Topic 7
Stream I/O & File Processing
Lecture 7b (Chapter 14)

CSCI 140: C++ Language & Objects

Prof. Dominick Atanasio

Book: C++: How to Program 10 ed.

Agenda

- Introduction
- Files and Streams
- Creating a Sequential File
- Reading Data from a Sequential File
- Updating Sequential Files
- Random-Access Files
- Creating a Random-Access File
- Writing Data Randomly to a Random-Access File
- Reading from a Random-Access File Sequentially

14.1 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 21.

14.2 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 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 13, 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.

14.2 Files and Streams (cont.)

- File-Processing Class Templates
- To perform file processing in C++, headers <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.

14.2 Files and Streams (cont.)

- 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.

14.2 Files and Streams (cont.)

- 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 13) also can be applied to file streams.
- Figure 14.2 summarizes the inheritance relationships of the I/O classes that we've discussed to this point.

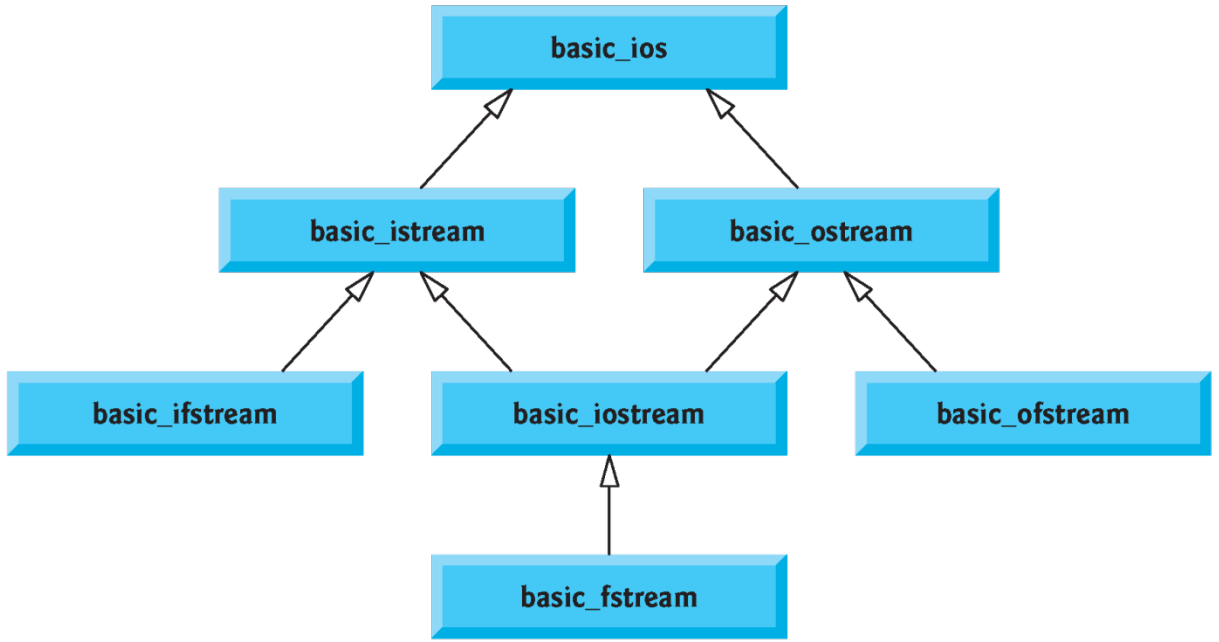

Fig. 14.2 | Portion of stream I/O template hierarchy.

14.3 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.
- You must structure files to meet the application’s requirements.
- Figure 14.3 creates a sequential file that might be used in an accounts-receivable system to help manage the money owed to 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.
- This program assumes the user enters the records in account number order.
  - In a comprehensive accounts receivable system, a sorting capability would be provided to eliminate this restriction.

```
1  // Fig. 14.3: Fig14_03.cpp
2  // Create a sequential file.
3  #include <iostream>
4  #include <string>
5  #include <fstream> // contains file stream processing types
6  #include <cstdlib> // exit function prototype
7  using namespace std;
8
9  int main()
10 {
11    // ofstream constructor opens file
12    ofstream outClientFile( "clients.txt", 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( EXIT_FAILURE );
19    } // end if
20
21    cout << "Enter the account, name, and balance." << endl
22       << "Enter end-of-file to end input.\n? ";
23
```

Fig. 14.3 | Create a sequential file. (Part 1 of 2.)

```
24      int account; // the account number
25      string name; // the account owner's name
26      double balance; // the account 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. 14.3 | Create a sequential file. (Part 2 of 2.)

14.3 Creating a Sequential File (cont.)

- Opening a File
- Figure 14.3 writes data to a file so we open the file output by creating an ofstream object.
- 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 (the default) 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).
- Since ios::out is the default, the second constructor argument in line 12 is not required.
- 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.
- Prior to C++11, the filename was specified as a pointer-based string—as of C++11, it can also be specified as a string object.

14.3 Creating a Sequential File (cont.)

- The ofstream constructor opens the file—this establishes a “line of communication” with the file.
- By default, ofstream objects are opened for output, so the open mode is not required in the constructor call.
- Figure 14.4 lists the file-open modes.

Modes

- ios::in Open for input operations.
- ios::out Open for output operations.
- ios::binary Open in binary mode.
- ios::ate Set the initial position at the end of the file.
  If this flag is not set, the initial position is the beginning of the file.
- ios::app All output operations are performed at the end of the file, appending the content to the current content of the file.
- ios::trunc If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.
- All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:
- ofstream myfile;
- myfile.open ("example.bin", ios::out | ios::app | ios::binary);

14.3 Creating a Sequential File (cont.)

- Opening a File via the open Member Function
- You can create an ofstream object without opening a specific file—in this case, a file can be attached to the object later.
- For example, the statement
  - ofstream outClientFile;
- creates an ofstream object that's not yet associated with a file.
- The ofstream member function open opens a file and attaches it to an existing ofstream object as follows:
  - outClientFile.open("clients.dat", ios::out);

14.3 Creating a Sequential File (cont.)

- Testing Whether a File Was Opened Successfully
- 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 nonexistent file for reading,
  - attempting to open a file for reading or writing with-out permission, and
  - opening a file for writing when no disk space is available.

14.3 Creating a Sequential File (cont.)

- Function exit terminates a program.
  - The argument to exit is returned to the environment from which the program was invoked.
  - Passing EXIT_SUCCESS (also defined in <cstdlib>) to exit indicates that the program terminated normally; passing any other value (in this case EXIT_FAILURE) indicates that the program terminated due to an error.

14.3 Creating a Sequential File (cont.)

- The Overloaded void * Operator
- 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 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.

14.3 Creating a Sequential File (cont.)

- The operator void * function can be used to test an input object for end-of-file, but you can also call member function eof on the input object.
- Processing Data
- Figure 14.5 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*)

Fig. 14.5 | End-of-file key combinations.

14.3 Creating a Sequential File (cont.)

- 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.
- Line 31 writes a set of data to the file clients.txt, 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 14.4).
- The file created in Fig. 14.3 is simply a text file, so it can be viewed by any text editor.

14.3 Creating a Sequential File (cont.)

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

14.4 Reading Data from a Sequential File

- Files store data so it may be retrieved for processing when needed.
- In this section, we discuss how to read data sequentially from a file.
- Figure 14.6 reads records from the clients.txt file that we created using the program of Fig. 14.3 and displays the contents of these records.

14.4 Reading Data from a Sequential File (cont.)

- 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.txt 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.

```
1  // Fig. 14.6: Fig14_06.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.txt", ios::in );
16
17    // exit program if ifstream could not open file
18    if ( !inClientFile )
19    {
20       cerr << "File could not be opened" << endl;
21       exit( EXIT_FAILURE );
22    } // end if
23
```

Fig. 14.6 | Reading and printing a sequential file. (Part 1 of 3.)

```
24       int account; // the account number
25       string name; // the account owner's name
26       double balance; // the account 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
```

Fig. 14.6 | Reading and printing a sequential file. (Part 2 of 3.)

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

Fig. 14.6 | Reading and printing a sequential file. (Part 3 of 3.)

14.4 Reading Data from a Sequential File (cont.)

- Opening a File for Input
- Objects of class ifstream are opened for input by default.
- We could have used the statement
  - ifstream inClientFile("clients.txt");
- 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.

14.4 Reading Data from a Sequential File (cont.)

- Reading from the File
- Line 32 reads a set of data (i.e., a record) from the file.
- Each time line 32 executes, it reads another record from the file into the variables account, name and balance.
- 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.

14.4 Reading Data from a Sequential File (cont.)

- File Position Pointers
- 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 functions for repositioning the file-position pointer (the byte number of the next byte in the file to be read or written).
  - seekg (“seek get”) for istream
  - seekp (“seek put”) for ostream

14.4 Reading Data from a Sequential File (cont.)

- 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.

14.4 Reading Data from a Sequential File (cont.)

- 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).

14.4 Reading Data from a Sequential File (cont.)

- 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.

14.4 Reading Data from a Sequential File (cont.)

- Member functions tellg and tellp are provided to return the current locations of the get and put pointers, respectively.

14.4 Reading Data from a Sequential File (cont.)

- Credit Inquiry Program
- Figure 14.7 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)

```
1  // Fig. 14.7: Fig14_07.cpp
2  // Credit inquiry program.
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
11 int getRequest();
12 bool shouldDisplay( int, double );
13 void outputLine( int, const string &, double );
14
15 int main()
16 {
17    // ifstream constructor opens the file
18    ifstream inClientFile( "clients.txt", ios::in );
19
```

Fig. 14.7 | Credit inquiry program. (Part 1 of 8.)

```
20      // exit program if ifstream could not open file
21      if ( !inClientFile )
22      {
23         cerr << "File could not be opened" << endl;
24         exit( EXIT_FAILURE );
25      } // end if
26
27      int account; // the account number
28      string name; // the account owner's name
29      double balance; // the account balance
30
31      // get user's request (e.g., zero, credit or debit balance)
32      int request = getRequest();
33
```

Fig. 14.7 | Credit inquiry program. (Part 2 of 8.)

```
34      // process user's request
35      while ( request != END )
36      {
37         switch ( request )
38         {
39            case ZERO_BALANCE:
40               cout << "\nAccounts with zero balances:\n";
41               break;
42            case CREDIT_BALANCE:
43               cout << "\nAccounts with credit balances:\n";
44               break;
45            case DEBIT_BALANCE:
46               cout << "\nAccounts with debit balances:\n";
47               break;
48         } // end switch
49
50         // read account, name and balance from file
51         inClientFile >> account >> name >> balance;
52
```

Fig. 14.7 | Credit inquiry program. (Part 3 of 8.)

```
53          // display file contents (until eof)
54          while ( !inClientFile.eof() )
55          {
56             // display record
57             if ( shouldDisplay( request, balance ) )
58                outputLine( account, name, balance );
59
60             // read account, name and balance from file
61             inClientFile >> account >> name >> balance;
62          } // end inner while
63
64          inClientFile.clear(); // reset eof for next input
65          inClientFile.seekg( 0 ); // reposition to beginning of file
66          request = getRequest(); // get additional request from user
67       } // end outer while
68
69       cout << "End of run." << endl;
70    } // end main
71
72    // obtain request from user
73    int getRequest()
74    {
75       int request; // request from user
76
```

Fig. 14.7 | Credit inquiry program. (Part 4 of 8.)

```
77       // display request options
78       cout << "\nEnter request" << endl
79          << " 1 - List accounts with zero balances" << endl
80          << " 2 - List accounts with credit balances" << endl
81          << " 3 - List accounts with debit balances" << endl
82          << " 4 - End of run" << fixed << showpoint;
83
84       do // input user request
85       {
86          cout << "\n? ";
87          cin >> request;
88       } while ( request < ZERO_BALANCE && request > END );
89
90       return request;
91    } // end function getRequest
92
93    // determine whether to display given record
94    bool shouldDisplay( int type, double balance )
95    {
96       // determine whether to display zero balances
97       if ( type == ZERO_BALANCE && balance == 0 )
98          return true;
99
```

Fig. 14.7 | Credit inquiry program. (Part 5 of 8.)

```
100     // determine whether to display credit balances
101     if ( type == CREDIT_BALANCE && balance < 0 )
102        return true;
103
104     // determine whether to display debit balances
105     if ( type == DEBIT_BALANCE && balance > 0 )
106        return true;
107
108     return false;
109  } // end function shouldDisplay
110
111  // display single record from file
112  void outputLine( int account, const string &name, double balance )
113  {
114     cout << left << setw( 10 ) << account << setw( 13 ) << name
115        << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
116  } // end function outputLine
```

Fig. 14.7 | Credit inquiry program. (Part 6 of 8.)

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

Accounts with zero balances:
300       White           0.00

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
```

Fig. 14.7 | Credit inquiry program. (Part 7 of 8.)

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 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. 14.7 | Credit inquiry program. (Part 8 of 8.)

14.5 Updating Sequential Files

- Data that is formatted and written to a sequential file as shown in Section 14.3 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.

14.5 Updating Sequential Files (cont.)

- 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 32-bit machines and eight bytes on 64-bit machines).
  - However, these integers become different-sized fields, depending on their actual values, when output as formatted text (character sequences).
  - Therefore, the formatted input/output model usually is not used to update records in place.

14.5 Updating Sequential Files (cont.)

- 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.
- Then the old file could be deleted and the new file renamed.
- 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.

14.6 Random-Access Files

- 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.

14.6 Random-Access Files (cont.)

- Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.
- C++ does not impose structure on a file. So, the application that wants to use random-access files must create them.
- 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 quickly 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.
- Figure 14.8 illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long).

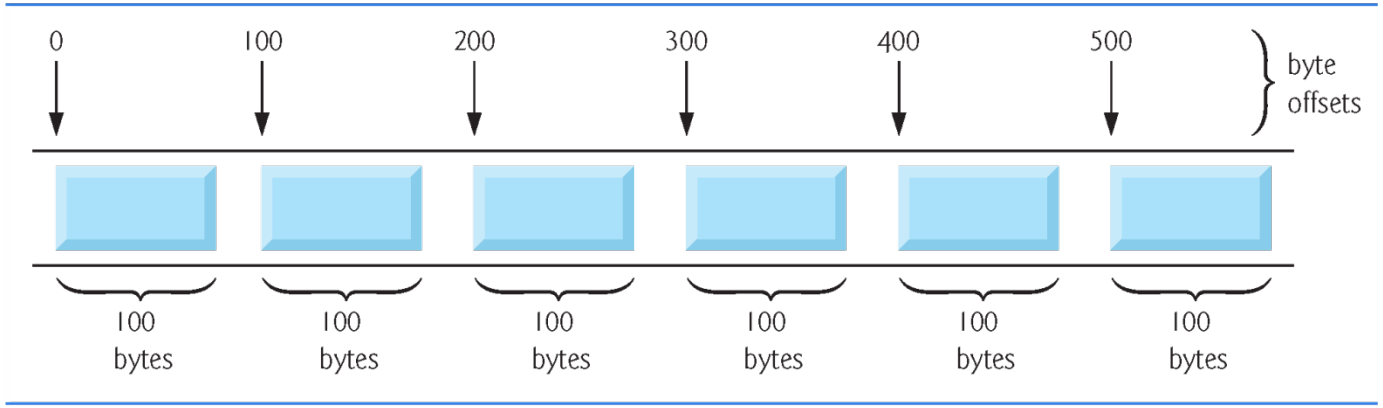

Fig. 14.8 | C++ view of a random-access file.

14.6 Random-Access Files (cont.)

- 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.

14.7 Creating a Random-Access File

- The ostream member function write outputs to the specified stream a fixed number of bytes, beginning at a specific location in memory.
- 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.

14.7 Creating a Random-Access File (cont.)

- Writing Bytes with ostream Member Function write
- Outputting a four-byte integer as text could print as few digits as one or as many as 11 (10 digits plus a sign, each requiring a single byte of storage)
- The following statement always writes the binary version of the integer's four bytes (on a machine with four-byte integers):
  - outFile.write(reinterpret_cast<
    const char * >(&number), sizeof(number));
- 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.
- istream function read can be used to read the four bytes back into an integer variable.

14.7 Creating a Random-Access File (cont.)

- Converting Between Pointer Types with the reinterpret_cast Operator
- Most pointers that we pass to function *write* as the first argument are not of type const char *.
- Must convert the pointers to those objects to type const char *; otherwise, the compiler will not compile calls to function 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 inconsistent.
- A reinterpret_cast is performed at compile time and does not change the value of the object to which its operand points.

14.7 Creating a Random-Access File (cont.)

- In Fig. 14.11, 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 typically write one object of a class at a time, as we show in the following examples.

14.7 Creating a Random-Access File (cont.)

- 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 few sections create this credit-processing program.

14.7 Creating a Random-Access File (cont.)

- Figure 14.11 illustrates opening a random-access file, defining the record format using an object of class ClientData (Figs. 14.9–14.10) 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 the account number 0, empty last and first name strings and the balance 0.0.
- Each record is initialized with the space in which the account data will be stored.

14.7 Creating a Random-Access File (cont.)

- 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. 14.9, lines 32–33).
- Member functions setLastName (Fig. 14.10, lines 35–42) and setFirstName (Fig. 14.10, lines 51–58) each copy the characters of a string object into the corresponding char array.

14.7 Creating a Random-Access File (cont.)

- Consider function setLastName.
- Line 38 invokes string member function size to get the length of lastNameString.
- Line 39 ensures that length is fewer than 15 characters, then line 40 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. 14.9: ClientData.h
2  // Class ClientData definition used in Fig. 14.11-Fig. 14.14.
3  #ifndef CLIENTDATA_H
4  #define CLIENTDATA_H
5
6  #include <string>
7
8  class ClientData
9  {
10 public:
11    // default ClientData constructor
12    ClientData( int = 0, const std::string & = "",
13       const std::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( const std::string & );
21    std::string getLastName() const;
22
```

Fig. 14.9 | `ClientData` class header. (Part 1 of 2.)

```
23      // accessor functions for firstName
24      void setFirstName( const std::string & );
25      std::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. 14.9 | `ClientData` class header. (Part 2 of 2.)

```
1  // Fig. 14.10: ClientData.cpp
2  // Class ClientData stores customer's credit information.
3  #include <string>
4  #include "ClientData.h"
5  using namespace std;
6
7  // default ClientData constructor
8  ClientData::ClientData( int accountNumberValue, const string &lastName,
9     const string &firstName, double balanceValue )
10    : accountNumber( accountNumberValue ), balance( balanceValue )
11 {
12    setLastName( lastNameValue );
13    setFirstName( firstNameValue );
14 } // end ClientData constructor
15
16 // get account-number value
17 int ClientData::getAccountNumber() const
18 {
19    return accountNumber;
20 } // end function getAccountNumber
21
```

Fig. 14.10 | `ClientData` class represents a customer's credit information. (Part 1 of 4.)

```
22  // set account-number value
23  void ClientData::setAccountNumber( int accountNumberValue )
24  {
25     accountNumber = accountNumberValue; // should validate
26  } // end function setAccountNumber
27
28  // get last-name value
29  string ClientData::getLastName() const
30  {
31     return lastName;
32  } // end function getLastName
33
34  // set last-name value
35  void ClientData::setLastName( const string &lastNameString )
36  {
37     // copy at most 15 characters from string to lastName
38     int length = lastNameString.size();
39     length = ( length < 15 ? length : 14 );
40     lastNameString.copy( lastName, length );
41     lastName[ length ] = '\0'; // append null character to lastName
42  } // end function setLastName
43
```

Fig. 14.10 | `ClientData` class represents a customer's credit information. (Part 2 of 4.)

```
44  // get first-name value
45  string ClientData::getFirstName() const
46  {
47     return firstName;
48  } // end function getFirstName
49
50  // set first-name value
51  void ClientData::setFirstName( const string &firstNameString )
52  {
53     // copy at most 10 characters from string to firstName
54     int length = firstNameString.size();
55     length = ( length < 10 ? length : 9 );
56     firstNameString.copy( firstName, length );
57     firstName[ length ] = '\0'; // append null character to firstName
58  } // end function setFirstName
59
60  // get balance value
61  double ClientData::getBalance() const
62  {
63     return balance;
64  } // end function getBalance
65
```

Fig. 14.10 | `ClientData` class represents a customer's credit information. (Part 3 of 4.)

```
66   // set balance value
67   void ClientData::setBalance( double balanceValue )
68   {
69      balance = balanceValue;
70   } // end function setBalance
```

Fig. 14.10 | `ClientData` class represents a customer's credit information. (Part 4 of 4.)

14.7 Creating a Random-Access File (cont.)

- Opening a File for Output in Binary Mode
- In Fig. 14.11, 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.
- Multiple file-open modes are combined by separating each open mode from the next with the | operator, which is known as the bitwise inclusive OR operator.

14.7 Creating a Random-Access File (cont.)

- Lines 24–25 cause the blankClient (which was constructed with default arguments at line 20) to be written to the credit.dat file associated with ofstream object outCredit.
- Operator sizeof returns the size in bytes of the object contained in parentheses (see Chapter 8).

14.7 Creating a Random-Access File (cont.)

- The first argument to function write at line 24 must be of type const char *.
- However, the data type of &blankClient is Client-Data *.
- 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. 14.11: Fig14_11.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( EXIT_FAILURE );
18    } // end if
19
20    ClientData blankClient; // constructor zeros out each data member
21
```

Fig. 14.11 | 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. 14.11 | Creating a random-access file with 100 blank records sequentially. (Part 2 of 2.)

14.8 Writing Data Randomly to a Random-Access File

- Figure 14.12 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 ClientData.h defined in Fig. 14.9, so the program can use ClientData objects.

```
1  // Fig. 14.12: Fig14_12.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( EXIT_FAILURE );
23    } // end if
24
```

Fig. 14.12 | Writing to a random-access file. (Part 1 of 4.)

```
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      {
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 ) );
```

Fig. 14.12 | Writing to a random-access file. (Part 2 of 4.)

```
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
```

Fig. 14.12 | Writing to a random-access file. (Part 3 of 4.)

```
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. 14.12 | Writing to a random-access file. (Part 4 of 4.)

14.8 Writing Data Randomly to a Random-Access File (cont.)

- Opening a File for Input and Output in Binary Mode
- 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.
- 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. 14.11, rather than creating the file from scratch.

14.8 Writing Data Randomly to a Random-Access File (cont.)

- Positioning the File Position Pointer
- 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.

14.9 Reading from a Random-Access File Sequentially

- In this section, we develop a program that reads a file sequentially and prints only those records that contain data.
- The istream function read inputs a specified number of bytes from the current position in the specified stream into an object.
- For example, lines 31–32 from Fig. 14.13 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. 14.13: Fig14_13.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( EXIT_FAILURE );
21    } // end if
22
```

Fig. 14.13 | Reading a random-access file sequentially. (Part 1 of 3.)

```
23       // output column heads
24       cout << left << setw( 10 ) << "Account" << setw( 16 )
25          << "Last Name" << setw( 11 ) << "First Name" << left
26          << setw( 10 ) << right << "Balance" << endl;
27
28       ClientData client; // create record
29
30       // read first record from file
31       inCredit.read( reinterpret_cast< char * >( &client ),
32          sizeof( ClientData ) );
33
34       // read all records from file
35       while ( inCredit && !inCredit.eof() )
36       {
37          // display record
38          if ( client.getAccountNumber() != 0 )
39             outputLine( cout, client );
40
41          // read next from file
42          inCredit.read( reinterpret_cast< char * >( &client ),
43             sizeof( ClientData ) );
44       } // end while
45    } // end main
46
```

Fig. 14.13 | Reading a random-access file sequentially. (Part 2 of 3.)

```
47  // display single record
48  void outputLine( ostream &output, const ClientData &record )
49  {
50     output << left << setw( 10 ) << record.getAccountNumber()
51        << setw( 16 ) << record.getLastName()
52        << setw( 11 ) << record.getFirstName()
53        << setw( 10 ) << setprecision( 2 ) << right << fixed
54        << showpoint << record.getBalance() << endl;
55  } // 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. 14.13 | Reading a random-access file sequentially. (Part 3 of 3.)

14.9 Reading from a Random-Access File Sequentially (cont.)

- Figure 14.13 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 35 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 48–55), 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.

14.9 Reading from a Random-Access File Sequentially (cont.)

- 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.
- 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, declining storage prices has made this less of an issue.

14.10 Case Study: A Transaction-Processing Program

- We now present a substantial transaction-processing program (Fig. 14.14) using a random-access file to achieve “instant-access” processing.
- The program updates existing bank 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. 14.11 has been executed to create the file credit.dat and that the program of Fig. 14.12 has been executed to insert the initial data.

```
1  // Fig. 14.14: Fig14_14.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
```

Fig. 14.14 | Bank account program. (Part 1 of 12.)

```
22  int main()
23  {
24     // open file for reading and writing
25     fstream inOutCredit( "credit.dat", ios::in | ios::out | ios::binary );
26
27     // exit program if fstream cannot open file
28     if ( !inOutCredit )
29     {
30        cerr << "File could not be opened." << endl;
31        exit ( EXIT_FAILURE );
32     } // end if
33
```

Fig. 14.14 | Bank account program. (Part 2 of 12.)

```
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
```

Fig. 14.14 | Bank account program. (Part 3 of 12.)

```
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. 14.14 | Bank account program. (Part 4 of 12.)

```
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( EXIT_FAILURE );
90     } // end if
91
92     // output column heads
93     outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
94        << "Last Name" << setw( 11 ) << "First Name" << right
95        << setw( 10 ) << "Balance" << endl;
96
97     // set file-position pointer to beginning of readFromFile
98     readFromFile.seekg( 0 );
99
```

Fig. 14.14 | Bank account program. (Part 5 of 12.)

```
100     // read first record from record file
101     ClientData client;
102     readFromFile.read( reinterpret_cast< char * >( &client ),
103        sizeof( ClientData ) );
104
105     // copy all records from record file into text file
106     while ( !readFromFile.eof() )
107     {
108        // write single record to text file
109        if ( client.getAccountNumber() != 0 ) // skip empty records
110           outputLine( outPrintFile, client );
111
112        // read next record from record file
113        readFromFile.read( reinterpret_cast< char * >( &client ),
114           sizeof( ClientData ) );
115     } // end while
116  } // end function createTextFile
117
118  // update balance in record
119  void updateRecord( fstream &updateFile )
120  {
121     // obtain number of account to update
122     int accountNumber = getAccount( "Enter account to update" );
123
```

Fig. 14.14 | Bank account program. (Part 6 of 12.)

```
124     // move file-position pointer to correct record in file
125     updateFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
126
127     // read first record from file
128     ClientData client;
129     updateFile.read( reinterpret_cast< char * >( &client ),
130        sizeof( ClientData ) );
131
132     // update record
133     if ( client.getAccountNumber() != 0 )
134     {
135        outputLine( cout, client ); // display the record
136
137        // request user to specify transaction
138        cout << "\nEnter charge (+) or payment (-): ";
139        double transaction; // charge or payment
140        cin >> transaction;
141
142        // update record balance
143        double oldBalance = client.getBalance();
144        client.setBalance( oldBalance + transaction );
145        outputLine( cout, client ); // display the record
146
```

Fig. 14.14 | Bank account program. (Part 7 of 12.)

```
147         // move file-position pointer to correct record in file
148         updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
149
150         // write updated record over old record in file
151         updateFile.write( reinterpret_cast< const char * >( &client ),
152            sizeof( ClientData ) );
153      } // end if
154      else // display error if account does not exist
155         cerr << "Account #" << accountNumber
156            << " has no information." << endl;
157   } // end function updateRecord
158
159   // create and insert record
160   void newRecord( fstream &insertInFile )
161   {
162      // obtain number of account to create
163      int accountNumber = getAccount( "Enter new account number" );
164
165      // move file-position pointer to correct record in file
166      insertInFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
167
```

Fig. 14.14 | Bank account program. (Part 8 of 12.)

```
168     // read record from file
169     ClientData client;
170     insertInFile.read( reinterpret_cast< char * >( &client ),
171        sizeof( ClientData ) );
172
173     // create record, if record does not previously exist
174     if ( client.getAccountNumber() == 0 )
175     {
176        string lastName;
177        string firstName;
178        double balance;
179
180        // user enters last name, first name and balance
181        cout << "Enter lastname, firstname, balance\n? ";
182        cin >> setw( 15 ) >> lastName;
183        cin >> setw( 10 ) >> firstName;
184        cin >> balance;
185
186        // use values to populate account values
187        client.setLastName( lastName );
188        client.setFirstName( firstName );
189        client.setBalance( balance );
190        client.setAccountNumber( accountNumber );
191
```

Fig. 14.14 | Bank account program. (Part 9 of 12.)

```
192         // move file-position pointer to correct record in file
193         insertInFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
194
195         // insert record in file
196         insertInFile.write( reinterpret_cast< const char * >( &client ),
197            sizeof( ClientData ) );
198      } // end if
199      else // display error if account already exists
200         cerr << "Account #" << accountNumber
201            << " already contains information." << endl;
202   } // end function newRecord
203
204   // delete an existing record
205   void deleteRecord( fstream &deleteFromFile )
206   {
207      // obtain number of account to delete
208      int accountNumber = getAccount( "Enter account to delete" );
209
210      // move file-position pointer to correct record in file
211      deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
212
```

Fig. 14.14 | Bank account program. (Part 10 of 12.)

```
213     // read record from file
214     ClientData client;
215     deleteFromFile.read( reinterpret_cast< char * >( &client ),
216        sizeof( ClientData ) );
217
218     // delete record, if record exists in file
219     if ( client.getAccountNumber() != 0 )
220     {
221        ClientData blankClient; // create blank record
222
223        // move file-position pointer to correct record in file
224        deleteFromFile.seekp( ( accountNumber - 1 ) *
225           sizeof( ClientData ) );
226
227        // replace existing record with blank record
228        deleteFromFile.write(
229           reinterpret_cast< const char * >( &blankClient ),
230           sizeof( ClientData ) );
231
232        cout << "Account #" << accountNumber << " deleted.\n";
233     } // end if
234     else // display error if record does not exist
235        cerr << "Account #" << accountNumber << " is empty.\n";
236  } // end deleteRecord
```

Fig. 14.14 | Bank account program. (Part 11 of 12.)

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

Fig. 14.14 | Bank account program. (Part 12 of 12.)

14.10 Case Study: A Transaction-Processing Program (cont.)

- The program has five options (Option 5 is for terminating the program).
- Option 1 calls function createTextFile to store a format-ted list of all the account information in a text file called print.txt that may be printed.
- Function createTextFile (lines 80–116) 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 102–103) and uses the sequential-file-access techniques of Fig. 14.13 to input data from credit.dat.
- Function outputLine, discussed in Section 14.9, is used to output the data to file print.txt.
- Note that function createTextFile uses istream member function seekg (line 98) to ensure that the file-position pointer is at the beginning of the file.

14.10 Case Study: A Transaction-Processing Program (cont.)

- Option 2 calls updateRecord (lines 119–157) to update an account.
- This function updates only an existing record, so the function first determines whether the specified record is empty.
- If the record contains information, line 135 displays the record, using function outputLine, line 140 inputs the transaction amount and lines 143–152 calculate the new balance and rewrite the record to the file.
- Option 3 calls function newRecord (lines 160–202) 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 200–201).
- This function adds a new account in the same manner as the program of Fig. 14.12.

14.10 Case Study: A Transaction-Processing Program (cont.)

- Option 4 calls function deleteRecord (lines 205–236) to delete a record from the file.
- Line 208 prompts the user to enter the account number.
- Only an existing record may be deleted, so, if the specified account is empty, line 235 displays an error message.
- If the account exists, lines 221–230 reinitialize that account by copying an empty record (blankClient) to the file.
- Line 232 displays a message to inform the user that the record has been deleted.

14.11 Object Serialization

- This chapter and Chapter 13 introduced the object-oriented style of input/output.
- 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.

14.11 Object Serialization (cont.)

- 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 12).
- 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.

14.11 Object Serialization (cont.)

- 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).