. Within `writefile`, the library function `fprintf` is used to transfer the information to the new data file, using the same procedures shown in Example 12.5.

When the program is executed, the name, account number and old balance are displayed for each customer. The user is then prompted for a value for the current payment. Once this value has been entered, the customer's new balance and current account status are shown.

A typical interactive session, based upon the data file created in Example 12.5, is shown below. The user's responses are underlined, as usual.

```
CUSTOMER BILLING SYSTEM - UPDATE

Please enter today's date (mm/dd/yyyy): 12/29/1998

Name:   Steve Johnson     Account number: 4208

Old balance:  247.88    Current payment: 25.00
New balance:  222.88    Account status: CURRENT


Name:   Susan Richards     Account number: 2219

Old balance:  135.00    Current payment: 135.00
New balance:    0.00    Account status: CURRENT


Name:   Martin Peterson     Account number: 8452

Old balance:  387.42    Current payment: 35.00
New balance:  352.42    Account status: OVERDUE


Name:   Phyllis Smith     Account number: 711

Old balance:  260.00    Current payment: 0
New balance:  260.00    Account status: DELINQUENT
```

After all of the customer records have been processed the new data file `records.new` will have been created, containing the following information.

```
Steve Johnson
123 Mountainview Drive
Denver, CO
4208
C
247.88
222.88
25.00
12/29/1998

Susan Richards
4383 Alligator Blvd
Fort Lauderdale, FL
2219
C
135.00
0.00
135.00
12/29/1998

Martin Peterson
1787 Pacific Parkway
San Diego, CA
8452
O
387.42
352.42
35.00
12/29/1998

Phyllis Smith
1000 Great White Way
New York, NY
711
D
260.00
260.00
0.00
5/24/1998

END
```

Note that the old data file, `records.old`, is still available in its original form; hence, it can be stored for archival purposes. Before this program can be run again, however, the new data file will have to be renamed `records.old`. (Usually, this is done at the operating system level, before entering the update program.)

12.4 UNFORMATTED DATA FILES

Some applications involve the use of data files to store blocks of data, where each block consists of a fixed number of contiguous bytes. Each block will generally represent a complex data structure, such as a structure or an array. For example, a data file may consist of multiple structures having the same composition, or it may contain multiple arrays of the same type and size. For such applications it may be desirable to read the

entire block from the data file, or write the entire block to the data file, rather than reading or writing the individual components (i.e., structure members or array elements) within each block separately.

The library functions `fread` and `fwrite` are intended to be used in situations of this type. These functions are often referred to as *unformatted* read and write functions. Similarly, data files of this type are often referred to as unformatted data files.

Each of these functions requires four arguments: a pointer to the data block, the size of the data block, the number of data blocks being transferred, and the stream pointer. Thus, a typical `fwrite` function might be written as

```
fwrite(&customer, sizeof(record), 1, fpt);
```

where `customer` is a structure variable of type `record`, and `fpt` is the stream pointer associated with a data file that has been opened for output.

EXAMPLE 12.7 Creating an Unformatted Data File Containing Customer Records Consider a variation of the program presented in Example 12.5, for creating a data file containing customer records. Now, however, we will write each customer record to the data file `data.bin` as a single, unformatted block of information. This is in contrast to the earlier program, where we wrote the items within each record (i.e., the individual structure members) as separate, formatted data items.

Here is the complete program.

```
/* create an unformatted data file containing customer records */

#include <stdio.h>
#include <string.h>

#define  TRUE  1

typedef struct  {
    int month;
    int day;
    int year;
}  date;

typedef struct  {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;                    /* (positive integer) */
    char acct_type;                 /* C (current), O (overdue), or D (delinquent) */
    float oldbalance;               /* (nonnegative quantity) */
    float newbalance;               /* (nonnegative quantity) */
    float payment;                  /* (nonnegative quantity) */
    date lastpayment;
}  record;

record readscreen(record customer);   /* function prototype */

FILE *fpt;                            /* pointer to predefined structure FILE */


main()

{
    int flag = TRUE;                  /* variable declaration */
    record customer;                  /* structure variable declaration */
```

```
    /* open a new data file for writing only */
    fpt = fopen("data.bin", "w");

    /* enter date and assign initial values */
    printf("CUSTOMER BILLING SYSTEM - INITIALIZATION\n\n");
    printf("Please enter today\'s date (mm/dd/yyyy): ");
    scanf("%d/%d/%d", &customer.lastpayment.month,
                      &customer.lastpayment.day,
                      &customer.lastpayment.year);
    customer.newbalance = 0;
    customer.payment = 0;
    customer.acct_type = 'C';

    /* main loop */
    while (flag)  {
        /* enter customer's name */
        printf("\nName (enter \'END\' when finished): ");
        scanf(" %[^\n]", customer.name);

        /* test for stopping condition */
        if (strcmp(customer.name, "END") == 0)
           break;

        /* enter remaining data and write to data file */
        customer = readscreen(customer);
        fwrite(&customer, sizeof(record), 1, fpt);

        /* erase strings */
        strset(customer.name, ' ');
        strset(customer.street, ' ');
        strset(customer.city, ' ');
    }

    fclose(fpt);
}


record readscreen(record customer)     /* read remaining data */

{
    printf("Street: ");
    scanf(" %[^\n]", customer.street);
    printf("City: ");
    scanf(" %[^\n]", customer.city);
    printf("Account number: ");
    scanf("%d", &customer.acct_no);
    printf("Current balance: ");
    scanf("%f", &customer.oldbalance);
    return(customer);
}
```

Comparing this program with that shown in Example 12.5, we see that the two programs are very similar. Within `main`, the present program reads each customer name and tests for a stopping condition (`END`), but does not write the customer name to the data file, as in the earlier program. Rather, if a stopping condition is not indicated, the present program reads the remainder of the customer record interactively, and then writes the entire customer record to the data file with the single `fwrite` statement

```
    fwrite(&customer, sizeof(record), 1, fpt);
```

Note that the data file created by this program is called `data.bin`, as indicated by the first argument within the call to the `fopen` function.

The programmer-defined `writefile` function shown in Example 12.5 is not required in this program, since the `fwrite` library function takes its place. On the other hand, both programs make use of the same programmer-defined function `readscreen`, which causes the information for a given customer record to be entered into the computer interactively.

After each record has been written to the data file, the string members `customer.name`, `customer.street` and `customer.city` are cleared (i.e., replaced with blanks), so that none of the previous information will be included in each new record. The library function `strset` is used for this purpose. Thus, the statement

```
strset(customer.name, ' ');
```

causes the contents of `customer.name` to be replaced with repeated blank characters, as indicated by `' '`. Note that the header file `string.h` is included in this program, in support of the `strset` function.

Execution of this program produces the same interactive dialog as that shown in Example 12.5. Thus, during program execution the user cannot tell whether the data file being created is formatted or unformatted. Once the new data file `data.bin` has been created, however, its contents will not be legible unless the file is read by a specially written program. Such a program will be presented in the next example.

Once an unformatted data file has been created, the question arises as to how to detect an end-of-file condition. The library function `feof` is available for this purpose. (Actually, `feof` will indicate an end-of-file condition for *any* stream-oriented data file, not just an unformatted data file.) This function returns a non-zero value (`TRUE`) if an end-of-file condition has been detected, and a value of zero (`FALSE`) if an end of file is *not* detected. Hence, a program that reads an unformatted data file can utilize a loop that continues to read successive records, as long as the value returned by `feof` is not `TRUE`.

EXAMPLE 12.8 Updating an Unformatted Data File Containing Customer Records We now consider a program for reading and updating the unformatted data file created in Example 12.7. We will again make use of a two-file update procedure, as in Example 12.6. Now, however, the files will be called `data.old` and `data.new`. Therefore, the file created in the previous example, called `data.bin`, will have to be renamed `data.old` before the present program can be run.

The overall program logic is similar to that presented in Example 12.6. That is, a record is read from `data.old`, updated interactively, and then written to `data.new`. This procedure continues until an end-of-file condition has been detected during the most recent `fread` operation. Note the manner in which the end-of-file test is built into the specification of the `while` loop, i.e., `while (!feof(ptold))`.

This program, however, will make use of the library functions `fread` and `fwrite` to read unformatted customer records from `data.old`, and to write the updated records to `data.new`. Therefore the present program will not make use of programmer-defined functions `readfile` and `writefile`, as in Example 12.6.

The updating of each record is carried out interactively, via the user-defined function `update`. This function is identical to that shown in Example 12.6.

The entire C program is shown below.

```
/* update an unformatted data file containing customer records */

#include <stdio.h>

#define  NULL  0

typedef struct  {
    int month;
    int day;
    int year;
}  date;
```

```
typedef struct  {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;                    /* (positive integer) */
    char acct_type;                 /* C (current), O (overdue), or D (delinquent) */
    float oldbalance;               /* (nonnegative quantity) */
    float newbalance;               /* (nonnegative quantity) */
    float payment;                  /* (nonnegative quantity) */
    date lastpayment;
}  record;

record update(record customer);     /* function prototype */

FILE *ptold, *ptnew;          ...../* pointers to pre-defined structure FILE */
int month, day, year;         ...../* global variable declarations */



main()

{
    record customer;                /* structure variable declaration */

    /* open data files */
    if ((ptold = fopen("data.old", "r")) == NULL)
       printf("\nERROR - Cannot open the designated read file\n");
    else   {
       ptnew = fopen("data.new", "w");

       /* enter current date */
       printf("CUSTOMER BILLING SYSTEM - UPDATE\n\n");
       printf("Please enter today\'s date (mm/dd/yyyy): ");
       scanf("%d/%d/%d", &month, &day, &year);

       /* read the first record from old data file */
       fread(&customer, sizeof(record), 1, ptold);

       /* main loop (continue until end-of-file is detected) */
       while (!feof(ptold))  {
           /* prompt for updated information */
           customer = update(customer);

           /* write updated information to new data file */
           fwrite(&customer, sizeof(record), 1, ptnew);

           /* read next record from old data file */
           fread(&customer, sizeof(record), 1, ptold);
       }

       fclose(ptold);
       fclose(ptnew);
    }  /* end else */
}
```

```
record update(record customer)   /* prompt for new information, update records and
                                                                display summary data */

{
    printf("\n\nName:   %s", customer.name);
    printf("     Account number: %d\n", customer.acct_no);
    printf("\nOld balance: %7.2f", customer.oldbalance);
    printf("    Current payment: ");
    scanf("%f", &customer.payment);

    if (customer.payment > 0)   {
       customer.lastpayment.month = month;
       customer.lastpayment.day = day;
       customer.lastpayment.year = year;
       customer.acct_type = (customer.payment < 0.1 * customer.oldbalance) ? 'O' : 'C';
    }
    else
       customer.acct_type = (customer.oldbalance > 0) ? 'D' : 'C';

    customer.newbalance = customer.oldbalance - customer.payment;
    printf("New balance: %7.2f", customer.newbalance);

    printf("    Account status: ");
    switch (customer.acct_type)   {
       case 'C':
                 printf("CURRENT\n");
                 break;
       case 'O':
                 printf("OVERDUE\n");
                 break;
       case 'D':
                 printf("DELINQUENT\n");
                 break;
       default:
                 printf("ERROR\n");
    }

    return(customer);
}
```

Execution of the program results in the same interactive dialog as that shown in Example 12.6.

We will not pursue the use of data files further within this book. Remember, however, that most versions of C contain many different library functions for carrying out various file-oriented operations. Some of these functions are intended to be used with standard input/output devices (i.e., reading from the keyboard and writing to the screen), some are intended for stream-oriented data files, and others are available for use with system-oriented data files. Thus, we have only scratched the surface of this important topic within the present chapter. You should find out what file-related functions are available for your particular version of the language.

Review Questions

12.1 What is the primary advantage to using a data file?

12.2 Describe the different ways in which data files can be categorized in C.

12.3 What is the purpose of a buffer area when working with a stream-oriented data file? How is a buffer area defined?