

# Complementos de Programação de Computadores – Aula 6c

## Ficheiros

Mestrado Integrado em Electrónica Industrial e Computadores

**Luís Paulo Reis**

[lpreis@dsi.uminho.pt](mailto:lpreis@dsi.uminho.pt)

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia,  
Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2012)



Programação - MIEEIC | Luis Paulo Reis | Universidade do Minho - Escola de Engenharia | 1

## Outline

- 17.1 Introduction
- 17.2 Files and Streams
- 17.3 Creating a Sequential File
- 17.4 Reading Data from a Sequential File
- 17.5 Updating Sequential Files
- 17.6 Random-Access Files
- 17.7 Creating a Random-Access File
- 17.8 Writing Data Randomly to a Random-Access File
- 17.9 Reading from a Random-Access File Sequentially
- 17.10 Case Study: A Transaction-Processing Program
- 17.11 Object Serialization
- 17.12 Wrap-Up

## 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, Class **string** and String Stream Processing

## 17.2 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
- 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

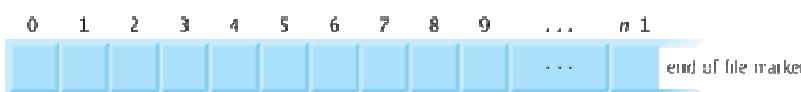


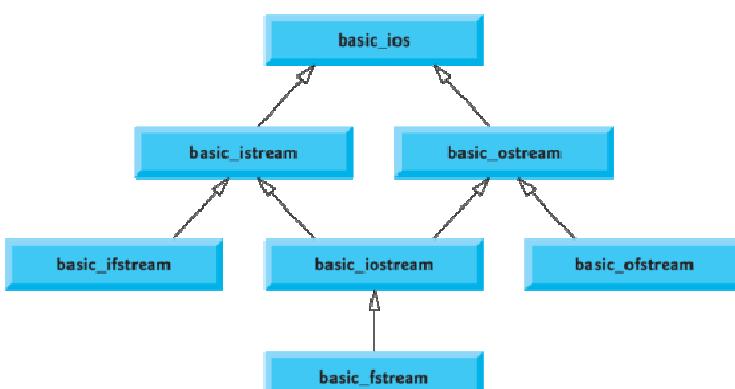
Fig. 17.1 | C++'s simple view of a file of  $n$  bytes.

## 17.2 Files and Streams (cont.)

- 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
- The `<fstream>` library provides `typedef` aliases for these template specializations
- `typedef ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file
- `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
- Files are opened by creating objects of these stream template specializations

## 17.2 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 15) also can be applied to file streams.
- Figure 17.2 summarizes the inheritance relationships of the I/O classes that we've discussed to this point.



## 17.3 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 17.3 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. 17.3: Fig17_03.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.txt", ios::out );
13 }
```

Fig. 17.3 | Creating a sequential file. (Part 1 of 3.)

```

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
  
```

**Fig. 17.3 | Creating a sequential file. (Part 2 of 3.)**

```

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
? AZ
  
```

**Fig. 17.3 | Creating a sequential file. (Part 3 of 3.)**

## 17.3 Creating a Sequential File (cont.)

- In Fig. 17.3, 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
- 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

## 17.3 Creating a Sequential File (cont.)

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

**Fig. 17.4** | File open modes.

## 17.3 Creating a Sequential File (cont.)

- 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);`

## 17.3 Creating a Sequential File (cont.)

- 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.

## 17.3 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.
  - 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.

## 17.3 Creating a Sequential File (cont.)

- 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`.

## 17.3 Creating a Sequential File (cont.)

- 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.
- Figure 17.5 lists the keyboard combinations for entering end-of-file for various computer systems.

Computer system	Keyboard combination
UNIX/Linux/Mac OS X	<code>&lt;Ctrl-d&gt;</code> (on a line by itself)
Microsoft Windows	<code>&lt;Ctrl-z&gt;</code> (sometimes followed by pressing <code>Enter</code> )

Fig. 17.5 | End-of-file key combinations for various popular

## 17.3 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 17.4).
- The file created in Fig. 17.3 is simply a text file, so it can be viewed by any text editor.

## 17.3 Creating a Sequential File (cont.)

- 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` in the statement

## 17.4 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 17.6 reads records from the `clients.txt` file that we created using the program of Fig. 17.3 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.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. 17.6: Fig17_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
```

Fig. 17.6 | Reading and printing a sequential file. (Part 1 of 3.)

```
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( 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
```

Fig. 17.6 | Reading and printing a sequential file. (Part 2 of 3.)

```
35 // display single record from file
36 void outputLine( int account, const string name, double balance )
37 {
38     cout << left << setw( 10 ) << account << setw( 13 ) << name
39         << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
40 } // 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. 17.6 | Reading and printing a sequential file. (Part 3 of 3.)

## 17.4 Reading Data from a Sequential File (cont.)

- 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.
- 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.

## 17.4 Reading Data from a Sequential File (cont.)

- 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`

- 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.
- Figure 17.7 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. 17.7: Fig17_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. 17.7 | 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( 1 );
25 } // end if
26
27 int request;
28 int account;
29 string name;
30 double balance;
31
32 // get user's request (e.g., zero, credit or debit balance)
33 request = getRequest();
34 }
```

Fig. 17.7 | Credit inquiry program. (Part 2 of 8.)

---

```

35 // process user's request
36 while ( request != END )
37 {
38     switch ( request )
39     {
40         case ZERO_BALANCE:
41             cout << "\nAccounts with zero balances:\n";
42             break;
43         case CREDIT_BALANCE:
44             cout << "\nAccounts with credit balances:\n";
45             break;
46         case DEBIT_BALANCE:
47             cout << "\nAccounts with debit balances:\n";
48             break;
49     } // end switch
50
51 // read account, name and balance from file
52 inClientFile >> account >> name >> balance;
53

```

---

**Fig. 17.7 | Credit inquiry program. (Part 3 of 8.)**




---

```

54 // display file contents (until eof)
55 while ( !inClientFile.eof() )
56 {
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

```

---

**Fig. 17.7 | Credit inquiry program. (Part 4 of 8.)**



```

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

```

**Fig. 17.7 | Credit inquiry program. (Part 5 of 8.)**

```

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
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

```

**Fig. 17.7 | 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
  
```

**Fig. 17.7 | Credit inquiry program. (Part 7 of 8.)**

```

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. 17.7 | Credit inquiry program. (Part 8 of 8.)**

## 17.5 Updating Sequential Files

- Data that is formatted and written to a sequential file as shown in Section 17.3 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.

## 17.5 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 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.

## 17.5 Updating Sequential Files (cont.)

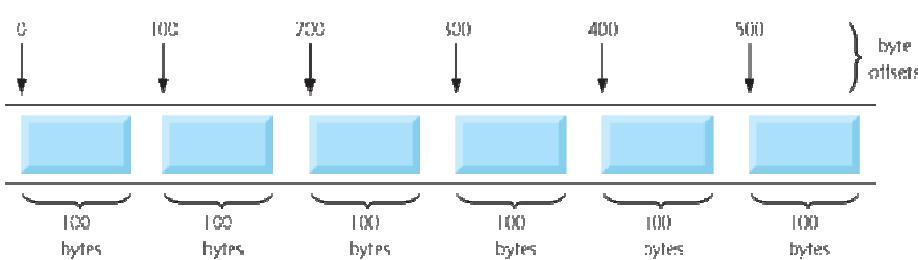
- Such updating can be done awkwardly.
- For example, to make the preceding name change, the records before **300 white 0.00** in a sequential file could be copied to a new file, the updated record then written to the new file, and the records after **300 white 0.00** copied to the new file.
- This requires 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.

## 17.6 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**.

## 17.6 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 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
- C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long)



## 17.6 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.

## 17.7 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.

## 17.7 Creating a Random-Access File (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.

## 17.7 Creating a Random-Access File (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 inconsistent.
- A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points.

## 17.7 Creating a Random-Access File (cont.)

- In Fig. 17.11, 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.

## 17.7 Creating a Random-Access File (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.

## 17.7 Creating a Random-Access File (cont.)

- Figure 17.11 illustrates opening a random-access file, defining the record format using an object of class **ClientData** (Figs. 17.9–17.10) 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 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.

## 17.7 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. 17.9, lines 32–33)
- Member functions **setLastName** (Fig. 17.10, lines 36–43) and **setFirstName** (Fig. 17.10, 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. 17.9: ClientData.h
2 // Class ClientData definition used in Fig. 17.11-Fig. 17.14.
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. 17.9 | ClientData class header. (Part 1 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. 17.9 | ClientData class header. (Part 2 of 2.)

```
1 // Fig. 17.10: 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,
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. 17.10 | 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. 17.10 | 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
```

**Fig. 17.10** | ClientData class represents a customer's credit information. (Part 3 of 4.)

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

## 17.7 Creating a Random-Access File (cont.)

- In Fig. 17.11, 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.
- Lines 24–25 cause the `blankClient` 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)
- 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

## 17.7 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. 17.11: Fig17_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
```

**Fig. 17.11 | Creating a random-access file with 100 blank records sequentially. (Part 1 of 2.)**

```
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
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. 17.11 | Creating a random-access file with 100 blank records sequentially. (Part 2 of 2.)**

## 17.8 Writing Data Randomly to a Random-Access File

- Figure 17.12 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. 17.9, so the program can use **ClientData** objects.

```
1 // Fig. 17.12: Fig17_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 }
```

Fig. 17.12 | Writing to a random-access file. (Part 1 of 4.)

```
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 {
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 }
```

Fig. 17.12 | Writing to a random-access file. (Part 2 of 4.)

```
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
```

Fig. 17.12 | 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. 17.12 | Writing to a random-access file. (Part 4 of 4.)

- 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 (`|`).

- 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 30–31 from Fig. 17.13 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. 17.13: Fig17_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 }
```

Fig. 17.13 | Reading a random-access file sequentially. (Part 1 of 4.)

```
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
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 }
```

Fig. 17.13 | Reading a random-access file sequentially. (Part 2 of 4.)

```
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
```

Fig. 17.13 | Reading a random-access file sequentially. (Part 3 of 4.)

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. 17.13 | Reading a random-access file sequentially. (Part 4 of 4.)

- Figure 17.13 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 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.

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

## 17.10 Case Study: A Transaction-Processing Program

- We now present a substantial transaction-processing program (Fig. 17.14) 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. 17.11 has been executed to create the file **credit.dat** and that the program of Fig. 17.12 has been executed to insert the initial data.

```
1 // Fig. 17.14: Fig17_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. 17.14 | Bank account program. (Part I of 14.)

```
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 ( 1 );
32     } // end if
33
34     int choice; // store user choice
35 }
```

Fig. 17.14 | Bank account program. (Part 2 of 14.)

```
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
```

Fig. 17.14 | Bank account program. (Part 3 of 14.)

```
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. 17.14 | Bank account program. (Part 4 of 14.)

```
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 }
```

Fig. 17.14 | Bank account program. (Part 5 of 14.)

```
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
```

Fig. 17.14 | Bank account program. (Part 6 of 14.)

```
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. 17.14 | Bank account program. (Part 7 of 14.)

```

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

```

**Fig. 17.14 | Bank account program. (Part 8 of 14.)**

```

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

```

**Fig. 17.14 | Bank account program. (Part 9 of 14.)**

```
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;
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
```

Fig. 17.14 | Bank account program. (Part 10 of 14.)

```
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
```

Fig. 17.14 | Bank account program. (Part 11 of 14.)

```
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 }
```

Fig. 17.14 | Bank account program. (Part 12 of 14.)

```
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. 17.14 | Bank account program. (Part 13 of 14.)

```
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. 17.14 | Bank account program. (Part 14 of 14.)

## 7.10 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 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. 17.13 to input data from `credit.dat`
- Function `outputLine`, discussed in Section 17.9, 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.

- 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.
- 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.
- 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. 17.11.

- 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.

## 17.11 Object Serialization

- 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.

## 17.11 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 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

## 17.11 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](http://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](http://www.boost.org/libs/serialization/doc/index.html)).

# Complementos de Programação de Computadores – Aula 6c Ficheiros

Mestrado Integrado em Electrónica Industrial e Computadores

**Luis Paulo Reis**

[lpreas@dsi.uminho.pt](mailto:lpreas@dsi.uminho.pt)

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia,  
Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2012)

