UNIT 4 A CASE STUDY

Structure **Page Nos.**

4.0 INTRODUCTION

In the units covered so far, we have discussed various features of C++ language that help in designing and implementing useful programs to solve various real world problems. The bottom up approach adopted by C++ language provides a better way to capture the details of real world problems and design efficient and adaptable solutions. Whether it is the mechanism of constructors and destructors, inheritance, or polymorphism; all taken collectively provide suitable design methodology and tool for solving various real world problems through C++ programming formulations. In order to solve a real world problem, the first and foremost requirement is to identify the various objects in the system along with their general attributes. This is then followed by the more involved process of identifying the methods/ functions that can be applied on the different objects. Once this is done, the effort gets more focused towards design, which involves designing various classes and implementing different functions.

This unit presents a case study of designing and implementing a transaction-processing system for banking domain. Unlike a database design for bank accounts, we have adopted a file processing approach to emphasize the C++ features and capabilities. The design involves creating necessary data files to store accounts and customer information and then accessing them through suitable code for writing data, appending data, performing credit operations and displaying the results. Our focus in the case study is on demonstrating the design methodology and the steps required to design a small real world application. The steps involved in design and implementation of the transaction-processing system are described with relevant explanations at various places. The program design also makes use of certain features of C++ which are used for larger programs. After a careful study of the unit, you would be able to clearly identify the broad guidelines and general steps involved in solving a real world problem and using C++ for solving variety of problems.

4.1 OBJECTIVES

At the end of the unit, you should be able to:

- explain the steps involved in solving real world problems;
- describe the overall framework of such designs;
- appreciate the usefulness of various features of C++ for solving different real problems;
- design C++ programming solutions for many other problems; and
- design C++ programs involving multiple files.

4.2 DESIGNING A TRANSACTION-PROCESSING SYSTEM

A transaction processing system is one where certain activities are carried out as part of some bigger goal. The kind of transactions performed in a system depends on the domain of the problem. For example, in a ticket reservation domain the key transactions may be booking a ticket, realizing payment for a ticket, cancelling a ticket, modifying or amending a ticket, refund of a cancelled ticket etc. Similarly in a banking domain, the transactions could be opening a customer account, updating the account records, processing withdrawal from an account, deposit into an account, printing summary of accounts etc. Every transaction processing system, irrespective of the domain, has to complete certain activities (referred to as transactions). Nowadays when a large number of transaction processing activities are automated to be performed on a computer system, it is very important to know and understand how can we design such a system. Moreover, with the increased use of Internet and web-based services many of these transactions are initiated and realized at different physical machines. The data related to transactions is sent over communication lines. One common thing however in all kind of transaction processing systems is the need to store and manipulate associated data. The data of the system may be stored either in a database format or in terms of a collection of various files. The general practice in most of the real world transaction processing systems is to go for a database design approach. However, in the example described below we have used a file processing approach.

Most of the transaction processing systems are quite big and result into large programs. The large programs often comprise of various modules. In order to have better understanding and design convenience, these transaction processing systems are designed as a collection of multiple programs. In the previous units we have largely seen example programs consisting of a single program file. Whenever we have to design programming solution for a larger problem, it is often required to organize the large code into multiple files. Using a multiple file organization not only helps in clarity of the design but also in appropriate use of class libraries and better coordination of programmers working on the large project. In fact, large programs are usually divided into separate files, where different files have code for different functionalities, such as one file for mathematical analysis, another for graphics display and a separate one for I/O etc. Large applications sometimes also involve multiple designers who coordinate their effort to design the final solution. Most of the C++ IDEs provide a feature called PROJECT which helps in designing and organizing larger programs comprising multiple files.

We will now see how we can use the various features and capabilities of C++ that we have learned so far to design a transaction-processing system. The proposed system involves some fixed-length account records for a company having certain number of customers. Each record consists of an account number that acts as the record key, a last name, a first name and a balance. The transaction-processing program is to be designed in such a manner that it can provide overall management functions for the accounts and transactions. It should be able to perform functions like update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.

The key components in this application design can be understood through following abstract diagram representation.

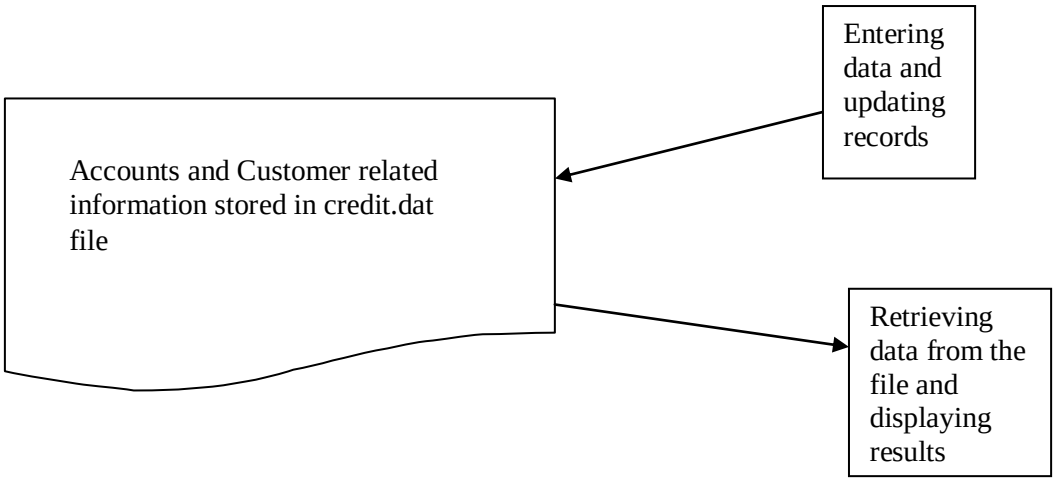

Figure 4.1 : Abstract Diagram of transaction processing system

As we can see, the entire data corresponding to accounts and customer information is stored in credit.dat file. We will design program to enter, add and update the data in this file. The data entered may also be retrieved and displayed as a formatted text output through the use of various stream manipulators. We first create a ClientData class header file that defines the format of the data and then define the constructor and certain basic methods. The following two program segments (Program 4.1 and 4.2) are written for this purpose.

```
 1  #ifndef CLIENTDATA_H
 2  #define CLIENTDATA_H
 3  #include <string>
 4  using namespace std;
 5  class ClientData
 6  {
 7  public:
 8  ClientData(int =0, string = " ", string = " ", double =
0.0);
 9
10  void setAccountNumber (int);
11  int getAccountNumber() const;
12  void setLastname(string);
13  string getLastName() const;
14  void setFirstName(string);
15  string getFirstName() const;
16  void setBalance(double);
17  double getBalance() const;
18
19  private:
20  int accountNumber;
21  char lastName[15];
22  char firstName[10];
23  double balance;
24  };
25  #endif
```

Program 4.1: ClientData class header file

The program 4.1 above defines client data header file which specifies the data format to be used in the application. The main data items are account number (an integer

value), last name of customer (a string), first name of customer (a string) and balance (a float value denoting the account balance). These data items are declared as private. The methods to access and modify this data are setAccountNumber(), getAccountNumber(), setLastName(), getLastName(), setFirstName(), getFirstName(), setBalance() and getBalance(). All these functions are used to define and retrieve values for various fields, and are defined in next program (Program 4.2). The client data represent a customer's credit information and the methods described in program 4.2 provide the code for manipulating the data. The exact code is described in in the program 4.2 given below:

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

```
50    firstNameString.copy(firstName, length);
51    firstName[length]='\0'; \\ appending null char
52   }
53
54    // get balance value
55    double ClientData::getBalance() const
56    {
57     return balance;
58    }
59
60    //set balance value
61    void ClientData::setBalance(double lalanceValue)
62    {
63     balance=balanceValue;
64    }
65
```

Program 4.2: ClientData class representing customer credit information

As you may easily notice, the program defines the ClientData constructor comprising of various functions, each of which have a specified code. The functions setLastName() and setFirstName() limit the number of characters read from input that are finally written to actual data.

We now look at the code (Program 4.3) for creating a file credit.dat with some data entries to be used in our transaction processing system. The program creates a binary file credit.dat for output. It then writes 100 blank records into the credit.dat data file. The next program (program 4.4) then uses various functions to actually write data into this file.

```
 1  // creating randomly accessible file credit.dat
 2  #include <iostream>
 3  #include <fstream>
 4  #include <cstdlib>
 5  #include "ClientData.h"
 6  using namespace std;
 7
 8  int main()
 9 {
10  ofstream outCredit ("credit.dat", ios::out | ios::binary);
11  if (!outCredit)
12  {
13    cerr << "File could not be opened." << endl;
14    exit(1);
15  }
16
17  ClientData blankClient; // constructor zeros out each data
member
18  // output 100 blank records to file
19  for (int i=0; i<100, i++)
20    outCredit.write(reinterpret_cast < const char * >
(&blankClient),
21     sizeof(ClientData));
22  }
```

Program 4.3: Creating the credit.dat file with 100 blank records

Once the file credit.dat is created we can use this file to store the desired data corresponding to the accounts and customers. After entering the basic data, actual

transaction-processing may be performed. The program below reads the data from user entered values through keyboard and then uses fstream functions to store data at desired locations in the file credit.dat. Note that the file is opened in out mode for writing. An example run of the program 4.4 is presented after the program code. The run shows how different data values can be entered into the data file. Note that line 19 includes the header file ClientData.h defined in Program 4.1 so the program can use ClientData objects.

```
 1
 2  // Writing to a random-access file.
 3  #include <iostream>
 4  using std::cerr;
 5  using std::cin;
 6  using std::cout;
 7  using std::endl;
 8  using std::ios;
 9
10  #include <iomanip>
11  using std::setw;
12
13  #include <fstream>
14  using std::fstream;
15
16  #include <cstdlib>
17  using std::exit; // exit function prototype
18
19  #include "ClientData.h" // ClientData class definition
20
21  int main()
22  {
23     int accountNumber;
24     char lastName[ 15 ];
25     char firstName[ 10 ];
26     double balance;
27
28     fstream outCredit( "credit.dat", ios::in | ios::out |
ios::binary );
29
30     // exit program if fstream cannot open file
31     if ( !outCredit )
32     {
33        cerr << "File could not be opened." << endl;
34        exit( 1 );
35     } // end if
36
37     cout << "Enter account number (1 to 100, 0 to end
input)\n? ";
38
39     // require user to specify account number
40     ClientData client;
41     cin >> accountNumber;
42
43     // user enters information, which is copied into file
44     while ( accountNumber > 0 && accountNumber <= 100 )
45     {
46        // user enters last name, first name and balance
47        cout << "Enter lastname, firstname, balance\n? ";
48        cin >> setw( 15 ) >> lastName;
49        cin >> setw( 10 ) >> firstName;
50        cin >> balance;
51
```

```
52        // set record accountNumber, lastName, firstName and
balance values
53        client.setAccountNumber( accountNumber );
54        client.setLastName( lastName );
55        client.setFirstName( firstName );
56        client.setBalance( balance );
57
58        // seek position in file of user-specified record
59        outCredit.seekp( ( client.getAccountNumber() - 1 ) *
60           sizeof( ClientData ) );
61
62        // write user-specified information in file
63        outCredit.write( reinterpret_cast< const char * >(
&client ),
64           sizeof( ClientData ) );
65
66        // enable user to enter another account
67        cout << "Enter account number\n? ";
68        cin >> accountNumber;
69     } // end while
70
71     return 0;
72  } // end main
```

Program 4.4: Writing data to credit.dat file

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Singh Shweta 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Tiwari Nisha -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Jolly Stellina 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Sen Ajay 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Ghosh Soumitra 314.33
Enter account number
? 0
```

The transaction processing system that we are designing can now be visualized in figure 4.2:

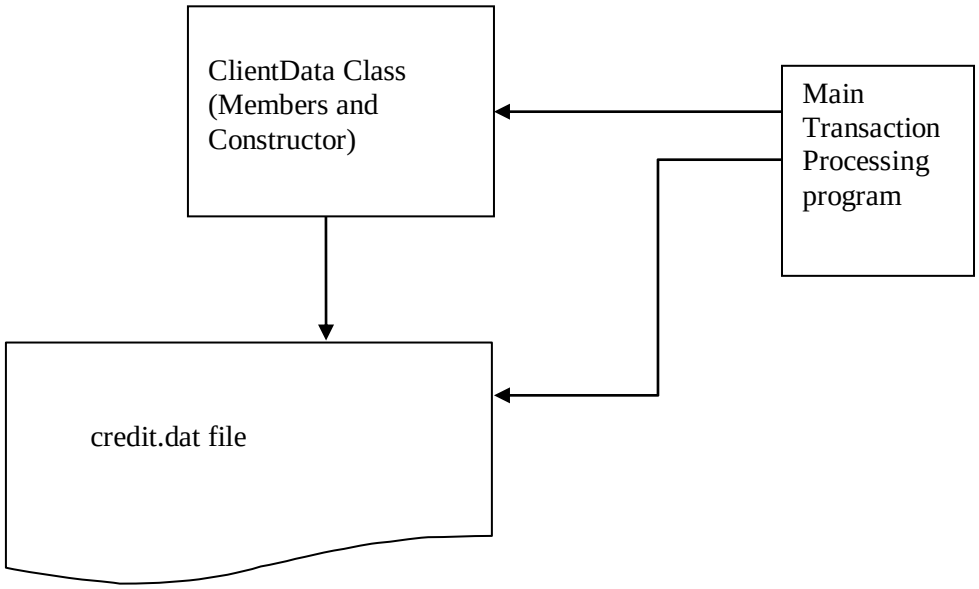

Figure 4.2: Transaction Processing System Program Structure

We now present our main transaction-processing program (Program 4.5) which uses the ClientData.h and credit.dat files to achieve "instant" -access processing. As we discussed earlier, the program manages a bank's account information. The program can perform all functions of accounts processing. It can update existing accounts, adds new accounts, deletes accounts and stores a formatted listing of all current accounts in a text file. We assume that the program 4.3 has been executed to create the file credit.dat and that the program of Program 4.4 has been executed to insert the initial data, before this program can be used for transaction-processing operations.

```
 1
 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   using std::cerr;
 7   using std::cin;
 8   using std::cout;
 9   using std::endl;
10   using std::fixed;
11   using std::ios;
12   using std::left;
13   using std::right;
14   using std::showpoint;
15
16   #include <fstream>
17   using std::ofstream;
18   using std::ostream;
19   using std::fstream;
20
21   #include <iomanip>
22   using std::setw;
```

```
23   using std::setprecision;
24
25   #include <cstdlib>
26   using std::exit; // exit function prototype
27
28   #include "ClientData.h" // ClientData class definition
29
30   int enterChoice();
31   void createTextFile( fstream& );
32   void updateRecord( fstream& );
33   void newRecord( fstream& );
34   void deleteRecord( fstream& );
35   void outputLine( ostream&, const ClientData & );
36   int getAccount( const char * const );
37
38   enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
39
40   int main()
41   {
42      // open file for reading and writing
43      fstream inOutCredit( "credit.dat", ios::in | ios::out |
ios::binary );
44
45      // exit program if fstream cannot open file
46      if ( !inOutCredit )
47      {
48         cerr << "File could not be opened." << endl;
49         exit ( 1 );
50      } // end if
51
52      int choice; // store user choice
53
54      // enable user to specify action
55      while ( ( choice = enterChoice() ) != END )
56      {
57         switch ( choice )
58         {
59            case PRINT: // create text file from record file
60               createTextFile( inOutCredit );
61               break;
62            case UPDATE: // update record
63               updateRecord( inOutCredit );
64               break;
65            case NEW: // create record
66               newRecord( inOutCredit );
67               break;
68            case DELETE: // delete existing record
69               deleteRecord( inOutCredit );
70               break;
71            default: // display error if user does not select
valid choice
72               cerr << "Incorrect choice" << endl;
73               break;
74         } // end switch
75
76         inOutCredit.clear(); // reset end-of-file indicator
77      } // end while
78
79      return 0;
80   } // end main
81
82   // enable user to input menu choice
83   int enterChoice()
```

```
84   {
85      // display available options
86      cout << "\nEnter your choice" << endl
87         << "1 - store a formatted text file of accounts" <<
endl
88         << " called \"print.txt\" for printing" << endl
89         << "2 - update an account" << endl
90         << "3 - add a new account" << endl
91         << "4 - delete an account" << endl
92         << "5 - end program\n? ";
93
94      int menuChoice;
95      cin >> menuChoice; // input menu selection from user
96      return menuChoice;
97   } // end function enterChoice
98
99   // create formatted text file for printing
100  void createTextFile( fstream &readFromFile )
101  {
102     // create text file
103     ofstream outPrintFile( "print.txt", ios::out );
104
105     // exit program if ofstream cannot create file
106     if ( !outPrintFile )
107     {
108        cerr << "File could not be created." << endl;
109        exit( 1 );
110     } // end if
111
112     outPrintFile << left << setw( 10 ) << "Account" <<
setw( 16 )
113        << "Last Name" << setw( 11 ) << "First Name" <<
right
114        << setw( 10 ) << "Balance" << endl;
115
116     // set file-position pointer to beginning of
readFromFile
117     readFromFile.seekg( 0 );
118
119     // read first record from record file
120     ClientData client;
121     readFromFile.read( reinterpret_cast< char * >( &client
),
122        sizeof( ClientData ) );
123
124     // copy all records from record file into text file
125     while ( !readFromFile.eof() )
126     {
127        // write single record to text file
128 if ( client.getAccountNumber() != 0 ) // skip empty records
129           outputLine( outPrintFile, client );
130
131        // read next record from record file
132   readFromFile.read( reinterpret_cast< char * >( &client ),
133           sizeof( ClientData ) );
134     } // end while
135  } // end function createTextFile
136
137  // update balance in record
138  void updateRecord( fstream &updateFile )
139  {
140     // obtain number of account to update
141     int accountNumber = getAccount( "Enter account to
```

```
update" );
142
143     // move file-position pointer to correct record in file
144     updateFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
145
146     // read first record from file
147     ClientData client;
148     updateFile.read( reinterpret_cast< char * >( &client ),
149        sizeof( ClientData ) );
150
151     // update record
152     if ( client.getAccountNumber() != 0 )
153     {
154        outputLine( cout, client ); // display the record
155
156        // request user to specify transaction
157        cout << "\nEnter charge (+) or payment (-): ";
158        double transaction; // charge or payment
159        cin >> transaction;
160
161        // update record balance
162        double oldBalance = client.getBalance();
163        client.setBalance( oldBalance + transaction );
164        outputLine( cout, client ); // display the record
165
166        // move file-position pointer to correct record in
file
167        updateFile.seekp( ( accountNumber - 1 ) * sizeof(
ClientData ) );
168
169        // write updated record over old record in file
170        updateFile.write( reinterpret_cast< const char * >(
&client ),
171           sizeof( ClientData ) );
172     } // end if
173     else // display error if account does not exist
174        cerr << "Account #" << accountNumber
175           << " has no information." << endl;
176  } // end function updateRecord
177
178  // create and insert record
179  void newRecord( fstream &insertInFile )
180  {
181     // obtain number of account to create
182     int accountNumber = getAccount( "Enter new account
number" );
183
184     // move file-position pointer to correct record in file
185     insertInFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
186
187     // read record from file
188     ClientData client;
189     insertInFile.read( reinterpret_cast< char * >( &client
),
190        sizeof( ClientData ) );
191
192     // create record, if record does not previously exist
193     if ( client.getAccountNumber() == 0 )
194     {
195        char lastName[ 15 ];
196        char firstName[ 10 ];
```

```
197         double balance;
198
199         // user enters last name, first name and balance
200         cout << "Enter lastname, firstname, balance\n? ";
201         cin >> setw( 15 ) >> lastName;
202         cin >> setw( 10 ) >> firstName;
203         cin >> balance;
204
205         // use values to populate account values
206         client.setLastName( lastName );
207         client.setFirstName( firstName );
208         client.setBalance( balance );
209         client.setAccountNumber( accountNumber );
210
211         // move file-position pointer to correct record in
file
212         insertInFile.seekp( ( accountNumber - 1 ) * sizeof(
ClientData ) );
213
214         // insert record in file
215         insertInFile.write( reinterpret_cast< const char *
>( &client ),
216            sizeof( ClientData ) );
217      } // end if
218      else // display error if account already exists
219         cerr << "Account #" << accountNumber
220            << " already contains information." << endl;
221   } // end function newRecord
222
223   // delete an existing record
224   void deleteRecord( fstream &deleteFromFile )
225   {
226      // obtain number of account to delete
227      int accountNumber = getAccount( "Enter account to
delete" );
228
229      // move file-position pointer to correct record in file
230      deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
231
232      // read record from file
233      ClientData client;
234      deleteFromFile.read( reinterpret_cast< char * >(
&client ),
235         sizeof( ClientData ) );
236
237      // delete record, if record exists in file
238      if ( client.getAccountNumber() != 0 )
239      {
240         ClientData blankClient; // create blank record
241
242         // move file-position pointer to correct record in
file
243         deleteFromFile.seekp( ( accountNumber - 1 ) *
244            sizeof( ClientData ) );
245
246         // replace existing record with blank record
247         deleteFromFile.write(
248            reinterpret_cast< const char * >( &blankClient ),
249            sizeof( ClientData ) );
250
251         cout << "Account #" << accountNumber << "
deleted.\n";
```

```
252     } // end if
253     else // display error if record does not exist
254        cerr << "Account #" << accountNumber << " is
empty.\n";
255  } // end deleteRecord
256
257  // display single record
258  void outputLine( ostream &output, const ClientData &record
)
259  {
260     output << left << setw( 10 ) <<
record.getAccountNumber()
261        << setw( 16 ) << record.getLastName()
262        << setw( 11 ) << record.getFirstName()
263        << setw( 10 ) << setprecision( 2 ) << right << fixed
264        << showpoint << record.getBalance() << endl;
265  } // end function outputLine
266
267  // obtain account-number value from user
268  int getAccount( const char * const prompt )
269  {
270     int accountNumber;
271
272     // obtain account-number value
273     do
274     {
275        cout << prompt << " (1 - 100): ";
276        cin >> accountNumber;
277     } while ( accountNumber < 1 || accountNumber > 100 );
278
279     return accountNumber;
280  } // end function getAccount
```

Program 4.5: Main transaction-processing program

The program presents a menu driven interface to the user. The choices available to the user are 1-Print, 2-Update, 3-New account, 4- Delete account and 5-End processing. These menu choices are realized through following five options:

Option1: calls function createtextFile to store a formatted list of all account information in a text file called print.txt that may be printed. The function createTextFile takes an fstream object as an argument to be used to input data from the credit.dat file. It invokes istream member function read and uses sequential access to input data from credit.dat. The function outputLine is used to output the data to file print.txt. Note that the createTextFile uses istream member function seekg to ensure that the file-position pointer is at the beginning of the file.

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

1


Account     Last Name      First Name    Balance
29          Tiwari         Nisha         -24.54
33          Ghosh          Soumitra      314.33
37          Singh          Shweta        0.0
88          Sen            Ajay          258.34
96          Jolly          Stellina      34.98
```

Option2 calls updateRecord to update an account. This function updates only an existing record, so the function first determines whether the specified record is empty. Lines 128-129 read data into object client, using istream member function read. Then line 132 compares the value returned by getAccountNumber of the client object to zero to determine whether the record contains information. If this value is zero, lines 154-155 print an error message indicating that the record is empty. If the record contains information, line 134 displays the record, using function outputLine, line 139 inputs the transaction amount and lines 142-151 calculate the new balance and rewrite the record to the file. A typical output for option 2 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

2


Enter account to update (1 – 100) : 37
37          Singh          Shweta        0.0
Enter charge (+) or payment (-): +87.9988
37          Singh          Shweta        87.99
```

Option3 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). A typical output for option 3 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

3


Enter new account number (1 – 100) : 22

Enter lastname, firstname, balance

?          Popli         Sukanya      247.45
```

Option4 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 (blank-Client) to the file. Line 231 displays a message to inform the user that the record has been deleted. A typical output for option 4 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

4

Enter account to delete (1 – 100) : 29

Account #29 deleted.
```

Option5 terminating the program.

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

5
```

This transaction-processing system presents an example of a large multi-file program. The program demonstrates use of various features of C++ ranging from classes and methods, constructors, polymorphism, templates and stream I/O capabilities. The program illustrates how C++ language features can be used to solve real world applications by designing and deploying C++ programs.

It would also be in order to discuss here the fact that the sequential files are inappropriate for instant-access applications, in which a particular record must be located immediately. Common instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated teller machines and other kinds of transaction-processing systems that require rapid access to specific data. A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds. This kind of instant access is made possible with random-access files. Individual records of a random-access file can be accessed directly (and quickly) without having to search other records. As we have said, C++ does not impose structure on a file. So the application that wants to use random-access files must create them. A variety of techniques can be used. Perhaps the easiest method is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. Using a database based approach is a common choice for many of these applications.

C++ provides rich set of features for designing various applications to solve various real world problems. A number of useful applications in different domain cab be designed using C++. Applications like passenger reservation systems, automated plant control, restaurant management, library management are some possible applications which can be implemented using C++ features.

4.5 SUMMARY

In the previous chapters, we have discussed various features of C++. C++ provides a rich set of features and capabilities that can be used to write useful programs to solve a number of real world problems. This unit presents a case study of designing and implementing a transaction- processing system in banking domain. The design involves creating necessary data files to store accounts and customer information and then accessing them through suitable code for writing data, appending data, performing credit operations and displaying the results. The program design makes use of various features of C++, including its capability to design multi-file programs. The file processing capability of C++ coupled with the rich I/O capability through stream classes can be used to design many interesting applications. This unit demonstrated use and application of various C++ features for solving one real world large scale problem. C++ can be used to solve many other simple and sophisticated real world problems.

4.6 FURTHER READINGS

1. E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2. P. Deitel and H. Deitel, *C++: How to Program*, PHI, 7th ed, 2010.
3. B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.
4. R. Lafore, *Object Oriented Programming in TURBO C++*, Galgotia Publications, 1994.