Chapter 11
C File Processing

C How to Program, 8/e, GE

11.1 Introduction

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

11.2 Files and Streams

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

11.2 Files and Streams (Cont.)

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

11.2 Files and Streams (Cont.)

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

11.2 Files and Streams (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

- Function **`fprintf`** is equivalent to **`printf`** *except* that **`fprintf`** also receives as an argument a *file pointer for the file* to which the data will be written.
- Function **`fprintf`** can output data to the standard output by using **`stdout`** as the *file pointer*, as in:

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

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

11.3 Creating a Sequential-Access File (Cont.)

- In addition, C11 provides *exclusive* write mode, which you indicate by adding an `x` to the end of the **`w`**, **`w+`**, **`wb`** or **`wb+`** modes.
- In exclusive write mode, **fopen** will fail if the file already exists or cannot be created.
- If opening a file in exclusive write mode is successful and the underlying system supports exclusive file access, then *only your program can access* the file while it's open.
- (Some compilers and platforms do not support exclusive write mode.)
- If an error occurs while opening a file in any mode, `fopen` returns `NULL`.

11.4 Reading Data from a Sequential-Access File

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

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

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

Resetting the File Position Pointer

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

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

- The statement
  - `rewind(cfPtr);`

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

Credit Inquiry Program

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

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

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

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

- Data in this type of sequential file cannot be modified without the risk of destroying other data.
- For example, if the name "**`White`**" needs to be changed to "**`Worthington`**," the old name cannot simply be overwritten.
- If the record is rewritten beginning at the same location in the file using the new name, the record will be
  - **`300 Worthington 0.00`**
- The new record is *larger* (has more characters) than the original record.
- The characters beyond the second "**`o`**" in "**`Worthington`**" will *overwrite* the beginning of the *next* sequential record in the file.
- The problem here is that in the **formatted input/output model** using **`fprintf`** and **`fscanf`**, fields—and hence records—can *vary in size*.

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

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

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

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

11.5 Random-Access Files

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

11.5 Random-Access Files (Cont.)

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

11.6 Creating a Random-Access File

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

11.6 Creating a Random-Access File (Cont.)

- Now, when writing an integer, instead of using

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

which could print a single digit or as many as 11 digits (10 digits plus a sign, each of which requires 1 byte of storage) for a four-byte integer, we can use

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

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

11.6 Creating a Random-Access File (Cont.)

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

11.6 Creating a Random-Access File (Cont.)

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

11.6 Creating a Random-Access File (Cont.)

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

11.7 Creating a Random-Access File (Cont.)

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

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

- *Lines 40–41* position the file position pointer for the file referenced by **`cfPtr`** to the byte location calculated by:

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

- The value of this expression is called the **offset** or the **displacement**.
- Because the account number is between 1 and 100 but the byte positions in the file *start with* **0**, **1** is subtracted from the account number when calculating the byte location of the record.

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

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

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

- The function prototype for **fseek** is:

```
int fseek(FILE *stream, long int offset, int whence);
```

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

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

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

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

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

11.8 Reading Data from a Random-Access File

- Function **`fread`** reads a specified number of bytes from a file into memory.
- For example,

```
fread(&client, sizeof(struct clientData), 1, cfPtr);
```

  reads the number of bytes determined by **`sizeof(struct clientData)`** from the file referenced by **`cfPtr`**, stores the data in **`client`** and returns the number of bytes read.

- The bytes are read from the location specified by the file position pointer.

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

- Function **`fread`** can read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read.
- Below statement reads ***one*** element:

```
fread(&client, sizeof(struct clientData), 1, cfPtr);
```

- To read *more than one*, specify the number of elements as **`fread`**'s third argument.
- Function **`fread`** returns the number of items it successfully input.

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

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

11.9 Case Study: Transaction-Processing Program

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

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

- The program has five options.
- Option 1 calls function **`textFile`** to store a formatted list of all the accounts (typically called a report) in a text file called **`accounts.txt`** that may be printed later.

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

- Option 2 calls the function **`updateRecord`** to update an account.
- The function will update only a record that already exists, so the function first checks whether the record specified by the user is empty.
- The record is read into structure **`client`** with **`fread`**, then member **`acctNum`** is compared to 0.

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

- If it's **0**, the record contains no information, and a message is printed stating that the record is empty.
- Then the menu choices are displayed.
- If the record contains information, function **`updateRecord`** inputs the transaction amount, calculates the new balance and rewrites the record to the file.

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

- Option 3 calls the function **`newRecord`** to add a new account to the file.
- If the user enters an account number for an existing account, **`newRecord`** displays an error message indicating that the record already contains information, and the menu choices are printed again.

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

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