Random-Access Files

Lab 14: Bank Account Transactions

Rung-Bin Lin
International Bachelor Program in Informatics
Yuan Ze University

5/26/2022

Outline

- Files and stream
- Sequential access file vs Random access file
- Transaction processing using random access file
- See Chapter 8 for Sequential Files
- See Chapter 17 for Random Access Files

Files and Streams

- C++ views each file as a sequence of bytes (Fig. 8.2). It imposes no structure on a file.

Fig. 8.2 | C++'s view of a file of *n* bytes.

- Each file ends generally with an end-of-file marker.
- When a file is opened, an object is created, and a stream is associated with the object.
- In Chapter 15, we saw that objects `cin`, `cout`, `cerr` and `clog` are created when `<iostream>` is included.

Files and Streams (cont.)

- The streams associated with these objects provide communication channels between a program and a particular file or device.
- To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included.
- Three kinds of file streams
  - **ifstream**: Input file stream for reading (input)
  - **ofstream**: Output file stream for writing (output)
  - **fstream**: File stream for reading and writing (input and output)

Formatted Data VS. Raw (Binary) Data

- Formatted data

  2^{31}-1 = +2147483647 stored in file as formatted data needs 11 bytes

  - Different numbers may need different number of bytes when they are stored in a file.

- Raw data (Binary data)

  2^{31}-1 = +2147483647 stored in file as formatted data needs 32 bits (4 bytes)whose content will be 01111111 11111111 11111111 11111111

  - Different numbers need the same number of bytes when they are stored in a file.

Sequential file

- File access is done from the beginning consecutively to the end of a file.
- Not suitable for random access

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

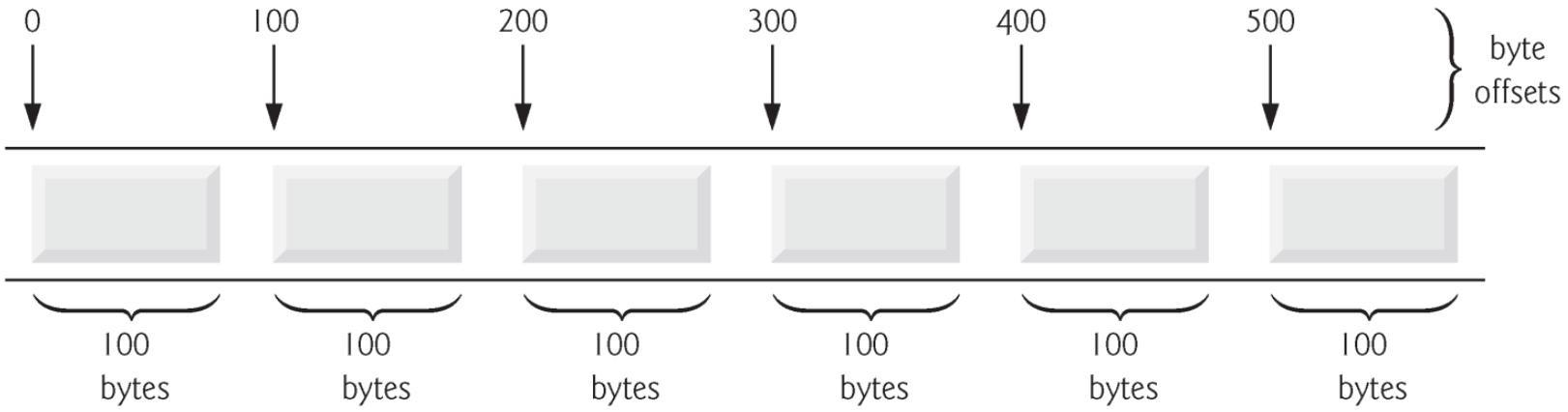


Fig. 17.1 | C++ view of a random-access file.

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.

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.

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 (raw data) 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.
    ```

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

Creating a Random-Access File (cont.)

- In Fig. 17.4, 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.

Create and Open an ofstream File at the Same Time

- **In Fig. 8.3,** ofstream outClientFile("clients.txt", ios::out); **creates an ofstream object outClientFile and open it at the same time for output by associating it with the file named clients.txt.**
  - **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).**
  - **By default, `ofstream` objects are opened for output, so the open mode is not required in the constructor call.**
- **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.**

File Open Modes

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

Fig. 8.4 | File open modes.

- Default open modes:
  - ifstream is ios::in
  - ofstream is ios::out
  - fstream is ios::in | ios::out
- **ios::trunc** is used when we would like to overwrite (replacing old data with new data)

Create and Open an ofstream File at Different Time

- An `ofstream` object can be created without opening a specific file—a file can be attached to the object later.
  - For example, the statement below creates an `ofstream` object named `outClientFile`.

```
ofstream outClientFile;
```

- The `ofstream` member function `open()` opens a file and attaches it to an existing `ofstream` object as follows:
  - `outClientFile.open("clients.txt", ios::out);`

Open a File for Reading

- Line 15, ifstream inClientFile("clients.txt", ios::in); creates an ifstream object and open the object by associating it with a file named **clients.txt**.
  - An ifstream object is opened for input by default, so to open `clients.txt` for input we could have used the statement
    - `ifstream inClientFile( "clients.txt" );`
- Just as with an `ofstream` object, an `ifstream` object can be created first and opened later by associating it with an existing 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 `while` condition returns `false`), terminating the `while` statement and the program; this causes the `ifstream` destructor function to close the file.

Reading All the File Data Several Times

- To get access to data, `ifstream` and `ofstream` provide member functions for repositioning the **file-position pointer**.
  - `seekg(n,dir)` for `ifstream` to set the "**get**" file-position pointer for "**read**"
  - seekg(position) sets the "**get**" file pointer to a designated position for "**read**". Position zero is the beginning of the file.
  - `Seekp(n,dir)` for `ofstream` to set the "**put**" file-position pointer for "**write**"
  - Seekp(position) sets the "**put**" file pointer to a designated position for "**write**". Position zero is the beginning of the file.
  - `seekg(n,dir), seekg(position)`, `seekp(n,dir)` seekp(position) for `fstream`

Setting the Value of File-Position Pointer

- The first argument **n** specified the number of bytes used to calculate the value of the file-position pointer.
- The second argument **dir** 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 value of the file-position pointer is calculated as follows:
  - If **dir** is **ios::beg**, the value is **beg+n**.
  - If **dir** is **ios::cur**, the value is **cur+n**.
  - If **dir** is ios::end, the value is **end-n**.
- The statement `inClientFile.seekg( 0 );`

  repositions the file-position pointer to the beginning of the file (location `beg`, i.e., 0) attached to `inClientFile`.

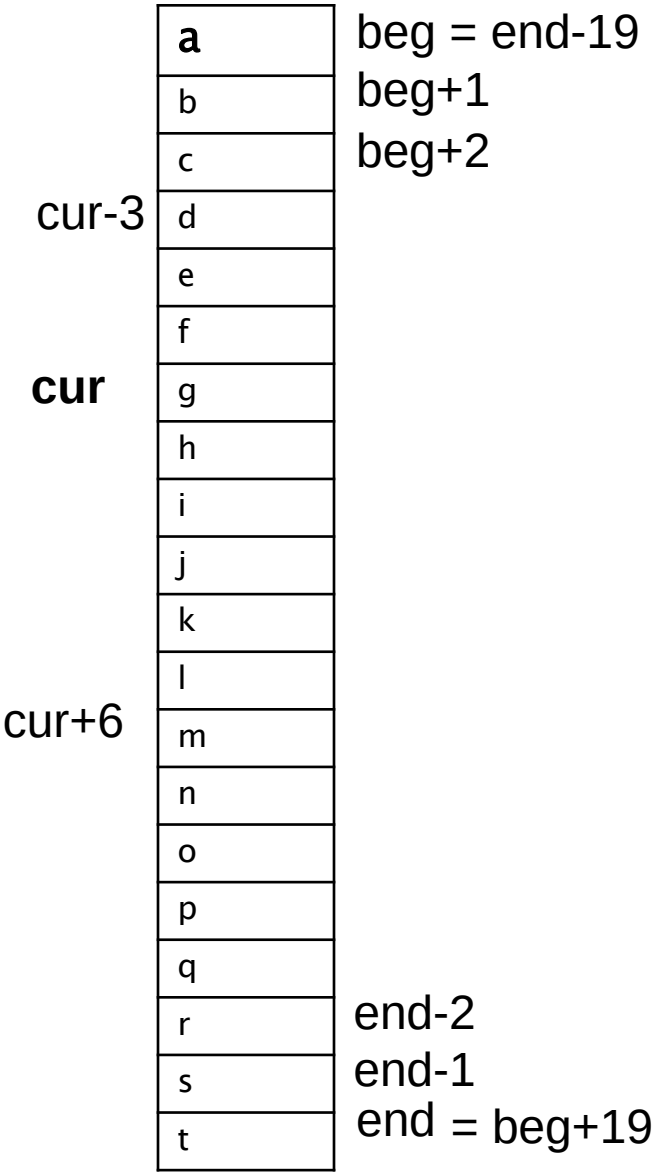
Some examples of Setting File-Position Pointer's Value

- `// 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 `ofstream` member function `seekp`.
- Member functions `tellg` and `tellp` are provided to return the current locations of the "get" and "put" pointers, respectively.

Example of File-Position Pointers

- File-position pointers are maintained in the file object. You typically need not explicitly specify the pointer when you read/ write a file. However, it can be updated by using seekg(n, dir) or seekp(n, dir).
- When a read/write is completed, the file-position pointer will be updated to point to the next place.
- If you would like to read a file again from the beginning of the file, you have two ways:
  - Close the file and then re-opened it. Not suggested.
  - Reposition the file-position pointer to the beginning of the file using fileObject.`Seekg(0,beg)` for input.
- If you would like to overwrite the file from the beginning of the file, you should use fileObject.Seekp(0,beg).
- To update a file, you should be more cautious. You normally can not just write the data at the location (pointed by cur) where the original data is stored



Example of access to a Randome access File

```
1  // Fig. 17.2: ClientData.h
2  // Class ClientData definition used in Fig. 17.4–Fig. 17.7.
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. 17.2 | `ClientData` class header file. (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
```

private:
int accountNumber;
string lastName;
string firstName;
double balance;

Fig. 17.2 | `ClientData` class header file. (Part 2 of 2.)

Private data section must have a fixed size of memory per object for random access. Using a string type for private data here is not suitable for creating a random access file for this sort of object because string type makes the size of data section varying with each object .

```
1  // Fig. 17.3: 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. 17.3 | `ClientData` class represents a customer's credit information. (Part 1 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. 17.3 | `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. 17.3 | `ClientData` class represents a customer's credit information. (Part 3 of 4.)

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

Fig. 17.3 | `ClientData` class represents a customer's credit information. (Part 4 of 4.)

```
1  // Fig. 17.4: Fig17_04.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. 17.4 | 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. 17.4 | Creating a random-access file with 100 blank records sequentially. (Part 2 of 2.)

Writing Data Randomly to Random Access File

```
1  // Fig. 17.5: 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; // customer's account number
12    string lastName; // customer's last name
13    string firstName; // customer's first name
14    double balance; // amount of money customer owes company
15
16    fstream outCredit( "credit.dat", ios::in | ios::out | ios::binary );
17
18    // exit program if fstream cannot open file
19    if ( !outCredit )
20    {
21       cerr << "File could not be opened." << endl;
22       exit( 1 );
23    } // end if
24
```

Fig. 17.5 | 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. 17.5 | 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
```

Fig. 17.5 | 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. 17.5 | Writing to a random-access file. (Part 4 of 4.)

Reading from Random Access File Sequentially

```
1  // Fig. 17.6: Fig17_06.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. 17.6 | Reading a random-access file sequentially. (Part 1 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. 17.6 | 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. 17.6 | Reading a random-access file sequentially. (Part 3 of 3.)

Lab 14: Bank Account Transactions using Random Access File

- Modify the code of the function void newRecord(fstream &insertInFile) in Fig. 17.7 into int newRecord(fstream &insertInFile). The modified function should automatically find an account number that is not used for creating a new record. The valid account number is from 1 to 100. The account number should be the smallest one among all the unused account numbers. Hence, line 162 in page 741 should be replaced by a function called int getSmallestUnusedActNo(fstream &). This function returns the smallest account number if some unused account numbers exist. Otherwise, it returns a zero. Moreover, If a new record can be created successfully, the modified function should return the account number of the new record. It should also print "A new account is created successfully: actNo" where actNo is the account number. Otherwise, it returns a zero and print "A new account will not be created!" Basically, you have to rewrite newRecord(…) because of using getSmallestUnusedActNo(…).

- **You must implement a new function void printAllRecords(fstream &) that will print out all the valid records, i.e., printing only the record whose account number is greater than zero. This function should make a call to outputLine() to print out a record.**
- **Run the modified program derived from the one in Fig. 17.7 with the following transactions using a given file credit.dat:**
  - Update the balance of account 35 by adding 100
  - Update the balance of account 43 by adding 154
  - Delete account 6
  - Create a new account with input: Johnston Biden 10000.0
  - Delete account 29 again
  - Delete account 29 again
  - End

Requirement: You have to use the header file given in Fig. 17.2.

Main() Function

- Main() function has been modified and given to you below.

```
int main()
{
   // open file for reading and writing
   fstream inOutCredit( "credit.dat", ios::in | ios::out | ios::binary );

   // exit program if fstream cannot open file
   if ( !inOutCredit )
   {
      cerr << "File could not be opened." << endl;
      exit ( 1 );
   } // end if

   int choice; // store user choice
    printAllRecords(inOutCredit);
    inOutCredit.clear(); // reset end-of-file indicator
   // enable user to specify action
   while ( ( choice = enterChoice() ) != END )
   {
      switch ( choice )
      {
         case PRINT: // create text file from record file
            createTextFile( inOutCredit );
            break;
         case UPDATE: // update record
            updateRecord( inOutCredit );
            break;
         case NEW: // create record
            newRecord( inOutCredit );
            break;
         case DELETE: // delete existing record
            deleteRecord( inOutCredit );
            break;
         default: // display error if user does not select valid choice
            cerr << "Incorrect choice" << endl;
            break;
      } // end switch

      inOutCredit.clear(); // reset end-of-file indicator
   } // end while
   printAllRecords(inOutCredit);
} // end main
```

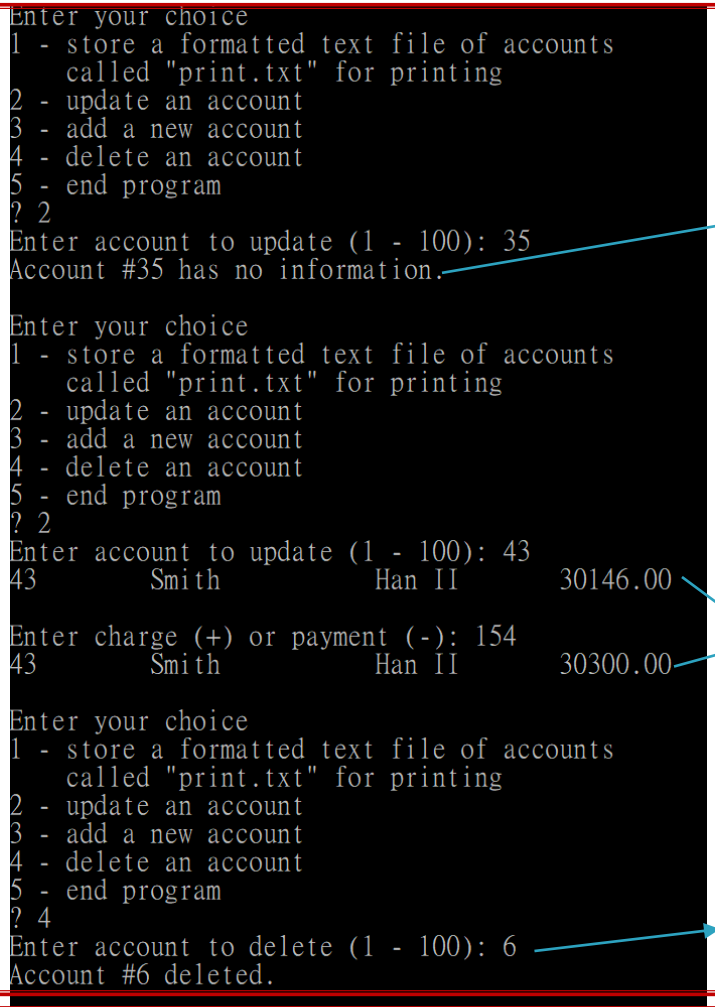
Tips

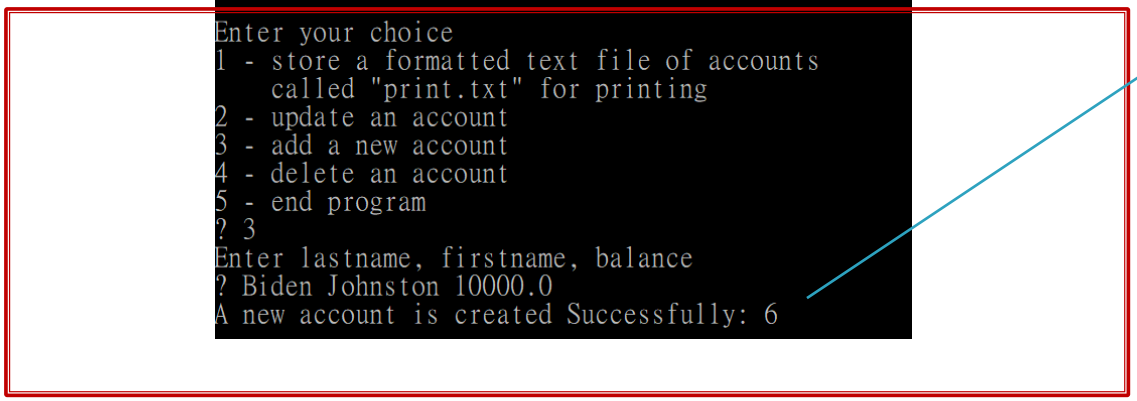
- Remember to clear the eof flag using fileHandle.clear() once the file position pointer reaches the end of file marker.

Input and Output of Running the Program

All the valid records before doing any transactions.

Account	Last Name	First Name	Balance
1	Adams	Berg X	7541.00
2	Adams	Mary IV	26685.00
3	Monteli	Pey II	26111.00
4	Monteli	Han IV	2448.00
5	Monteli	Pey IV	28658.00
6	Brams	Will VI	20950.00
7	Subert	Jane I	11871.00
8	Hamilton	Pey X	4905.00
9	Subert	John VI	9610.00
10	Bach	Berg VI	20759.00
11	Brams	John IX	16234.00
19	Adams	Mary VIII	835.00
20	Heisenberg	John V	18041.00
21	Adams	Mary V	12227.00
29	Jhonston	Pey I	25990.00
30	Subert	Jane VIII	3918.00
31	Bach	Mary II	18843.00
33	Subert	Gord VIII	5320.00
34	Smith	Tom I	8430.00
39	Jhonston	Pey IV	25179.00
43	Smith	Han II	30146.00
53	Brams	Gord IX	23003.00
56	Monteli	Will I	8807.00
61	Bach	Will X	23280.00
62	Smith	Mary VII	24286.00
64	Brams	Han IV	28054.00
72	Brams	Berg II	25796.00
89	Morzart	John X	31524.00
90	Brams	Tom IV	5023.00
92	Subert	Gord II	31564.00
94	Monteli	Vick II	9281.00
95	Monteli	John II	12007.00
98	Brams	Jane X	32603.00
100	Jhonston	Pey VI	25175.00

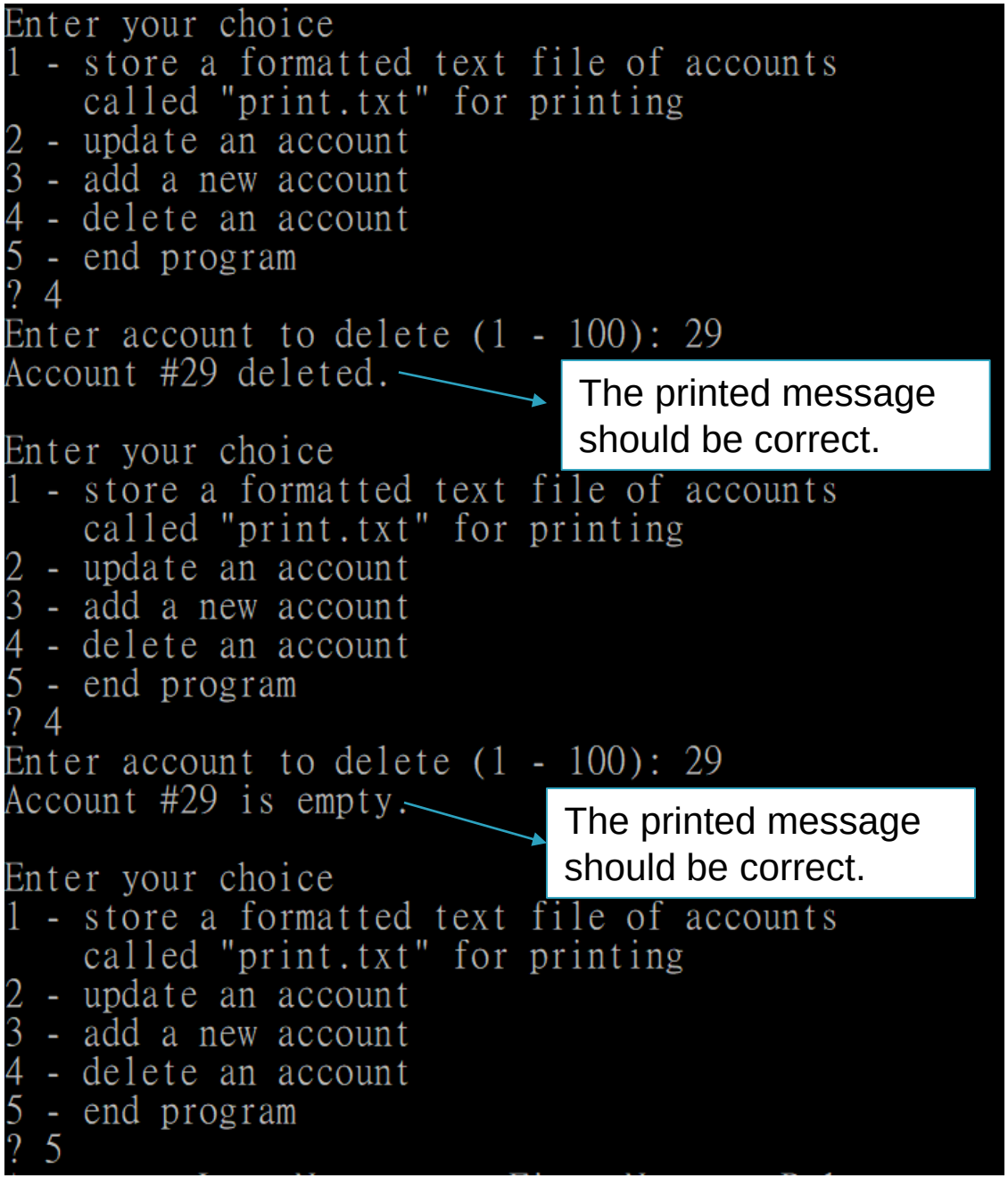




If you meet something like this, just ignore it. This might be the problem of Code::Blocks.

The account number and its balance should be correct.

The account number should be correct.



The printed message should be correct.

The printed message should be correct.

File content after processing all transactions:

- Account 6 updated with a new record “Biden Johnston 10000.0”
- Account 29 deleted
- Account 43’s balance updated to 30300.0

Account	Last Name	First Name	Balance
1	Adams	Berg X	7541.00
2	Adams	Mary IV	26685.00
3	Monteli	Pey II	26111.00
4	Monteli	Han IV	2448.00
5	Monteli	Pey IV	28658.00
6	Biden	Johnston	10000.00
7	Subert	Jane I	11871.00
8	Hamilton	Pey X	4905.00
9	Subert	John VI	9610.00
10	Bach	Berg VI	20759.00
11	Brams	John IX	16234.00
19	Adams	Mary VIII	835.00
20	Heisenberg	John V	18041.00
21	Adams	Mary V	12227.00
30	Subert	Jane VIII	3918.00
31	Bach	Mary II	18843.00
33	Subert	Gord VIII	5320.00
34	Smith	Tom I	8430.00
39	Jhonston	Pey IV	25179.00
43	Smith	Han II	30300.00
53	Brams	Gord IX	23003.00
56	Monteli	Will I	8807.00
61	Bach	Will X	23280.00
62	Smith	Mary VII	24286.00
64	Brams	Han IV	28054.00
72	Brams	Berg II	25796.00
89	Morzart	John X	31524.00
90	Brams	Tom IV	5023.00
92	Subert	Gord II	31564.00
94	Monteli	Vick II	9281.00
95	Monteli	John II	12007.00
98	Brams	Jane X	32603.00
100	Jhonston	Pey VI	25175.00

Fig. 17.7: A Transaction-Processing Program

```
1  // Fig. 17.7: Fig17_07.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. 17.7 | Bank account program. (Part 1 of 13.)

```
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. 17.7 | Bank account program. (Part 2 of 13.)

```
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. 17.7 | Bank account program. (Part 3 of 13.)

```
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. 17.7 | Bank account program. (Part 4 of 13.)

```
79  // create formatted text file for printing
80  void createTextFile( fstream &readFromFile )
81  {
82     // create text file
83     ofstream outPrintFile( "print.txt", ios::out );
84
85     // exit program if ofstream cannot create file
86     if ( !outPrintFile )
87     {
88        cerr << "File could not be created." << endl;
89        exit( 1 );
90     } // end if
91
92     outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
93        << "Last Name" << setw( 11 ) << "First Name" << right
94        << setw( 10 ) << "Balance" << endl;
95
96     // set file-position pointer to beginning of readFromFile
97     readFromFile.seekg( 0 );
98
99     // read first record from record file
100    ClientData client;
101    readFromFile.read( reinterpret_cast< char * >( &client ),
102       sizeof( ClientData ) );
```

Fig. 17.7 | Bank account program. (Part 5 of 13.)

```
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. 17.7 | Bank account program. (Part 6 of 13.)

```
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
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
```

Fig. 17.7 | Bank account program. (Part 7 of 13.)

```
141         // update record balance
142         double oldBalance = client.getBalance();
143         client.setBalance( oldBalance + transaction );
144         outputLine( cout, client ); // display the record
145
146         // move file-position pointer to correct record in file
147         updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );
148
149         // write updated record over old record in file
150         updateFile.write( reinterpret_cast< const char * >( &client ),
151            sizeof( ClientData ) );
152      } // end if
153      else // display error if account does not exist
154         cerr << "Account #" << accountNumber
155            << " has no information." << endl;
156   } // end function updateRecord
157
```

Fig. 17.7 | Bank account program. (Part 8 of 13.)

```
158 // create and insert record
159 void newRecord( fstream &insertInFile )
160 {
161    // obtain number of account to create
162    int accountNumber = getAccount( "Enter new account number" );
163
164    // move file-position pointer to correct record in file
165    insertInFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
166
167    // read record from file
168    ClientData client;
169    insertInFile.read( reinterpret_cast< char * >( &client ),
170       sizeof( ClientData ) );
171
172    // create record, if record does not previously exist
173    if ( client.getAccountNumber() == 0 )
174    {
175       string lastName;
176       string firstName;
177       double balance;
178
179       // user enters last name, first name and balance
180       cout << "Enter lastname, firstname, balance\n? ";
181       cin >> setw( 15 ) >> lastName;
```

Fig. 17.7 | Bank account program. (Part 9 of 13.)

```
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
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. 17.7 | Bank account program. (Part 10 of 13.)

```
203 // delete an existing record
204 void deleteRecord( fstream &deleteFromFile )
205 {
206    // obtain number of account to delete
207    int accountNumber = getAccount( "Enter account to delete" );
208
209    // move file-position pointer to correct record in file
210    deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
211
212    // read record from file
213    ClientData client;
214    deleteFromFile.read( reinterpret_cast< char * >( &client ),
215       sizeof( ClientData ) );
216
217    // delete record, if record exists in file
218    if ( client.getAccountNumber() != 0 )
219    {
220       ClientData blankClient; // create blank record
221
222       // move file-position pointer to correct record in file
223       deleteFromFile.seekp( ( accountNumber - 1 ) *
224          sizeof( ClientData ) );
225
```

Fig. 17.7 | Bank account program. (Part 11 of 13.)

```
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
246
```

Fig. 17.7 | Bank account program. (Part 12 of 13.)

```
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. 17.7 | Bank account program. (Part 13 of 13.)