17

File Processing

I read part of it all the way through.
—Samuel Goldwyn

A great memory does not make a philosopher,
any more than a dictionary can be called grammar.
—John Henry, Cardinal Newman

I can only assume that a "Do Not File" document is filed in a "Do Not File" file.
—Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975

Objectives

In this chapter you'll learn:

- To create, read, write and update files.
- Sequential file processing.
- Random-access file processing.
- To use high-performance unformatted I/O operations.
- The differences between formatted-data and raw-data file processing.
- To build a transaction-processing program using random-access file processing.

17.1 Introduction

Storage of data in memory is temporary. **Files** are used for **data persistence**—permanent retention of data. Computers store files on **secondary storage devices,** such as hard disks, CDs, DVDs, flash drives and tapes. In this chapter, we explain how to build C++ programs that create, update and process data files. We consider both sequential files and random-access files. We compare formatted-data file processing and raw-data file processing. We examine techniques for input of data from, and output of data to, `string` streams rather than files in Chapter 18, Class `string` and String Stream Processing.

17.2 Data Hierarchy

Ultimately, all data items that digital computers process are reduced to combinations of zeros and ones. This occurs because it's simple and economical to build electronic devices that can assume two stable states—one state represents `0` and the other represents `1`. It's remarkable that the impressive functions performed by computers ultimately involve only the most fundamental manipulations of `0`s and `1`s.

The smallest data item that computers support is called a **bit** (short for "**binary digit**"—a digit that can assume one of two values). Each data item, or bit, can assume either the value `0` or the value `1`. Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from `1` to `0` or from `0` to `1`).

Programming with data in the low-level form of bits is cumbersome. It's preferable to program with data in forms such as **decimal digits** (0–9), **letters** (A–Z and a–z) and **special symbols** (e.g., $, @, %, &, * and many others). Digits, letters and special symbols are referred to as **characters**. The set of all characters used to write programs and represent data items on a particular computer is called that computer's **character set**. Because computers can process only `1`s and `0`s, every character in a computer's character set is represented as a pattern of `1`s and `0`s. **Bytes** are composed of eight bits. You create programs and data items with characters; computers manipulate and process these characters as patterns of bits. For example, C++ provides data type `char`. Each `char` typically occupies one byte. C++ also provides data type `wchar_t`, which can occupy more than one byte (to support larger character sets, such as the **Unicode® character set**; for more information on Unicode®, visit `www.unicode.org`).

Just as characters are composed of bits, **fields** are composed of characters. A field is a group of characters that conveys some meaning. For example, a field consisting of uppercase and lowercase letters can represent a person's name.

Data items processed by computers form a **data hierarchy** (Fig. 17.1), in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and to larger data aggregates.

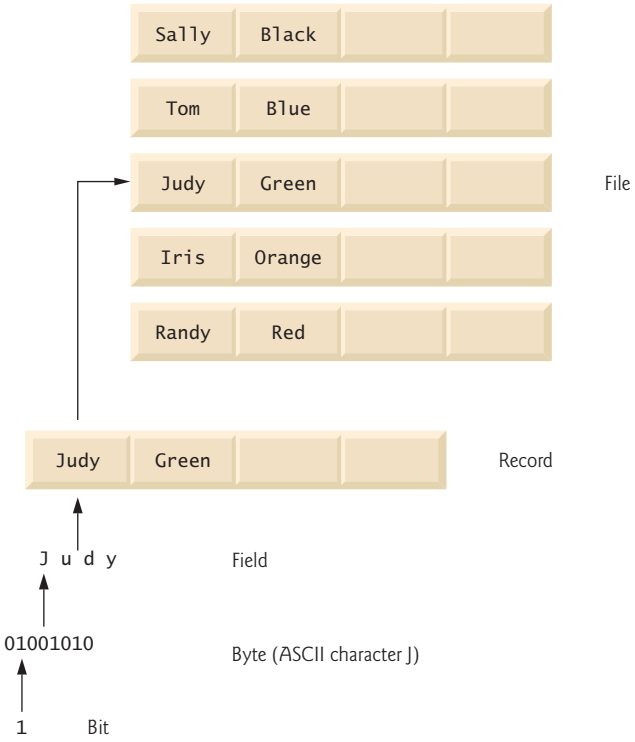

Fig. 17.1 | Data hierarchy.

Typically, a **record** (which can be represented as a `class` in C++) is composed of several fields (called data members in C++). In a payroll system, for example, a record for a particular employee might include the following fields:

1. Employee identification number
2. Name
3. Address
4. Hourly pay rate
5. Number of exemptions claimed
6. Year-to-date earnings
7. Amount of taxes withheld

Thus, a record is a group of related fields. In the preceding example, each field is associated with the same employee. A file is a group of related records.[1] A company's payroll file normally contains one record for each employee. Thus, a payroll file for a small company might contain only 22 records, whereas one for a large company might contain 100,000 records. It isn't unusual for a company to have many files, some containing millions, billions, trillions or more characters of information.

To facilitate retrieving specific records from a file, at least one field in each record is chosen as a **record key**. A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all others. In the payroll record described previously, the employee identification number normally would be chosen as the record key.

There are many ways of organizing records in a file. A common type of organization is called a **sequential file,** in which records typically are stored in order by a record-key field. In a payroll file, records usually are placed in order by employee identification number. The first employee record in the file contains the lowest employee identification number, and subsequent records contain increasingly higher ones.

Most businesses use many different files to store data. For example, a company might have payroll files, accounts-receivable files (listing money due from clients), accounts-payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types of files. A group of related files often are stored in a **database.** A collection of programs designed to create and manage databases is called a **database management system (DBMS)**.

17.3 Files and Streams

C++ views each file as a sequence of bytes (Fig. 17.2). Each file ends either with an **end-of-file marker** or at a specific byte number recorded in an operating-system-maintained, administrative data structure. When a file is *opened*, an object is created, and a stream is associated with the object. In Chapter 15, we saw that objects `cin`, `cout`, `cerr` and `clog` are created when `<iostream>` is included. The streams associated with these objects provide communication channels between a program and a particular file or device. For example, the `cin` object (standard input stream object) enables a program to input data from the keyboard or from other devices, the `cout` object (standard output stream object) enables a program to output data to the screen or other devices, and the `cerr` and `clog` objects (standard error stream objects) enable a program to output error messages to the screen or other devices.

Fig. 17.2 | C++'s view of a file of *n* bytes.

1. Generally, a file can contain arbitrary data in arbitrary formats. In some operating systems, a file is viewed as nothing more than a collection of bytes. In such an operating system, any organization of the bytes in a file (such as organizing the data into records) is a view created by the application programmer.

To perform file processing in C++, header files <iostream> and <fstream> must be included. Header <fstream> includes the definitions for the stream class templates basic_ifstream (for file input), basic_ofstream (for file output) and basic_fstream (for file input and output). Each class template has a predefined template specialization that enables char I/O. In addition, the <fstream> library provides typedef aliases for these template specializations. For example, the typedef ifstream represents a specialization of basic_ifstream that enables char input from a file. Similarly, typedef ofstream represents a specialization of basic_ofstream that enables char output to files. Also, typedef fstream represents a specialization of basic_fstream that enables char input from, and output to, files.

Files are opened by creating objects of these stream template specializations. These templates "derive" from class templates basic_istream, basic_ostream and basic_iostream, respectively. Thus, all member functions, operators and manipulators that belong to these templates (which we described in Chapter 15) also can be applied to file streams. Figure 17.3 summarizes the inheritance relationships of the I/O classes that we've discussed to this point.

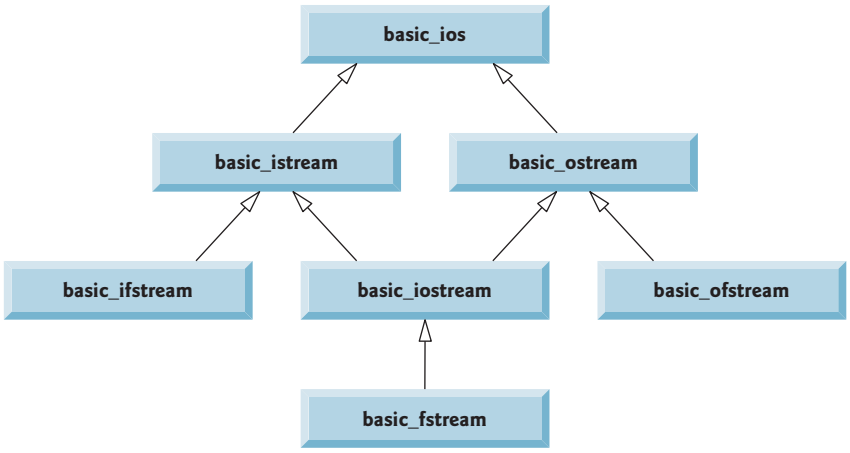

Fig. 17.3 | Portion of stream I/O template hierarchy.

17.4 Creating a Sequential File

C++ imposes no structure on a file. Thus, a concept like that of a "record" does not exist in a C++ file. Therefore, you must structure files to meet the application's requirements. In the following example, we see how you can impose a simple record structure on a file.

Figure 17.4 creates a sequential file that might be used in an accounts-receivable system to help manage the money owed by a company's credit clients. For each client, the program obtains the client's account number, name and 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 serves as the record key; that is, the program creates and maintains the file in account number order. This program assumes the user enters the records in account number order. In a comprehensive

accounts receivable system, a sorting capability would be provided for the user to enter records in any order—the records then would be sorted and written to the file.

```
1  // Fig. 17.4: Fig17_04.cpp
2  // Create a sequential file.
3  #include <iostream>
4  #include <string>
5  #include <fstream> // file stream
6  #include <cstdlib>
7  using namespace std;
8
9  int main()
10 {
11    // ofstream constructor opens file
12    ofstream outClientFile( "clients.dat", ios::out );
13
14    // exit program if unable to create file
15    if ( !outClientFile ) // overloaded ! operator
16    {
17       cerr << "File could not be opened" << endl;
18       exit( 1 );
19    } // end if
20
21    cout << "Enter the account, name, and balance." << endl
22       << "Enter end-of-file to end input.\n? ";
23
24    int account;
25    string name;
26    double balance;
27
28    // read account, name and balance from cin, then place in file
29    while ( cin >> account >> name >> balance )
30    {
31       outClientFile << account << ' ' << name << ' ' << balance << endl;
32       cout << "? ";
33    } // end while
34 } // end main
```

```
Enter the account, name, and balance.
Enter end-of-file 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. 17.4 | Creating a sequential file.

Let's examine this program. As stated previously, files are opened by creating `ifstream`, `ofstream` or `fstream` objects. In Fig. 17.4, the file is to be opened for output, so an `ofstream` object is created. Two arguments are passed to the object's constructor—the **filename** and the **file-open mode** (line 12). For an `ofstream` object, the file-open

mode can be either **ios::out** to output data to a file or **ios::app** to append data to the end of a file (without modifying any data already in the file). Existing files opened with mode ios::out are **truncated**—all data in the file is discarded. If the specified file does not yet exist, then the ofstream object creates the file, using that filename.

Line 12 creates an ofstream object named outClientFile associated with the file clients.dat that is opened for output. The arguments "clients.dat" and ios::out are passed to the ofstream constructor, which opens the file—this establishes a "line of communication" with the file. By default, ofstream objects are opened for output, so line 12 could have used the alternate statement

```
ofstream outClientFile( "clients.dat" );
```

to open clients.dat for output. Figure 17.5 lists the file-open modes.

Common Programming Error 17.1

Use caution when opening an existing file for output (ios::out), especially when you want to preserve the file's contents, which will be discarded without warning.

Mode	Description
ios::app	Append all output to the end of the file.
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::in	Open a file for input.
ios::out	Open a file for output.
ios::trunc	Discard the file's contents (this also is the default action for ios::out).
ios::binary	Open a file for binary (i.e., nontext) input or output.

Fig. 17.5 | File open modes.

An ofstream object can be created without opening a specific file—a file can be attached to the object later. For example, the statement

```
ofstream outClientFile;
```

creates an ofstream object named outClientFile. The ofstream member function **open** opens a file and attaches it to an existing ofstream object as follows:

```
outClientFile.open( "clients.dat", ios::out );
```

Common Programming Error 17.2

Not opening a file before attempting to reference it in a program will result in an error.

After creating an ofstream object and attempting to open it, the program tests whether the open operation was successful. The if statement in lines 15–19 uses the overloaded ios member function operator! to determine whether the open operation succeeded. The condition returns a true value if either the failbit or the badbit is set for the stream on the open operation. Some possible errors are attempting to open a nonexis-

tent file for reading, attempting to open a file for reading or writing without permission, and opening a file for writing when no disk space is available.

If the condition indicates an unsuccessful attempt to open the file, line 17 outputs the error message "File could not be opened", and line 18 invokes function exit to terminate the program. The argument to exit is returned to the environment from which the program was invoked. Argument 0 indicates that the program terminated normally; any other value indicates that the program terminated due to an error. The calling environment (most likely the operating system) uses the value returned by exit to respond appropriately to the error.

Another overloaded ios member function—operator void *—converts the stream to a pointer, so it can be tested as 0 (i.e., the null pointer) or nonzero (i.e., any other pointer value). When a pointer value is used as a condition, C++ interprets a null pointer in a condition as the bool value false and interprets a non-null pointer as the bool value true. If the failbit or badbit (see Chapter 15) has been set for the stream, 0 (false) is returned. The condition in the while statement of lines 29–33 invokes the operator void * member function on cin implicitly. The condition remains true as long as neither the failbit nor the badbit has been set for cin. Entering the end-of-file indicator sets the failbit for cin. The operator void * function can be used to test an input object for end-of-file instead of calling the eof member function explicitly on the input object.

If line 12 opened the file successfully, the program begins processing data. Lines 21–22 prompt the user to enter either the various fields for each record or the end-of-file indicator when data entry is complete. Figure 17.6 lists the keyboard combinations for entering end-of-file for various computer systems.

Computer system	Keyboard combination
UNIX/Linux/Mac OS X	<*Ctrl-d*> (on a line by itself)
Microsoft Windows	<*Ctrl-z*> (sometimes followed by pressing *Enter*)
VAX (VMS)	<*Ctrl-z*>

Fig. 17.6 | End-of-file key combinations for various popular computer systems.

Line 29 extracts each set of data and determines whether end-of-file has been entered. When end-of-file is encountered or bad data is entered, operator void * returns the null pointer (which converts to the bool value false) and the while statement terminates. The user enters end-of-file to inform the program to process no additional data. The end-of-file indicator is set when the user enters the end-of-file key combination. The while statement loops until the end-of-file indicator is set.

Line 31 writes a set of data to the file clients.dat, using the stream insertion operator << and the outClientFile object associated with the file at the beginning of the program. The data may be retrieved by a program designed to read the file (see Section 17.5). The file created in Fig. 17.4 is simply a text file, so it can be viewed by any text editor.

Once the user enters the end-of-file indicator, main terminates. This implicitly invokes outClientFile's destructor, which closes the clients.dat file. You also can close the ofstream object explicitly, using member function **close** in the statement

```
outClientFile.close();
```

Performance Tip 17.1

Closing files explicitly when the program no longer needs to reference them can reduce resource usage (especially if the program continues execution after closing the files).

In the sample execution for the program of Fig. 17.4, the user enters information for five accounts, then signals that data entry is complete by entering end-of-file (^Z is displayed for Microsoft Windows). This dialog window does not show how the data records appear in the file. To verify that the program created the file successfully, the next section shows how to create a program that reads this file and prints its contents.

17.5 Reading Data from a Sequential File

Files store data so it may be retrieved for processing when needed. The previous section demonstrated how to create a file for sequential access. In this section, we discuss how to read data sequentially from a file.

Figure 17.7 reads records from the `clients.dat` file that we created using the program of Fig. 17.4 and displays the contents of these records. Creating an `ifstream` object opens a file for input. The `ifstream` constructor can receive the filename and the file open mode as arguments. Line 15 creates an `ifstream` object called `inClientFile` and associates it with the `clients.dat` file. The arguments in parentheses are passed to the `ifstream` constructor function, which opens the file and establishes a "line of communication" with the file.

Good Programming Practice 17.1

Open a file for input only (using ***`ios::in`****) if the file's contents should not be modified. This prevents unintentional modification of the file's contents and is an example of the principle of least privilege.*

```
1  // Fig. 17.7: Fig17_07.cpp
2  // Reading and printing a sequential file.
3  #include <iostream>
4  #include <fstream> // file stream
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 void outputLine( int, const string, double ); // prototype
11
12 int main()
13 {
14    // ifstream constructor opens the file
15    ifstream inClientFile( "clients.dat", ios::in );
16
17    // exit program if ifstream could not open file
18    if ( !inClientFile )
19    {
20       cerr << "File could not be opened" << endl;
```

Fig. 17.7 | Reading and printing a sequential file. (Part 1 of 2.)

```
21         exit( 1 );
22      } // end if
23
24      int account;
25      string name;
26      double balance;
27
28      cout << left << setw( 10 ) << "Account" << setw( 13 )
29         << "Name" << "Balance" << endl << fixed << showpoint;
30
31      // display each record in file
32      while ( inClientFile >> account >> name >> balance )
33         outputLine( account, name, balance );
34   } // end main
35
36   // display single record from file
37   void outputLine( int account, const string name, double balance )
38   {
39      cout << left << setw( 10 ) << account << setw( 13 ) << name
40         << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
41   } // end function outputLine
```

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

Fig. 17.7 | Reading and printing a sequential file. (Part 2 of 2.)

Objects of class `ifstream` are opened for input by default. We could have used the statement

```
ifstream inClientFile( "clients.dat" );
```

to open `clients.dat` for input. Just as with an `ofstream` object, an `ifstream` object can be created without opening a specific file, because a file can be attached to it later.

The program uses the condition `!inClientFile` to determine whether the file was opened successfully before attempting to retrieve data from the file. Line 32 reads a set of data (i.e., a record) from the file. After the preceding line is executed the first time, `account` has the value `100`, `name` has the value `"Jones"` and `balance` has the value `24.98`. Each time line 32 executes, it reads another record from the file into the variables `account`, `name` and `balance`. Line 33 displays the records, using function `outputLine` (lines 37–41), which uses parameterized stream manipulators to format the data for display. When the end of file has been reached, the implicit call to `operator void *` in the `while` condition returns the null pointer (which converts to the `bool` value `false`), the `ifstream` destructor function closes the file and the program terminates.

To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found. It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program. Both `istream` and `ostream` provide member func-

tions for repositioning the **file-position pointer** (the byte number of the next byte in the file to be read or written). These member functions are **seekg** ("seek get") for `istream` and **seekp** ("seek put") for `ostream`. Each `istream` object has a "get pointer," which indicates the byte number in the file from which the next input is to occur, and each `ostream` object has a "put pointer," which indicates the byte number in the file at which the next output should be placed. The statement

```
inClientFile.seekg( 0 );
```

repositions the file-position pointer to the beginning of the file (location 0) attached to `inClientFile`. The argument to `seekg` normally is a `long` integer. A second argument can be specified to indicate the **seek direction**, which can be **`ios::beg`** (the default) for positioning relative to the beginning of a stream, **`ios::cur`** for positioning relative to the current position in a stream or **`ios::end`** for positioning relative to the end of a stream. The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location (this is also referred to as the **offset** from the beginning of the file). Some examples of positioning the "get" file-position pointer are

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

The same operations can be performed using `ostream` member function `seekp`. Member functions **`tellg`** and **`tellp`** are provided to return the current locations of the "get" and "put" pointers, respectively. The following statement assigns the "get" file-position pointer value to variable `location` of type `long`:

```
location = fileObject.tellg();
```

Figure 17.8 enables a credit manager to display the account information for those customers with zero balances (i.e., customers who do not owe the company any money), credit (negative) balances (i.e., customers to whom the company owes money), and debit (positive) balances (i.e., customers who owe the company money for goods and services received in the past). 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. Entering an invalid option displays the prompt to enter another choice. Lines 65–66 enable the program to read from the beginning of the file after the EOF marker has been read.

```
1  // Fig. 17.8: Fig17_08.cpp
2  // Credit inquiry program.
3  #include <iostream>
4  #include <fstream>
```

Fig. 17.8 | Credit inquiry program. (Part 1 of 4.)

```
#include <iomanip>
#include <string>
#include <cstdlib>
using namespace std;

enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
int getRequest();
bool shouldDisplay( int, double );
void outputLine( int, const string, double );

int main()
{
   // ifstream constructor opens the file
   ifstream inClientFile( "clients.dat", ios::in );

   // exit program if ifstream could not open file
   if ( !inClientFile )
   {
      cerr << "File could not be opened" << endl;
      exit( 1 );
   } // end if

   int request;
   int account;
   string name;
   double balance;

   // get user's request (e.g., zero, credit or debit balance)
   request = getRequest();

   // process user's request
   while ( request != END )
   {
      switch ( request )
      {
         case ZERO_BALANCE:
            cout << "\nAccounts with zero balances:\n";
            break;
         case CREDIT_BALANCE:
            cout << "\nAccounts with credit balances:\n";
            break;
         case DEBIT_BALANCE:
            cout << "\nAccounts with debit balances:\n";
            break;
      } // end switch

      // read account, name and balance from file
      inClientFile >> account >> name >> balance;

      // display file contents (until eof)
      while ( !inClientFile.eof() )
      {
```

Fig. 17.8 | Credit inquiry program. (Part 2 of 4.)

```
57           // display record
58           if ( shouldDisplay( request, balance ) )
59              outputLine( account, name, balance );
60
61           // read account, name and balance from file
62           inClientFile >> account >> name >> balance;
63        } // end inner while
64
65        inClientFile.clear(); // reset eof for next input
66        inClientFile.seekg( 0 ); // reposition to beginning of file
67        request = getRequest(); // get additional request from user
68     } // end outer while
69
70     cout << "End of run." << endl;
71  } // end main
72
73  // obtain request from user
74  int getRequest()
75  {
76     int request; // request from user
77
78     // display request options
79     cout << "\nEnter request" << endl
80        << " 1 - List accounts with zero balances" << endl
81        << " 2 - List accounts with credit balances" << endl
82        << " 3 - List accounts with debit balances" << endl
83        << " 4 - End of run" << fixed << showpoint;
84
85     do // input user request
86     {
87        cout << "\n? ";
88        cin >> request;
89     } while ( request < ZERO_BALANCE && request > END );
90
91     return request;
92  } // end function getRequest
93
94  // determine whether to display given record
95  bool shouldDisplay( int type, double balance )
96  {
97     // determine whether to display zero balances
98     if ( type == ZERO_BALANCE && balance == 0 )
99        return true;
100
101    // determine whether to display credit balances
102    if ( type == CREDIT_BALANCE && balance < 0 )
103       return true;
104
105    // determine whether to display debit balances
106    if ( type == DEBIT_BALANCE && balance > 0 )
107       return true;
108
```

Fig. 17.8 | Credit inquiry program. (Part 3 of 4.)

```
109     return false;
110  } // end function shouldDisplay
111
112  // display single record from file
113  void outputLine( int account, const string name, double balance )
114  {
115     cout << left << setw( 10 ) << account << setw( 13 ) << name
116        << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
117  } // end function outputLine
```

```
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

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 2

Accounts with credit balances:
400       Stone         -42.16

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 4
End of run.
```

Fig. 17.8 | Credit inquiry program. (Part 4 of 4.)

17.6 Updating Sequential Files

Data that is formatted and written to a sequential file as shown in Section 17.4 cannot be modified without the risk of destroying other data in the file. For example, if the name

"White" needs to be changed to "Worthington," the old name cannot be overwritten without corrupting the file. The record for White was written to the file as

```
300 White 0.00
```

If this record were rewritten beginning at the same location in the file using the longer name, the record would be

```
300 Worthington 0.00
```

The new record contains six more characters than the original record. Therefore, the characters beyond the second "o" in "Worthington" would overwrite the beginning of the next sequential record in the file. The problem is that, in the formatted input/output model using the stream insertion operator << and the stream extraction operator >>, fields—and hence records—can vary in size. For example, values 7, 14, –117, 2074, and 27383 are all ints, which store the same number of "raw data" bytes internally (typically four bytes on today's popular 32-bit machines). However, these integers become different-sized fields when output as formatted text (character sequences). Therefore, the formatted input/output model usually is not used to update records in place.

Such updating can be done awkwardly. For example, to make the preceding name change, the records before 300 White 0.00 in a sequential file could be copied to a new file, the updated record then written to the new file, and the records after 300 White 0.00 copied to the new file. This requires processing every record in the file to update one record. If many records are being updated in one pass of the file, though, this technique can be acceptable.

17.7 Random-Access Files

So far, we've seen how to create sequential files and search them to locate information. Sequential files are inappropriate for **instant-access applications**, in which a particular record must be located immediately. Common instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated teller machines and other kinds of **transaction-processing systems** that require rapid access to specific data. A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds. This kind of instant access is made possible with **random-access files.** Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.

As we've said, C++ does not impose structure on a file. So the application that wants to use random-access files must create them. A variety of techniques can be used. Perhaps the easiest method is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. We soon will see how this facilitates immediate access to specific records, even in large files.

Figure 17.9 illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long). A random-access file is like a railroad train with many same-size cars—some empty and some with contents.

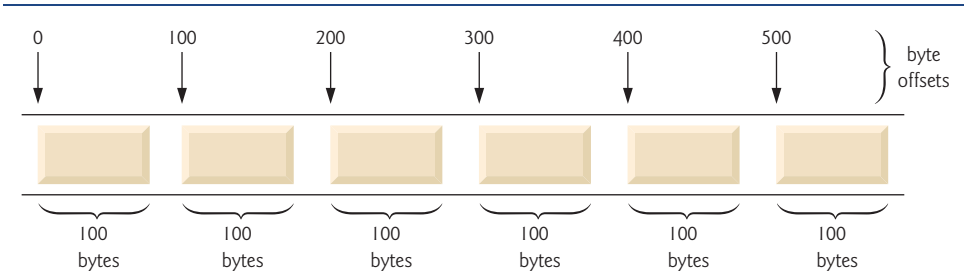

Fig. 17.9 | C++ view of a random-access file.

Data can be inserted into 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 following sections, we explain how to create a random-access file, enter data into the file, read the data both sequentially and randomly, update the data and delete data that is no longer needed.

17.8 Creating a Random-Access File

The `ostream` member function `write` outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream. When the stream is associated with a file, function `write` writes the data at the location in the file specified by the "put" file-position pointer. The `istream` member function `read` inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address. If the stream is associated with a file, function `read` inputs bytes at the location in the file specified by the "get" file-position pointer.

Writing Bytes with `ostream` Member Function `write`

When writing the integer `number` to a file, instead of using the statement

```
outFile << number;
```

which for a four-byte integer could print as few digits as one or as many as 11 (10 digits plus a sign, each requiring a single byte of storage), we can use the statement

```
outFile.write( reinterpret_cast< const char * >( &number ),
   sizeof( number ) );
```

which always writes the binary version of the integer's four bytes (on a machine with four-byte integers). Function `write` treats its first argument as a group of bytes by viewing the object in memory as a `const char *`, which is a pointer to a byte. Starting from that location, function `write` outputs the number of bytes specified by its second argument—an integer of type `size_t`. As we'll see, `istream` function `read` can subsequently be used to read the four bytes back into integer variable `number`.

Converting Between Pointer Types with the `reinterpret_cast` Operator

Unfortunately, most pointers that we pass to function `write` as the first argument are not of type `const char *`. To output objects of other types, we must convert the pointers to those objects to type `const char *`; otherwise, the compiler will not compile calls to func-

tion `write`. C++ provides the **`reinterpret_cast`** operator for cases like this in which a pointer of one type must be cast to an unrelated pointer type. Without a `reinterpret_cast`, the `write` statement that outputs the integer `number` will not compile because the compiler does not allow a pointer of type `int *` (the type returned by the expression `&number`) to be passed to a function that expects an argument of type `const char *`—as far as the compiler is concerned, these types are incompatible.

A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points. Instead, it requests that the compiler reinterpret the operand as the target type (specified in the angle brackets following the keyword `reinterpret_cast`). In Fig. 17.12, we use `reinterpret_cast` to convert a `ClientData` pointer to a `const char *`, which reinterprets a `ClientData` object as bytes to be output to a file. Random-access file-processing programs rarely write a single field to a file. Typically, they write one object of a class at a time, as we show in the following examples.

Error-Prevention Tip 17.1

It's easy to use `reinterpret_cast` to perform dangerous manipulations that could lead to serious execution-time errors.

Portability Tip 17.1

Using `reinterpret_cast` is compiler dependent and can cause programs to behave differently on different platforms. The `reinterpret_cast` operator should not be used unless absolutely necessary.

Portability Tip 17.2

A program that reads unformatted data (written by `write`) must be compiled and executed on a system compatible with the program that wrote the data, because different systems may represent internal data differently.

Credit Processing Program

Consider the following problem statement:

> *Create a credit-processing program capable of storing at most 100 fixed-length records for a company that can have up to 100 customers. Each record should consist of an account number that acts as the record key, a last name, a first name and a balance. The program should be able to update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.*

The next several sections introduce the techniques for creating this credit-processing program. Figure 17.12 illustrates opening a random-access file, defining the record format using an object of class `ClientData` (Figs. 17.10–17.11) and writing data to the disk in binary format. This program initializes all 100 records of the file `credit.dat` with empty objects, using function `write`. Each empty object contains `0` for the account number, the null string (represented by empty quotation marks) for the last and first name and `0.0` for the balance. Each record is initialized with the amount of empty space in which the account data will be stored.

Objects of class `string` do not have uniform size, rather they use dynamically allocated memory to accommodate strings of various lengths. We must maintain fixed-length records, so class `ClientData` stores the client's first and last name in fixed-length `char` arrays (declared in Fig. 17.10, lines 32–33). Member functions `setLastName` (Fig. 17.11,

lines 36–43) and `setFirstName` (Fig. 17.11, lines 52–59) each copy the characters of a `string` object into the corresponding `char` array. Consider function `setLastName`. Line 39 invokes `string` member function **`size`** to get the length of `lastNameString`. Line 40 ensures that `length` is fewer than 15 characters, then line 41 copies `length` characters from `lastNameString` into the `char` array `lastName` using string member function `copy`. Member function `setFirstName` performs the same steps for the first name.

```
1  // Fig. 17.10: ClientData.h
2  // Class ClientData definition used in Fig. 17.12–Fig. 17.15.
3  #ifndef CLIENTDATA_H
4  #define CLIENTDATA_H
5
6  #include <string>
7  using namespace std;
8
9  class ClientData
10 {
11 public:
12    // default ClientData constructor
13    ClientData( int = 0, string = "", string = "", double = 0.0 );
14
15    // accessor functions for accountNumber
16    void setAccountNumber( int );
17    int getAccountNumber() const;
18
19    // accessor functions for lastName
20    void setLastName( string );
21    string getLastName() const;
22
23    // accessor functions for firstName
24    void setFirstName( string );
25    string getFirstName() const;
26
27    // accessor functions for balance
28    void setBalance( double );
29    double getBalance() const;
30 private:
31    int accountNumber;
32    char lastName[ 15 ];
33    char firstName[ 10 ];
34    double balance;
35 }; // end class ClientData
36
37 #endif
```

Fig. 17.10 | `ClientData` class header file.

```
1  // Fig. 17.11: ClientData.cpp
2  // Class ClientData stores customer's credit information.
3  #include <string>
```

Fig. 17.11 | `ClientData` class represents a customer's credit information. (Part 1 of 3.)

```
 4  #include "ClientData.h"
 5  using namespace std;
 6
 7  // default ClientData constructor
 8  ClientData::ClientData( int accountNumberValue,
 9     string lastNameValue, string firstNameValue, double balanceValue )
10  {
11     setAccountNumber( accountNumberValue );
12     setLastName( lastNameValue );
13     setFirstName( firstNameValue );
14     setBalance( balanceValue );
15  } // end ClientData constructor
16
17  // get account-number value
18  int ClientData::getAccountNumber() const
19  {
20     return accountNumber;
21  } // end function getAccountNumber
22
23  // set account-number value
24  void ClientData::setAccountNumber( int accountNumberValue )
25  {
26     accountNumber = accountNumberValue; // should validate
27  } // end function setAccountNumber
28
29  // get last-name value
30  string ClientData::getLastName() const
31  {
32     return lastName;
33  } // end function getLastName
34
35  // set last-name value
36  void ClientData::setLastName( string lastNameString )
37  {
38     // copy at most 15 characters from string to lastName
39     int length = lastNameString.size();
40     length = ( length < 15 ? length : 14 );
41     lastNameString.copy( lastName, length );
42     lastName[ length ] = '\0'; // append null character to lastName
43  } // end function setLastName
44
45  // get first-name value
46  string ClientData::getFirstName() const
47  {
48     return firstName;
49  } // end function getFirstName
50
51  // set first-name value
52  void ClientData::setFirstName( string firstNameString )
53  {
54     // copy at most 10 characters from string to firstName
55     int length = firstNameString.size();
56     length = ( length < 10 ? length : 9 );
```

Fig. 17.11 | `ClientData` class represents a customer's credit information. (Part 2 of 3.)

```
57    firstNameString.copy( firstName, length );
58    firstName[ length ] = '\0'; // append null character to firstName
59 } // end function setFirstName
60
61 // get balance value
62 double ClientData::getBalance() const
63 {
64    return balance;
65 } // end function getBalance
66
67 // set balance value
68 void ClientData::setBalance( double balanceValue )
69 {
70    balance = balanceValue;
71 } // end function setBalance
```

Fig. 17.11 | `ClientData` class represents a customer's credit information. (Part 3 of 3.)

In Fig. 17.12, line 11 creates an `ofstream` object for the file `credit.dat`. The second argument to the constructor—`ios::out | ios::binary`—indicates that we are opening the file for output in binary mode, which is required if we are to write fixed-length records. Lines 24–25 cause the `blankClient` to be written to the `credit.dat` file associated with `ofstream` object `outCredit`. Remember that operator `sizeof` returns the size in bytes of the object contained in parentheses (see Chapter 8). The first argument to function `write` at line 24 must be of type `const char *`. However, the data type of `&blankClient` is `ClientData *`. To convert `&blankClient` to `const char *`, line 24 uses the cast operator `reinterpret_cast`, so the call to `write` compiles without issuing a compilation error.

```
1  // Fig. 17.12: Fig17_12.cpp
2  // Creating a randomly accessed file.
3  #include <iostream>
4  #include <fstream>
5  #include <cstdlib>
6  #include "ClientData.h" // ClientData class definition
7  using namespace std;
8
9  int main()
10 {
11    ofstream outCredit( "credit.dat", ios::out | ios::binary );
12
13    // exit program if ofstream could not open file
14    if ( !outCredit )
15    {
16       cerr << "File could not be opened." << endl;
17       exit( 1 );
18    } // end if
19
20    ClientData blankClient; // constructor zeros out each data member
21
```

Fig. 17.12 | Creating a random-access file with 100 blank records sequentially. (Part 1 of 2.)

```
22     // output 100 blank records to file
23     for ( int i = 0; i < 100; i++ )
24        outCredit.write( reinterpret_cast< const char * >( &blankClient ),
25           sizeof( ClientData ) );
26  } // end main
```

Fig. 17.12 | Creating a random-access file with 100 blank records sequentially. (Part 2 of 2.)

17.9 Writing Data Randomly to a Random-Access File

Figure 17.13 writes data to the file `credit.dat` and uses the combination of `fstream` functions `seekp` and `write` to store data at exact locations in the file. Function `seekp` sets the "put" file-position pointer to a specific position in the file, then `write` outputs the data. Line 6 includes the header file `ClientData.h` defined in Fig. 17.10, so the program can use `ClientData` objects.

```
1  // Fig. 17.13: Fig17_13.cpp
2  // Writing to a random-access file.
3  #include <iostream>
4  #include <fstream>
5  #include <cstdlib>
6  #include "ClientData.h" // ClientData class definition
7  using namespace std;
8
9  int main()
10 {
11    int accountNumber;
12    string lastName;
13    string firstName;
14    double balance;
15
16    fstream outCredit( "credit.dat", ios::in | ios::out | ios::binary );
17
18    // exit program if fstream cannot open file
19    if ( !outCredit )
20    {
21       cerr << "File could not be opened." << endl;
22       exit( 1 );
23    } // end if
24
25    cout << "Enter account number (1 to 100, 0 to end input)\n? ";
26
27    // require user to specify account number
28    ClientData client;
29    cin >> accountNumber;
30
31    // user enters information, which is copied into file
32    while ( accountNumber > 0 && accountNumber <= 100 )
33    {
```

Fig. 17.13 | Writing to a random-access file. (Part 1 of 2.)

```
34         // user enters last name, first name and balance
35         cout << "Enter lastname, firstname, balance\n? ";
36         cin >> lastName;
37         cin >> firstName;
38         cin >> balance;
39
40         // set record accountNumber, lastName, firstName and balance values
41         client.setAccountNumber( accountNumber );
42         client.setLastName( lastName );
43         client.setFirstName( firstName );
44         client.setBalance( balance );
45
46         // seek position in file of user-specified record
47         outCredit.seekp( ( client.getAccountNumber() - 1 ) *
48            sizeof( ClientData ) );
49
50         // write user-specified information in file
51         outCredit.write( reinterpret_cast< const char * >( &client ),
52            sizeof( ClientData ) );
53
54         // enable user to enter another account
55         cout << "Enter account number\n? ";
56         cin >> accountNumber;
57      } // end while
58   } // end main
```

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Fig. 17.13 | Writing to a random-access file. (Part 2 of 2.)

Lines 47–48 position the "put" file-position pointer for object `outCredit` to the byte location calculated by

```
( client.getAccountNumber() - 1 ) * sizeof( ClientData )
```

Because the account number is between 1 and 100, 1 is subtracted from the account number when calculating the byte location of the record. Thus, for record 1, the file-position pointer is set to byte 0 of the file. Line 16 uses the `fstream` object `outCredit` to open the existing `credit.dat` file. The file is opened for input and output in binary mode by combining the file-open modes `ios::in`, `ios::out` and `ios::binary`. Multiple file-open modes are combined by separating each open mode from the next with the bitwise inclusive OR operator (`|`). Opening the existing `credit.dat` file in this manner ensures that this program can manipulate the records written to the file by the program of Fig. 17.12, rather than creating the file from scratch. Chapter 21, Bits, Characters, C Strings and `structs`, discusses the bitwise inclusive OR operator in detail.

17.10 Reading from a Random-Access File Sequentially

In the previous sections, we created a random-access file and wrote data to that file. In this section, we develop a program that reads the file sequentially and prints only those records that contain data. These programs produce an additional benefit. See if you can determine what it is; we'll reveal it at the end of this section.

The `istream` function `read` inputs a specified number of bytes from the current position in the specified stream into an object. For example, lines 30–31 from Fig. 17.14 read the number of bytes specified by `sizeof(ClientData)` from the file associated with `ifstream` object `inCredit` and store the data in the `client` record. Function `read` requires a first argument of type `char *`. Since `&client` is of type `ClientData *`, `&client` must be cast to `char *` using the cast operator `reinterpret_cast`.

```
1  // Fig. 17.14: Fig17_14.cpp
2  // Reading a random-access file sequentially.
3  #include <iostream>
4  #include <iomanip>
5  #include <fstream>
6  #include <cstdlib>
7  #include "ClientData.h" // ClientData class definition
8  using namespace std;
9
10 void outputLine( ostream&, const ClientData & ); // prototype
11
12 int main()
13 {
14    ifstream inCredit( "credit.dat", ios::in | ios::binary );
15
16    // exit program if ifstream cannot open file
17    if ( !inCredit )
18    {
19       cerr << "File could not be opened." << endl;
20       exit( 1 );
21    } // end if
22
```

Fig. 17.14 | Reading a random-access file sequentially. (Part 1 of 2.)

```
23      cout << left << setw( 10 ) << "Account" << setw( 16 )
24         << "Last Name" << setw( 11 ) << "First Name" << left
25         << setw( 10 ) << right << "Balance" << endl;
26
27      ClientData client; // create record
28
29      // read first record from file
30      inCredit.read( reinterpret_cast< char * >( &client ),
31         sizeof( ClientData ) );
32
33      // read all records from file
34      while ( inCredit && !inCredit.eof() )
35      {
36         // display record
37         if ( client.getAccountNumber() != 0 )
38            outputLine( cout, client );
39
40         // read next from file
41         inCredit.read( reinterpret_cast< char * >( &client ),
42            sizeof( ClientData ) );
43      } // end while
44   } // end main
45
46   // display single record
47   void outputLine( ostream &output, const ClientData &record )
48   {
49      output << left << setw( 10 ) << record.getAccountNumber()
50         << setw( 16 ) << record.getLastName()
51         << setw( 11 ) << record.getFirstName()
52         << setw( 10 ) << setprecision( 2 ) << right << fixed
53         << showpoint << record.getBalance() << endl;
54   } // end function outputLine
```

```
Account   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. 17.14 | Reading a random-access file sequentially. (Part 2 of 2.)

Figure 17.14 reads every record in the `credit.dat` file sequentially, checks each record to determine whether it contains data, and displays formatted outputs for records containing data. The condition in line 34 uses the `ios` member function `eof` to determine when the end of file is reached and causes execution of the `while` statement to terminate. Also, if an error occurs when reading from the file, the loop terminates, because `inCredit` evaluates to `false`. The data input from the file is output by function `outputLine` (lines 47–54), which takes two arguments—an `ostream` object and a `clientData` structure to be output. The `ostream` parameter type is interesting, because any `ostream` object (such as `cout`) or any object of a derived class of `ostream` (such as an object of type `ofstream`) can be supplied as the argument. This means that the same function can be used, for example,

to perform output to the standard-output stream and to a file stream without writing separate functions.

What about that additional benefit we promised? If you examine the output window, you'll notice that the records are listed in sorted order (by account number). This is a consequence of how we stored these records in the file, using direct-access techniques. Compared to the insertion sort we used in Chapter 7, sorting using direct-access techniques is relatively fast. The speed is achieved by making the file large enough to hold every possible record that might be created. This, of course, means that the file could be occupied sparsely most of the time, resulting in a waste of storage. This is another example of the space-time trade-off: By using large amounts of space, we can develop a much faster sorting algorithm. Fortunately, the continuous reduction in price of storage units has made this less of an issue.

17.11 Case Study: A Transaction-Processing Program

We now present a substantial transaction-processing program (Fig. 17.15) using a random-access file to achieve "instant-access" processing. The program maintains a bank's account information. The program updates existing accounts, adds new accounts, deletes accounts and stores a formatted listing of all current accounts in a text file. We assume that the program of Fig. 17.12 has been executed to create the file `credit.dat` and that the program of Fig. 17.13 has been executed to insert the initial data.

```
1  // Fig. 17.15: Fig17_15.cpp
2  // This program reads a random-access file sequentially, updates
3  // data previously written to the file, creates data to be placed
4  // in the file, and deletes data previously stored in the file.
5  #include <iostream>
6  #include <fstream>
7  #include <iomanip>
8  #include <cstdlib>
9  #include "ClientData.h" // ClientData class definition
10 using namespace std;
11
12 int enterChoice();
13 void createTextFile( fstream& );
14 void updateRecord( fstream& );
15 void newRecord( fstream& );
16 void deleteRecord( fstream& );
17 void outputLine( ostream&, const ClientData & );
18 int getAccount( const char * const );
19
20 enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
21
22 int main()
23 {
24    // open file for reading and writing
25    fstream inOutCredit( "credit.dat", ios::in | ios::out | ios::binary );
26
```

Fig. 17.15 | Bank account program. (Part 1 of 6.)

```
27     // exit program if fstream cannot open file
28     if ( !inOutCredit )
29     {
30        cerr << "File could not be opened." << endl;
31        exit ( 1 );
32     } // end if
33
34     int choice; // store user choice
35
36     // enable user to specify action
37     while ( ( choice = enterChoice() ) != END )
38     {
39        switch ( choice )
40        {
41           case PRINT: // create text file from record file
42              createTextFile( inOutCredit );
43              break;
44           case UPDATE: // update record
45              updateRecord( inOutCredit );
46              break;
47           case NEW: // create record
48              newRecord( inOutCredit );
49              break;
50           case DELETE: // delete existing record
51              deleteRecord( inOutCredit );
52              break;
53           default: // display error if user does not select valid choice
54              cerr << "Incorrect choice" << endl;
55              break;
56        } // end switch
57
58        inOutCredit.clear(); // reset end-of-file indicator
59     } // end while
60  } // end main
61
62  // enable user to input menu choice
63  int enterChoice()
64  {
65     // display available options
66     cout << "\nEnter your choice" << endl
67        << "1 - store a formatted text file of accounts" << endl
68        << "    called \"print.txt\" for printing" << endl
69        << "2 - update an account" << endl
70        << "3 - add a new account" << endl
71        << "4 - delete an account" << endl
72        << "5 - end program\n? ";
73
74     int menuChoice;
75     cin >> menuChoice; // input menu selection from user
76     return menuChoice;
77  } // end function enterChoice
78
```

Fig. 17.15 | Bank account program. (Part 2 of 6.)

```
79  // create formatted text file for printing
80  void createTextFile( fstream &readFromFile )
81  {
82     // create text file
83     ofstream outPrintFile( "print.txt", ios::out );
84
85     // exit program if ofstream cannot create file
86     if ( !outPrintFile )
87     {
88        cerr << "File could not be created." << endl;
89        exit( 1 );
90     } // end if
91
92     outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
93        << "Last Name" << setw( 11 ) << "First Name" << right
94        << setw( 10 ) << "Balance" << endl;
95
96     // set file-position pointer to beginning of readFromFile
97     readFromFile.seekg( 0 );
98
99     // read first record from record file
100    ClientData client;
101    readFromFile.read( reinterpret_cast< char * >( &client ),
102       sizeof( ClientData ) );
103
104    // copy all records from record file into text file
105    while ( !readFromFile.eof() )
106    {
107       // write single record to text file
108       if ( client.getAccountNumber() != 0 ) // skip empty records
109          outputLine( outPrintFile, client );
110
111       // read next record from record file
112       readFromFile.read( reinterpret_cast< char * >( &client ),
113          sizeof( ClientData ) );
114    } // end while
115 } // end function createTextFile
116
117 // update balance in record
118 void updateRecord( fstream &updateFile )
119 {
120    // obtain number of account to update
121    int accountNumber = getAccount( "Enter account to update" );
122
123    // move file-position pointer to correct record in file
124    updateFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
125
126    // read first record from file
127    ClientData client;
128    updateFile.read( reinterpret_cast< char * >( &client ),
129       sizeof( ClientData ) );
130
```

Fig. 17.15 | Bank account program. (Part 3 of 6.)

```
131    // update record
132    if ( client.getAccountNumber() != 0 )
133    {
134       outputLine( cout, client ); // display the record
135
136       // request user to specify transaction
137       cout << "\nEnter charge (+) or payment (-): ";
138       double transaction; // charge or payment
139       cin >> transaction;
140
141       // update record balance
142       double oldBalance = client.getBalance();
143       client.setBalance( oldBalance + transaction );
144       outputLine( cout, client ); // display the record
145
146       // move file-position pointer to correct record in file
147       updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
148
149       // write updated record over old record in file
150       updateFile.write( reinterpret_cast< const char * >( &client ),
151          sizeof( ClientData ) );
152    } // end if
153    else // display error if account does not exist
154       cerr << "Account #" << accountNumber
155          << " has no information." << endl;
156 } // end function updateRecord
157
158 // create and insert record
159 void newRecord( fstream &insertInFile )
160 {
161    // obtain number of account to create
162    int accountNumber = getAccount( "Enter new account number" );
163
164    // move file-position pointer to correct record in file
165    insertInFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
166
167    // read record from file
168    ClientData client;
169    insertInFile.read( reinterpret_cast< char * >( &client ),
170       sizeof( ClientData ) );
171
172    // create record, if record does not previously exist
173    if ( client.getAccountNumber() == 0 )
174    {
175       string lastName;
176       string firstName;
177       double balance;
178
179       // user enters last name, first name and balance
180       cout << "Enter lastname, firstname, balance\n? ";
181       cin >> setw( 15 ) >> lastName;
182       cin >> setw( 10 ) >> firstName;
183       cin >> balance;
```

Fig. 17.15 | Bank account program. (Part 4 of 6.)

```
184
185         // use values to populate account values
186         client.setLastName( lastName );
187         client.setFirstName( firstName );
188         client.setBalance( balance );
189         client.setAccountNumber( accountNumber );
190
191         // move file-position pointer to correct record in file
192         insertInFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
193
194         // insert record in file
195         insertInFile.write( reinterpret_cast< const char * >( &client ),
196            sizeof( ClientData ) );
197      } // end if
198      else // display error if account already exists
199         cerr << "Account #" << accountNumber
200            << " already contains information." << endl;
201   } // end function newRecord
202
203   // delete an existing record
204   void deleteRecord( fstream &deleteFromFile )
205   {
206      // obtain number of account to delete
207      int accountNumber = getAccount( "Enter account to delete" );
208
209      // move file-position pointer to correct record in file
210      deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
211
212      // read record from file
213      ClientData client;
214      deleteFromFile.read( reinterpret_cast< char * >( &client ),
215         sizeof( ClientData ) );
216
217      // delete record, if record exists in file
218      if ( client.getAccountNumber() != 0 )
219      {
220         ClientData blankClient; // create blank record
221
222         // move file-position pointer to correct record in file
223         deleteFromFile.seekp( ( accountNumber - 1 ) *
224            sizeof( ClientData ) );
225
226         // replace existing record with blank record
227         deleteFromFile.write(
228            reinterpret_cast< const char * >( &blankClient ),
229            sizeof( ClientData ) );
230
231         cout << "Account #" << accountNumber << " deleted.\n";
232      } // end if
233      else // display error if record does not exist
234         cerr << "Account #" << accountNumber << " is empty.\n";
235   } // end deleteRecord
236
```

Fig. 17.15 | Bank account program. (Part 5 of 6.)

```
237 // display single record
238 void outputLine( ostream &output, const ClientData &record )
239 {
240    output << left << setw( 10 ) << record.getAccountNumber()
241       << setw( 16 ) << record.getLastName()
242       << setw( 11 ) << record.getFirstName()
243       << setw( 10 ) << setprecision( 2 ) << right << fixed
244       << showpoint << record.getBalance() << endl;
245 } // end function outputLine
246
247 // obtain account-number value from user
248 int getAccount( const char * const prompt )
249 {
250    int accountNumber;
251
252    // obtain account-number value
253    do
254    {
255       cout << prompt << " (1 - 100): ";
256       cin >> accountNumber;
257    } while ( accountNumber < 1 || accountNumber > 100 );
258
259    return accountNumber;
260 } // end function getAccount
```

Fig. 17.15 | Bank account program. (Part 6 of 6.)

The program has five options (Option 5 is for terminating the program). Option 1 calls function `createTextFile` to store a formatted list of all the account information in a text file called `print.txt` that may be printed. Function `createTextFile` (lines 80–115) takes an `fstream` object as an argument to be used to input data from the `credit.dat` file. Function `createTextFile` invokes `istream` member function `read` (lines 101–102) and uses the sequential-file-access techniques of Fig. 17.14 to input data from `credit.dat`. Function `outputLine`, discussed in Section 17.10, is used to output the data to file `print.txt`. Note that `createTextFile` uses `istream` member function `seekg` (line 97) to ensure that the file-position pointer is at the beginning of the file. After choosing Option 1, the `print.txt` file contains

```
Account   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 calls `updateRecord` (lines 118–156) to update an account. This function updates only an existing record, so the function first determines whether the specified record is empty. Lines 128–129 read data into object `client`, using `istream` member function `read`. Then line 132 compares the value returned by `getAccountNumber` of the `client` object to zero to determine whether the record contains information. If this value

is zero, lines 154–155 print an error message indicating that the record is empty. If the record contains information, line 134 displays the record, using function `outputLine`, line 139 inputs the transaction amount and lines 142–151 calculate the new balance and rewrite 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 calls function `newRecord` (lines 159–201) 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 account exists (lines 199–200). This function adds a new account in the same manner as the program of Fig. 17.12. A typical output for Option 3 is

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

Option 4 calls function `deleteRecord` (lines 204–235) to delete a record from the file. Line 207 prompts the user to enter the account number. Only an existing record may be deleted, so, if the specified account is empty, line 234 displays an error message. If the account exists, lines 227–229 reinitialize that account by copying an empty record (`blankClient`) to the file. Line 231 displays a message to inform the user that the record has been deleted. A typical output for Option 4 is

```
Enter account to delete (1 - 100): 29
Account #29 deleted.
```

Line 25 opens the `credit.dat` file by creating an `fstream` object for both reading and writing, using modes `ios::in` and `ios::out` "or-ed" together.

17.12 Overview of Object Serialization

This chapter and Chapter 15 introduced the object-oriented style of input/output. However, our examples concentrated on I/O of fundamental types rather than objects of user-defined types. In Chapter 11, we showed how to input and output objects using operator overloading. We accomplished object input by overloading the stream extraction operator, `>>`, for the appropriate `istream`. We accomplished object output by overloading the stream insertion operator, `<<`, for the appropriate `ostream`. In both cases, only an object's data members were input or output, and, in each case, they were in a format meaningful only for objects of that particular type. An object's member functions are not input or output with the object's data; rather, one copy of the class's member functions remains available internally and is shared by all objects of the class.

When object data members are output to a disk file, we lose the object's type information. We store only the values of the object's attributes, not type information, on the disk. If the program that reads this data knows the object type to which the data corresponds, the program can read the data into an object of that type as we did in our random-access file examples.

An interesting problem occurs when we store objects of different types in the same file. How can we distinguish them (or their collections of data members) as we read them into a program? The problem is that objects typically do not have type fields (we discussed this issue in Chapter 13).

One approach used by several programming languages is called **object serialization**. A so-called **serialized object** is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. After a serialized object has been written to a file, it can be read from the file and **deserialized**—that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory. C++ does not provide a built-in serialization mechanism; however, there are third party and open source C++ libraries that support object serialization. The open source Boost C++ Libraries (`www.boost.org`) provide support for serializing objects in text, binary and extensible markup language (XML) formats (`www.boost.org/libs/serialization/doc/index.html`). We overview the Boost C++ Libraries in Chapter 23.

17.13 Wrap-Up

In this chapter, we presented various file-processing techniques to manipulate persistent data. You learned that data is stored in computers in the form of `0`s and `1`s, and that combinations of these values form bytes, fields, records and eventually files. You were introduced to the differences between character-based and byte-based streams, and to several file-processing class templates in header file `<fstream>`. Then, you learned how to use sequential file processing to manipulate records stored in order, by the record-key field. You also learned how to use random-access files to instantly retrieve and manipulate fixed-length records. We presented a substantial transaction-processing case study using a random-access file to achieve "instant-access" processing. Finally, we discussed the basic concepts of object serialization. In the next chapter, we discuss typical string-manipulation operations provided by class template `basic_string`. We also introduce string stream-processing capabilities that allow strings to be input from and output to memory.

Summary

Section 17.1 Introduction

- Files are used for data persistence—permanent retention of data.
- Computers store files on secondary storage devices, such as hard disks, CDs, DVDs, flash memory and tapes.

Section 17.2 Data Hierarchy

- The smallest data item that computers support is called a bit (short for "binary digit"—a digit that can assume one of two values, 0 or 1).
- Digits, letters and special symbols are referred to as characters.

- The set of all characters used to write programs and represent data items on a particular computer is called that computer's character set.
- Bytes are composed of eight bits.
- Just as characters are composed of bits, fields are composed of characters. A field is a group of characters that conveys some meaning.
- Typically, a record (i.e., a class in C++) is composed of several fields (i.e., data members in C++).
- At least one field in a record is chosen as a record key to identify a record as belonging to a particular person or entity that is distinct from all other records in the file.
- In a sequential file, records typically are stored in order by a record-key field.

Section 17.3 Files and Streams

- C++ views each file as a sequence of bytes.
- 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, an object is created, and a stream is associated with the object.
- To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included.
- Header `<fstream>` includes the definitions for the stream class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input and output).
- Each class template has a predefined template specialization that enables char I/O. The `<fstream>` library provides `typedef` aliases for these template specializations. The `typedef ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file. The `typedef ofstream` represents a specialization of `basic_ofstream` that enables `char` output to files. The `typedef fstream` represents a specialization of `basic_fstream` that enables `char` input from, and output to, files.
- The file-processing templates derive from class templates `basic_istream`, `basic_ostream` and `basic_iostream`, respectively. Thus, all member functions, operators and manipulators that belong to these templates also can be applied to file streams.

Section 17.4 Creating a Sequential File

- C++ imposes no structure on a file; you must structure files to meet the application's requirements.
- A file can be opened for output when an `ofstream` object is created. Two arguments are passed to the object's constructor—the filename and the file-open mode.
- For an `ofstream` object, the file-open mode can be either `ios::out` to output data to a file or `ios::app` to append data to the end of a file. Existing files opened with mode `ios::out` are truncated. If the specified file does not exist, the `ofstream` object creates the file using that filename.
- By default, `ofstream` objects are opened for output.
- An `ofstream` object can be created without opening a specific file—a file can be attached to the object later with member function `open`.
- The `ios` member function `operator!` determines whether a stream was opened correctly. This operator can be used in a condition that returns a true value if either the `failbit` or the `badbit` is set for the stream on the open operation.
- The `ios` member function `operator void *` converts a stream to a pointer, so it can be compared to `0`. When a pointer value is used as a condition, a null pointer represents `false` and a non-null pointer represents `true`. If the `failbit` or `badbit` has been set for a stream, `0` (`false`) is returned.
- Entering the end-of-file indicator sets the `failbit` for `cin`.
- The `operator void *` function can be used to test an input object for end-of-file instead of calling the `eof` member function explicitly on the input object.

- When a stream object's destructor is called, the corresponding stream is closed. You also can close the stream object explicitly, using the stream's `close` member function.
- Closing files explicitly when they're no longer needed can reduce a program's resource usage.

Section 17.5 Reading Data from a Sequential File

- Files store data so it may be retrieved for processing when needed.
- Creating an `ifstream` object opens a file for input. The `ifstream` constructor can receive the filename and the file open mode as arguments.
- Open a file for input only if the file's contents should not be modified.
- Objects of class `ifstream` are opened for input by default.
- An `ifstream` object can be created without opening a specific file; a file can be attached to it later.
- To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found.
- Both `istream` and `ostream` provide member functions for repositioning the file-position pointer. These member functions are `seekg` ("seek get") for `istream` and `seekp` ("seek put") for `ostream`. Each `istream` object has a "get pointer," which indicates the byte number in the file from which the next input is to occur, and each `ostream` object has a "put pointer," which indicates the byte number in the file at which the next output should be placed.
- The argument to `seekg` normally is a long integer. A second argument can be specified to indicate the seek direction, which can be `ios::beg` (the default) for positioning relative to the beginning of a stream, `ios::cur` for positioning relative to the current position in a stream or `ios::end` for positioning relative to the end of a stream.
- The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location (i.e., the offset from the beginning of the file).
- Member functions `tellg` and `tellp` are provided to return the current locations of the "get" and "put" pointers, respectively.

Section 17.6 Updating Sequential Files

- Data that is formatted and written to a sequential file cannot be modified without the risk of destroying other data in the file. The problem is that records can vary in size.

Section 17.7 Random-Access Files

- Sequential files are inappropriate for instant-access applications, in which a particular record must be located immediately.
- Instant access is made possible with random-access files. Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.
- The easiest method to format files for random access is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file.
- Data can be inserted into a random-access file without destroying other data in the file.
- Data stored previously can be updated or deleted without rewriting the entire file.

Section 17.8 Creating a Random-Access File

- The `ostream` member function `write` outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream. Function `write` writes the data at the location in the file specified by the "put" file-position pointer.

- The `istream` member function `read` inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address. If the stream is associated with a file, function read inputs bytes at the location in the file specified by the "get" file-position pointer.
- Function `write` treats its first argument as a group of bytes by viewing the object in memory as a `const char *`, which is a pointer to a byte (remember that a `char` is one byte). Starting from that location, function `write` outputs the number of bytes specified by its second argument. The `istream` function read can subsequently be used to read the bytes back into memory.
- The `reinterpret_cast` operator converts a pointer of one type to an unrelated pointer type.
- A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points.
- A program that reads unformatted data must be compiled and executed on a system compatible with the program that wrote the data—different systems may represent internal data differently.
- Objects of class `string` do not have uniform size, rather they use dynamically allocated memory to accommodate strings of various lengths.
- The `string` member function `data` returns an array containing the characters of the `string`. This array is not guaranteed to be null terminated.
- The string member function `size` gets the length of a `string`.
- The file open mode `ios::binary` indicates that a file should be opened in binary mode.

Section 17.9 Writing Data Randomly to a Random-Access File

- Multiple file-open modes are combined by separating each open mode from the next with the bitwise inclusive OR operator (|).

Section 17.10 Reading from a Random-Access File Sequentially

- The `istream` function `read` inputs a specified number of bytes from the current position in the specified stream into an object.
- A function that receives an `ostream` parameter can receive any `ostream` object (such as `cout`) or any object of a derived class of `ostream` (such as an object of type `ofstream`) as an argument. This means that the same function can be used, for example, to perform output to the standard-output stream and to a file stream without writing separate functions.

Section 17.12 Overview of Object Serialization

- When object data members are output to a disk file, we lose the object's type information. We store only the values of the object's attributes, not type information, on the disk. If the program that reads this data knows the object type to which the data corresponds, the program can read the data into an object of that type.
- Several programming languages support object serialization. A so-called serialized object is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. A serialized object can be read from the file and deserialized.
- The open source Boost Libraries provide support for serializing objects in text, binary and extensible markup language (XML) formats (`www.boost.org/libs/serialization/doc/index.html`).

Terminology

binary digit 714
bit 714
byte 714
character set 714
characters 714
`close` member function of `ofstream` 720
database 716
database management system (DBMS) 716

data hierarchy 715
data persistence 714
decimal digit 714
deserialized 744
end-of-file marker 716
field 715
file 714
filename 718
file-open mode 718
file-position pointer 723
`fstream` 717
`<fstream>` `header file` 717
`ifstream` 718
instant-access application 727
`ios::app` file open mode 719
`ios::beg` seek starting point 723
`ios::binary` file open mode 732
`ios::cur` seek starting point 723
`ios::end` seek starting point 723
`ios::in` file open mode 721
`ios::out` file open mode 719
letters 714
object serialization 744
offset from the beginning of a file 723
`ofstream` 718
`open` member function of `ofstream` 719
random-access file 727
`read` member function of `istream` 728
record 715
record key 716
`reinterpret_cast` 729
secondary storage device 714
seek direction 723
`seekg` `istream` member function 723
`seekp` `ostream` member function 723
sequential file 716
serialized object 744
`size` function of `string` 730
special symbol 714
`tellg` `istream` member function 723
`tellp` `ostream` member function 723
transaction-processing system 727
truncate an existing file 719
Unicode® character set 714

Self-Review Exercises

17.1 Fill in the blanks in each of the following:

a) Ultimately, all data items processed by a computer are reduced to combinations of _____and _____.
b) The smallest data item a computer can process is called a(n) _____.
c) A(n) _____ is a group of related records.
d) Digits, letters and special symbols are referred to as _____.
e) A group of related files is called a(n) _____.
f) Member function _____ of the file streams `fstream`, `ifstream` and `ofstream` closes a file.
g) The `istream` member function _____ reads a character from the specified stream.
h) Member function _____ of the file streams `fstream`, `ifstream` and `ofstream` opens a file.
i) The `istream` member function _____ is normally used when reading data from a file in random-access applications.
j) Member functions _____ and _____ of `istream` and `ostream` set the file-position pointer to a specific location in an input or output stream, respectively.

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

a) Member function `read` cannot be used to read data from the input object `cin`.
b) You must create the `cin`, `cout`, `cerr` and `clog` objects explicitly.
c) A program must call function `close` explicitly to close a file associated with an `ifstream`, `ofstream` or `fstream` object.
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) The `ostream` member function `write` can write to standard-output stream `cout`.
f) Data in sequential files always is updated without overwriting nearby data.
g) Searching all records in a random-access file to find a specific record is unnecessary.

h) Records in random-access files must be of uniform length.
i) Member functions `seekp` and `seekg` must seek relative to the beginning of a file.

17.3 Assume that each of the following statements applies to the same program.
a) Write a statement that opens file `oldmast.dat` for input; use an `ifstream` object called `inOldMaster`.
b) Write a statement that opens file `trans.dat` for input; use an `ifstream` object called `inTransaction`.
c) Write a statement that opens file `newmast.dat` for output (and creation); use `ofstream` object `outNewMaster`.
d) Write a statement that reads a record from the file `oldmast.dat`. The record consists of integer `accountNumber`, string `name` and floating-point `currentBalance`; use `ifstream` object `inOldMaster`.
e) Write a statement that reads a record from the file `trans.dat`. The record consists of integer `accountNum` and floating-point `dollarAmount`; use `ifstream` object `inTransaction`.
f) Write a statement that writes a record to the file `newmast.dat`. The record consists of integer `accountNum`, string `name`, and floating-point `currentBalance`; use `ofstream` object `outNewMaster`.

17.4 Find the error(s) and show how to correct it (them) in each of the following.
a) File `payables.dat` referred to by `ofstream` object `outPayable` has not been opened.

```
outPayable << account << company << amount << endl;
```

b) The following statement should read a record from the file `payables.dat`. The `ifstream` object `inPayable` refers to this file, and `istream` object `inReceivable` refers to the file `receivables.dat`.

```
inReceivable >> account >> company >> amount;
```

c) The file `tools.dat` should be opened to add data to the file without discarding the current data.

```
ofstream outTools( "tools.dat", ios::out );
```

Answers to Self-Review Exercises

17.1 a) 1s, 0s. b) bit. c) file. d) characters. e) database. f) `close`. g) `get`. h) `open`. i) `read`. j) `seekg`, `seekp`.

17.2 a) False. Function `read` can read from any input stream object derived from `istream`.
b) False. These four streams are created automatically for you. The `<iostream>` header must be included in a file to use them. This header includes declarations for each stream object.
c) False. The files will be closed when destructors for `ifstream`, `ofstream` or `fstream` objects execute when the stream objects go out of scope or before program execution terminates, but it's a good programming practice to close all files explicitly with `close` once they're no longer needed.
d) False. Member function `seekp` or `seekg` can be used to reposition the "put" or "get" 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 normally are 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 position in the file.

17.3 a) `ifstream inOldMaster( "oldmast.dat", ios::in );`
b) `ifstream inTransaction( "trans.dat", ios::in );`
c) `ofstream outNewMaster( "newmast.dat", ios::out );`
d) `inOldMaster >> accountNumber >> name >> currentBalance;`
e) `inTransaction >> accountNum >> dollarAmount;`
f) `outNewMaster << accountNum << name << currentBalance;`

17.4 a) *Error:* The file `payables.dat` has not been opened before the attempt is made to output data to the stream.
Correction: Use `ostream` function `open` to open `payables.dat` for output.
b) *Error:* The incorrect `istream` object is being used to read a record from the file named `payables.dat`.
Correction: Use `istream` object `inPayable` to refer to `payables.dat`.
c) *Error:* The file's contents are discarded because the file is opened for output (`ios::out`).
Correction: To add data to the file, open the file either for updating (`ios::ate`) or for appending (`ios::app`).

Exercises

17.5 Fill in the blanks in each of the following:
a) Computers store large amounts of data 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) The vast majority of information stored in computer systems is stored in _____ files.
e) A group of related characters that conveys meaning is called a(n) _____.
f) The standard stream objects declared by header `<iostream>` are _____, _____, _____ and _____.
g) `ostream` member function _____ outputs a character to the specified stream.
h) `ostream` member function _____ is generally used to write data to a randomly accessed file.
i) `istream` member function _____ repositions the file-position pointer in a file.

17.6 State which of the following are *true* and which are *false*. If *false*, explain why.
a) The impressive functions performed by computers essentially involve the manipulation of zeros and ones.
b) People prefer to manipulate bits instead of characters and fields because bits are more compact.
c) People specify programs and data items as characters; computers then manipulate and process these characters as groups of zeros and ones.
d) A person's 5-digit zip code is an example of a numeric field.
e) A person's street address is generally considered to be an alphabetic field in computer applications.
f) Data items represented in computers form a data hierarchy in which data items become larger and more complex as we progress from fields to characters to bits, etc.
g) A record key identifies a record as belonging to a particular field.
h) Most organizations store all information in a single file to facilitate computer processing.
i) When a program creates a file, the file is automatically retained by the computer for future reference; i.e., files are said to be persistent.

17.7 ***(File Matching)*** Exercise 17.3 asked you to write a series of single statements. Actually, 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 appli-

cation system. In an accounts receivable system, for example, there is 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 (e.g., sales are made and cash payments arrive), they're entered into a file. At the end of each business period (a month for some companies, a week for others and a day in some cases), the file of transactions (called `trans.dat` in Exercise 17.3) is applied to the master file (called `oldmast.dat` in Exercise 17.3), thus updating each account's record of purchases and payments. During an updating run, 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 may 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 from Exercise 17.3 as a basis for writing 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 order by account number.

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

17.8 ***(File Matching Test Data)*** After writing the program of Exercise 17.7, write a simple program to create some test data for checking out the program. 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	Transaction amount
100	27.14
300	62.11
400	100.56
900	82.17

17.9 ***(File Matching Test)*** Run the program of Exercise 17.7, using the files of test data created in Exercise 17.8. Print the new master file. Check that the accounts have been updated correctly.

17.10 ***(File Matching Enhancement)*** It's common to have several transaction records with the same record key, because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts receivable file-matching program of Exercise 17.7 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of Exercise 17.8 to include the following additional transaction records:

Account number	Dollar amount
300	83.89
700	80.78
700	1.53

17.11 Write a series of statements that accomplish each of the following. Assume that we've defined class `Person` that contains the `private` data members

```
string lastName;
string firstName;
string age;
int id;
```

and `public` member functions

```
// accessor functions for id
void setId( int );
int getId() const;

// accessor functions for lastName
void setLastName( string );
string getLastName() const;

// accessor functions for firstName
void setFirstName( string );
string getFirstName() const;

// accessor functions for age
void setAge( string );
string getAge() const;
```

Also assume that any random-access files have been opened properly.

a) Initialize the file `nameage.dat` with 100 records that store values `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 that already contains information. If the record does not contain information, inform the user `"No info"`.
d) Delete a record that contains information by reinitializing that particular record.

17.12 ***(Hardware Inventory)*** You are the owner of a hardware store and need to keep an inventory that can tell you what different tools you have, how many of each you have on hand and the cost of each one. Write a program that initializes the random-access file `hardware.dat` to 100 empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update *any* information in the file. The tool identification number should be the record number. Use the following information to start your file:

Record #	Tool name	Quantity	Cost
3	Electric sander	7	57.98
17	Hammer	76	11.99

Record #	Tool name	Quantity	Cost
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

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

Digit	Letter
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
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 barber shop would be pleased to know that the shop's telephone number, 424-7288, corresponds to "HAIRCUT." A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to "PETCARE."

Write a 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.

17.14 Write a program that uses the `sizeof` operator to determine the sizes in bytes of the various data types on your computer system. Write the results to the file `datasize.dat`, so that you may print the results later. The results should be displayed in two-column format with the type name in the left column and the size of the type in right column, as in:

```
char                    1
unsigned char           1
short int               2
unsigned short int      2
int                     4
unsigned int            4
long int                4
unsigned long int       4
float                   4
double                  8
long double            10
```

[*Note:* The sizes of the built-in data types on your computer might differ from those listed above.]

Making a Difference

17.15 ***(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 McAfee® (`www.mcafee.com/us/threat_center/anti_phishing/phishing_top10.html`), Security Extra (`www.securityextra.com/`), `www.snopes.com` and other websites to find lists of the top phishing scams. Also check out the Anti-Phishing Working Group (`www.antiphishing.org/`), and the FBI's Cyber Investigations website (`www.fbi.gov/cyberinvest/cyberhome.htm`), 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?