

# **Random Access File**

## **I / O in C++**

Sukhbir Tatla

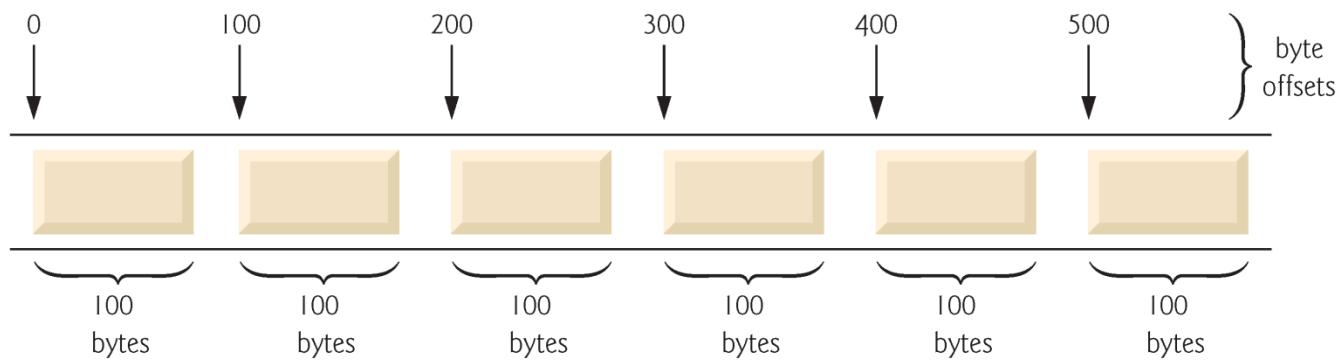
[sukhbir.tatla@senecacollege.ca](mailto:sukhbir.tatla@senecacollege.ca)

# 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.

# 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 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 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).



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



# 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, you'll learn 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.

# 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.

# Random-Access Files (cont.)

- ▶ 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.

# Random-Access Files (cont.)

- ▶ 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 incompatible.
- ▶ A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points.

# Random-Access Files (cont.)

- ▶ 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.

---

```
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
```

---

**Fig. 17.10** | ClientData class header file. (Part I of 2.)



---

```
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. (Part 2 of 2.)



---

```
1 // Fig. 17.11: 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,
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
```

---

**Fig. 17.11** | ClientData class represents a customer's credit information. (Part I of 4.)



---

```
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
```

---

**Fig. 17.11** | ClientData class represents a customer's credit information. (Part 2 of 4.)



---

```
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 );
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
```

---

**Fig. 17.11** | ClientData class represents a customer's credit information. (Part 3 of 4.)



---

```
67  s
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 4 of 4.)



---

```
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.)



---

```
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
```

---

**Fig. 17.13** | Writing to a random-access file. (Part I of 4.)



---

```
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 {
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
```

---

**Fig. 17.13** | Writing to a random-access file. (Part 2 of 4.)



---

```
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
```

---

**Fig. 17.13** | Writing to a random-access file. (Part 3 of 4.)



```
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 4 of 4.)

# Reading from a Random-Access File Sequentially

- ▶ The next 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 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 I of 3.)



---

```
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
```

---

**Fig. 17.14** | Reading a random-access file sequentially. (Part 2 of 3.)



```
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 3 of 3.)

# 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.
- ▶ Compared to insertion sort, 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.

# Practice

- ▶ REALITY CHECK (Week 12)
- ▶ Question #1 (Word problem)

# 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 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. 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 {
```

---

**Fig. 17.15** | Bank account program. (Part I of 12.)



---

```
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 ( 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;
```

---

**Fig. 17.15** | Bank account program. (Part 2 of 12.)



---

```
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
```

---

**Fig. 17.15** | Bank account program. (Part 3 of 12.)



---

```
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
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 }
```

---

**Fig. 17.15** | Bank account program. (Part 4 of 12.)



---

```
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 {
```

---

**Fig. 17.15** | Bank account program. (Part 5 of 12.)



---

```
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 6 of 12.)



---

```
I31 // update record
I32 if ( client.getAccountNumber() != 0 )
I33 {
I34     outputLine( cout, client ); // display the record
I35
I36     // request user to specify transaction
I37     cout << "\nEnter charge (+) or payment (-): ";
I38     double transaction; // charge or payment
I39     cin >> transaction;
I40
I41     // update record balance
I42     double oldBalance = client.getBalance();
I43     client.setBalance( oldBalance + transaction );
I44     outputLine( cout, client ); // display the record
I45
I46     // move file-position pointer to correct record in file
I47     updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
I48
I49     // write updated record over old record in file
I50     updateFile.write( reinterpret_cast< const char * >( &client ),
I51                     sizeof( ClientData ) );
I52 } // end if
```

---

**Fig. 17.15** | Bank account program. (Part 7 of 12.)



---

```
153     else
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     {
```

---

**Fig. 17.15** | Bank account program. (Part 8 of 12.)



---

```
I75     string lastName;
I76     string firstName;
I77     double balance;
I78
I79     // user enters last name, first name and balance
I80     cout << "Enter lastname, firstname, balance\n? ";
I81     cin >> setw( 15 ) >> lastName;
I82     cin >> setw( 10 ) >> firstName;
I83     cin >> balance;
I84
I85     // use values to populate account values
I86     client.setLastName( lastName );
I87     client.setFirstName( firstName );
I88     client.setBalance( balance );
I89     client.setAccountNumber( accountNumber );
I90
I91     // move file-position pointer to correct record in file
I92     insertInFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
I93
I94     // insert record in file
I95     insertInFile.write( reinterpret_cast< const char * >( &client ),
I96                         sizeof( ClientData ) );
I97 } // end if
```

---

**Fig. 17.15** | Bank account program. (Part 9 of 12.)



---

```
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
```

---

**Fig. 17.15** | Bank account program. (Part 10 of 12.)



---

```
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
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
```

---

**Fig. 17.15** | Bank account program. (Part 11 of 12.)



---

```
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 12 of 12.)

