Sequential-Access Files

Lab 11: Processing Data in Files

Week 12, Fall 2020

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

Collegiate Programming Exam (CPE)

- **Held in March, June, September, and December. The exam is free of charge.**
- **The exam in December (will be counted as a Lab)**
  - **Registration period**: From 14:25 on **12/08** to 18:00 on **12/18**.
  - **Exam date and time**: From 17:30-21:40 on 12/22.
    - On 12/22, registration time 17:30~17:40, practice time 17:40~18:30, exam time 18:40~21:40. You are not allowed to take the exam if you arrive after 18:00.
  - **Registration web site** https://cpe.cse.nsysu.edu.tw/
    - Must obtain an account before you can sign up for the exam
- **Notes**
  - You will use Coding-Frenzy for coding and evaluating
  - No on-site registration
  - You can cancel a registration before registration due date.
  - You can apply for a leave of absence by filing an application form on CPE web site before the exam. **Otherwise, you are not allowed to take the next exam.**
  - Materials for your reference before an exam can be found at ： http://cpe.cse.nsysu.edu.tw/environment.php
  - There will be a one-star problem which is an easy problem.

Outline

- Introduction
- Data Hierarchy
- Files and streams
- Creating a Sequential File
- Reading Data from a Sequential File
- Updating Sequential Files
- See Chapter 8 for Details

8.1 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.
- Here, we explain how to build C++ programs that create, update and process sequential files.

8.2 Data Hierarchy

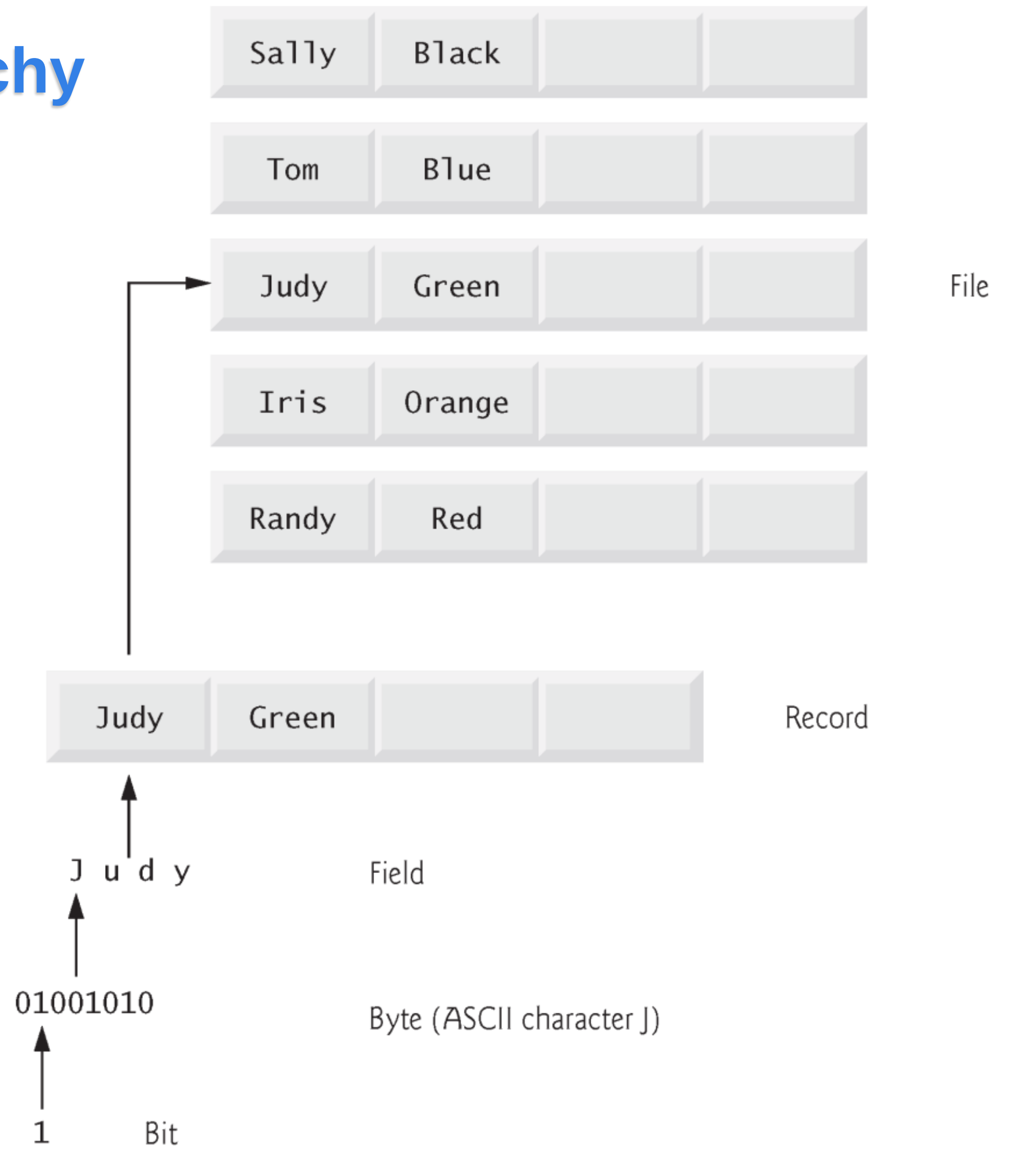


Fig. 8.1 | Data hierarchy.

8.3 Files and Streams

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

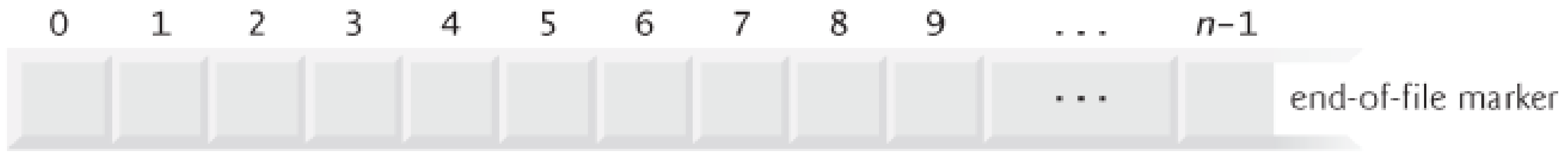


Fig. 8.2 | C++'s view of a file of *n* bytes.

- 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.
- The streams associated with these objects provide communication channels between a program and a particular file or device.

8.3 Files and Streams (cont.)

- 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)

Example of Creating a Sequential File

```
// Fig. 8.3: Fig08_03.cpp
// Create a sequential file.
#include <iostream>
#include <string>
#include <fstream> // file stream
#include <cstdlib>
using namespace std;

int main()
{
    // ofstream constructor opens file
    ofstream outClientFile( "clients.txt", ios::out );

    // exit program if unable to create file
    if ( !outClientFile ) // overloaded ! operator
    {
        cerr << "File could not be opened" << endl;
        exit( 1 );
    } // end if

    cout << "Enter the account, name, and balance." << endl
        << "Enter end-of-file to end input.\n? ";

```

In CodeBlock, here this file is created and located at the same directory of the program that creates this file.

```cpp
   int account; // customer's account number
   string name; //customer's name
   double balance; // amount of money customer owes company

   // read account, name and balance from cin, then place in file
   while ( cin >> account >> name >> balance )
   {
      outClientFile << account << ' ' << name << ' ' << balance << endl;
      cout << "? ";
   } // end while
} // end main
```

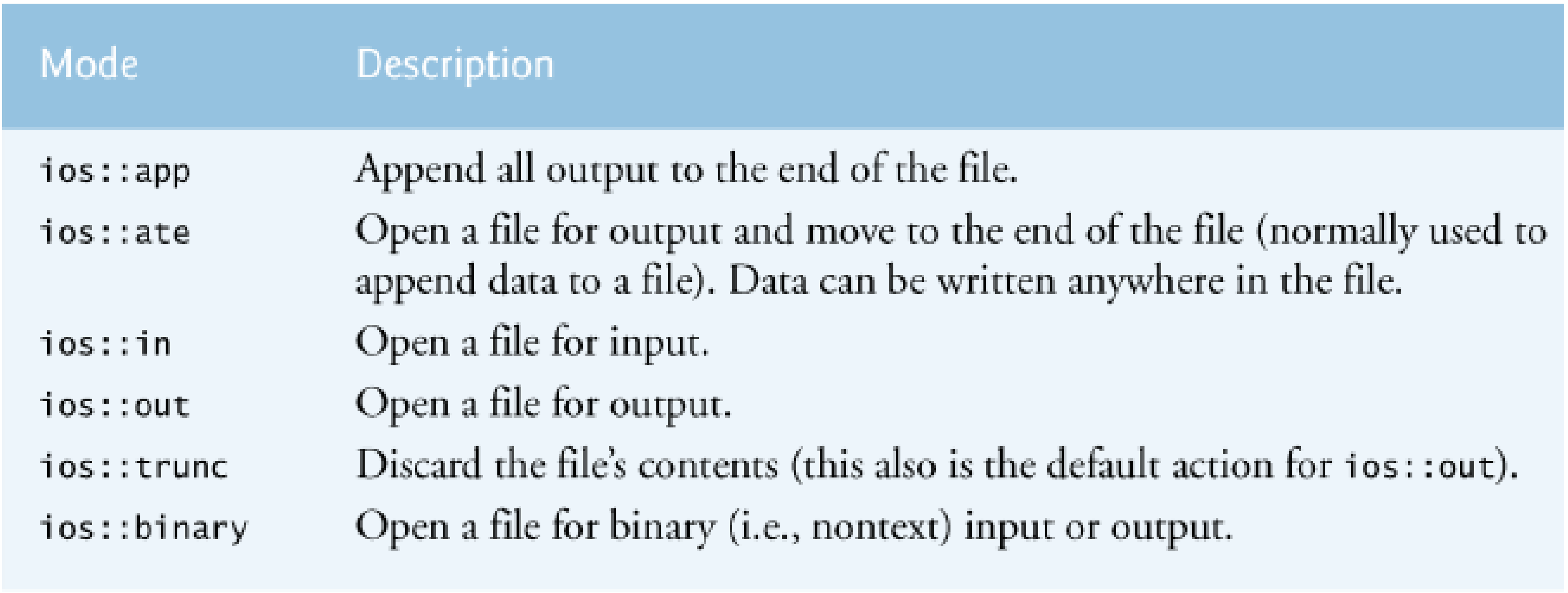
```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 8.3 | Creating a sequential file. (Part 2 of 2.)

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 Times

- 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:

```
const string fileName = “clients.txt”;
outClientFile.open(filename, ios::out);
```

Terminating Input from Keyboard and Closing a File

- Once the user enters the end-of-file indicator (^Z), Input from the keyboard are terminated.
- You can close the `ofstream` object explicitly, using member function `close()` in the statement, for example outClientFile.close().

8.5 Reading Data from a Sequential File

- Creating an `ifstream` object opens a file for input.
- The `ifstream` constructor can receive the filename and the file open mode as arguments.

Note that if a file is opened for input and it does not exist, this will cause an error. Especially, this will occur when you want to create an fstream file for input&output, but the file does not exist.
To solve this problem, you can first create the file with output mode, then close it, and then re-opened it with input&output mode.

```cpp
// Fig. 8.6: Fig08_06.cpp
// Reading and printing a sequential file.
#include <iostream>
#include <fstream> // file stream
#include <iomanip>
#include <string>
#include <cstdlib>
using namespace std;

void outputLine( int, const string, double ); // prototype

int main()
{
   // ifstream constructor opens the file
   ifstream inClientFile( "clients.txt", ios::in );

   // exit program if ifstream could not open file
   if ( !inClientFile )
   {
      cerr << "File could not be opened" << endl;
      exit( 1 );
   } // end if

```

Fig. 8.6 | Reading and printing a sequential file. (Part 1 of 3.)

```cpp
   int account; // customer's account number
   string name; // customer's name
   double balance; //amount of money customer owes company

   cout << left << setw( 10 ) << "Account" << setw( 13 )
      << "Name" << "Balance" << endl << fixed << showpoint;

   // display each record in file
   while ( inClientFile >> account >> name >> balance )
      outputLine( account, name, balance );
} // end main

// display single record from file
void outputLine( int account, const string name, double balance )
{
   cout << left << setw( 10 ) << account << setw( 13 ) << name
      << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
} // end function outputLine
```

Fig. 8.6 | Reading and printing a sequential file. (Part 2 of 3.)

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

Fig. 8.6 | Reading and printing a sequential file. (Part 3 of 3.)

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.

Where to read or write next?

- By a file-position pointer which is associated with reading (or writing).
- It is an integer value that specifies the location in the file as a number of bytes from the file's starting location.
- If a file is opened for input and output, it has two file-position pointers, one for input and the other for output.
- When a record is read or written, its file-position pointer is advanced to the location where the next record starts.
- A file-position pointer can be set to a certain location using seekg member function for an input stream and seekp member function for an output stream.

Reading All the File Data Several Times

- 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.
- To process the file sequentially several times, each time we must reposition a file-position pointer to the beginning of the file. Both `ifstream` and `ofstream` provide member functions for repositioning a file-position pointer.
  - `Seekg(n,dir)` for `ifstream` to set the "**get**" file-position pointer for "**read**"
  - `Seekp(n,dir)` for `ofstream` to set the "**put**" file-position pointer for "**write**"
  - `seekg(n,dir)` and `seekp(n,dir)` for `fstream`
- A file position pointer is modified based on the two arguments **n** and **dir**. The argument **n** normally is a `long` integer.

Modifying a 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** specifies 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`.

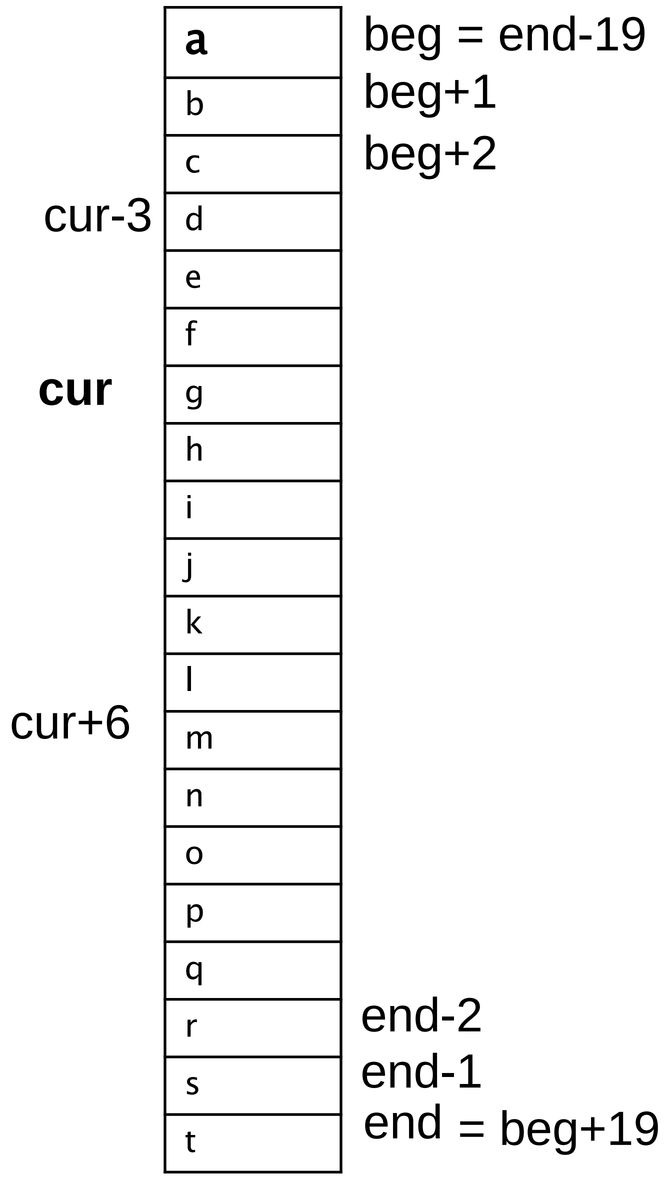
Some examples of Setting File-Position Pointer's Value

- `// 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.
- 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 Processing a File

- Figure 8.7 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)

```
// Fig. 8.7: Fig08_08.cpp
// Credit inquiry program.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <cstdlib>
using namespace std;

enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
int getRequest();
bool shouldDisplay( int, double );
void outputLine( int, const string, double );

int main()
{
   // ifstream constructor opens the file
   ifstream inClientFile( "clients.txt", ios::in );

   // exit program if ifstream could not open file
   if ( !inClientFile )
   {
      cerr << "File could not be opened" << endl;
      exit( 1 );
   } // end if
```

Fig. 8.7 | Credit inquiry program. (Part 1 of 7.)

```

   int request; // request type: zero, credit or debit balance
   int account; // customer's account number
   string name; // customer's name
   double balance; // amount of money customer owes company

   // get user's request (e.g., zero, credit or debit balance)
   request = getRequest();

   // process user's request
   while ( request != END )
   {
      switch ( request )
      {
         case ZERO_BALANCE:
            cout << "\nAccounts with zero balances:\n";
            break;
         case CREDIT_BALANCE:
            cout << "\nAccounts with credit balances:\n";
            break;
         case DEBIT_BALANCE:
            cout << "\nAccounts with debit balances:\n";
            break;
      } // end switch
```

Fig. 8.7 | Credit inquiry program. (Part 2 of 7.)

The statement in line 52 should read a record first and then check **eof** (end of file) in the while loop below. If this statement is omitted, the program will not work correctly.

```
50
51          // read account, name and balance from file
52          inClientFile >> account >> name >> balance;
53
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. 8.7 | Credit inquiry program. (Part 3 of 7.)

```
// obtain request from user
int getRequest()
{
   int request; // request from user

   // display request options
   cout << "\nEnter request" << endl
      << " 1 - List accounts with zero balances" << endl
      << " 2 - List accounts with credit balances" << endl
      << " 3 - List accounts with debit balances" << endl
      << " 4 - End of run" << fixed << showpoint;

   do // input user request
   {
      cout << "\n? ";
      cin >> request;
   } while ( request < ZERO_BALANCE && request > END );

   return request;
} // end function getRequest

```

Fig. 8.7 | Credit inquiry program. (Part 4 of 7.)

```
// determine whether to display given record
bool shouldDisplay( int type, double balance )
{
   // determine whether to display zero balances
   if ( type == ZERO_BALANCE && balance == 0 )
      return true;

   // determine whether to display credit balances
   if ( type == CREDIT_BALANCE && balance < 0 )
      return true;

   // determine whether to display debit balances
   if ( type == DEBIT_BALANCE && balance > 0 )
      return true;

   return false;
} // end function shouldDisplay

```

Fig. 8.7 | Credit inquiry program. (Part 5 of 7.)

```
// display single record from file
void outputLine( int account, const string name, double balance )
{
   cout << left << setw( 10 ) << account << setw( 13 ) << name
      << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
} // end function outputLine
```

```
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. 8.7 | Credit inquiry program. (Part 6 of 7.)

```
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. 8.7 | Credit inquiry program. (Part 7 of 7.)

8.6 Updating Sequential Files

- **Data that is formatted and written to a sequential file as shown in Section 8.4 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.

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 `int`s, which store the same number of “raw data” bytes internally (typically four bytes on today’s popular 32-bit machines). Hence, if they are stored in file with a **binary open mode**, all integers use four bytes.
  - However, these integers become different-sized fields when output as **formatted text** (character sequences, ASCII code for example). It uses one byte to store 7, two bytes to store 14, four bytes for -117, four bytes for 2047 and five bytes for 27383.
  - Therefore, the formatted input/output model usually is not used to update records in place.

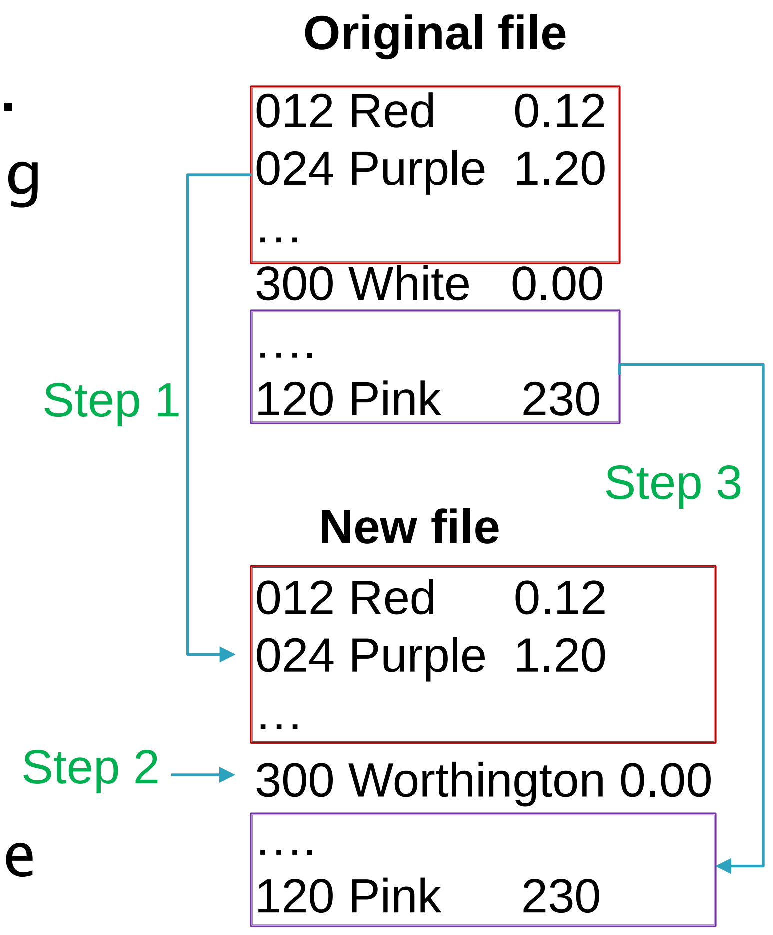
Suggestion for Updating File

- Such updating can be done with formatted I/O, but a bit awkwardly.
- For example, to make the preceding name change

  Step1: the records before **300 White 0.00** in a sequential file could be copied to a new file

  Step2: the updated record then written to the new file

  Step 3: the records after **300 White 0.00** copied to the new file.
- This requires processing *every* record in the file to update *only* one record.
- If many records are being updated in one pass of the file, though, this technique can be acceptable.



Lab 11: File Processing

- You may be given a file name clientFile.txt that consists of some bank account records. Each record has four fields. The first field is firstName, the second field is lastName, the third field is actBalance which is a real number. The fourth field is the date and time when the record is created. You are asked to do the following tasks:

  Step 1: Check whether clientFile.txt exists or not. If it does exist, print the prompt message "Existing file content: " and then display the file content. Otherwise, print the prompt message "Creating a new file: ".

  Step 2: Print the prompt message "Enter first name, last name, and balance: ".

  Step 3: Read firstName, lastName, actBalance from keyboard and form a record. Add this record to the end of this file. Repeat this step until a ctrl ^z is entered from keyboard to stop reading data.

  Step 4: Remove any duplicate records from the file. That is, if any two records have the same firstName and lastName, keep only the one created last.

  Step 5: Display the file content.

Requirements

- The number of records is at most 1000.
- You must write the following function to remove duplicate records:

  void removeDuplicateRecords(fstream &fptr, const string fileN);

  where fileN is the file name associated with fptr.
- You must create the following function for Step 3:

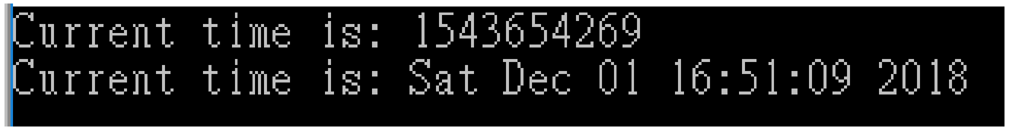
  void createRecord(ofstream &fptr, string firstN, string lastN, double actBal);

  This function stores a record into the file. **fptr** is a file object, **firstN** is first name, **lastN** is last name, and **actBal** is account balance. You should store date and time in the number of seconds passed since 00:00:00 GMT, Jan 1, 1970, to the end of each record.
- You will be given the function void displayFile( ) to print all the records in the file. You should not modify this function.
- Check TA grading notes to understand how your program will be tested.

Time and Date in C++

```
#include <iostream>
#include <ctime>        /* time_t, time, ctime */
using namespace std;
int main ()
{
  time_t currentTime;
  time (&currentTime);
 // get the time and date which is the total number of seconds passed since
00:00:00 GMT, Jan 1, 1970
Cout << “current time is: “ << currentTime << endl;
cout << “Current time is: “ << ctime (&currentTime) << endl;
// ctime converts the number of seconds into ASCII representation of date and
time.
  return 0;
}
```

Below is the output of running the program.



```
Current time is: 1543654269
Current time is: Sat Dec 01 16:51:09 2018
```

void displayFile()

```
//This function prints all the records stored in the file clientFile.txt.
void displayFile()
{
    string str1;
    string str2;
    string fileN = “clientFile.txt”;
    ifstream  fptr.open(fileN);
    double balance;
    int numAct = 0;
    time_t transTime;
    if ( !ftp) {
      cerr << "File could not be opened" << endl;
      exit( 1 );
    } // end if
    while(fptr >> str1 >> str2 >> balance >> transTime) {
        cout << setw(15) << str1  << ' ' << setw(15) << str2 << ' '
        << setw(7) << balance << "     " << ctime(&transTime);
        numAct++;
    }
    cout << "Number of records in the file " << fileN << ": " << numAct << endl;
}
```

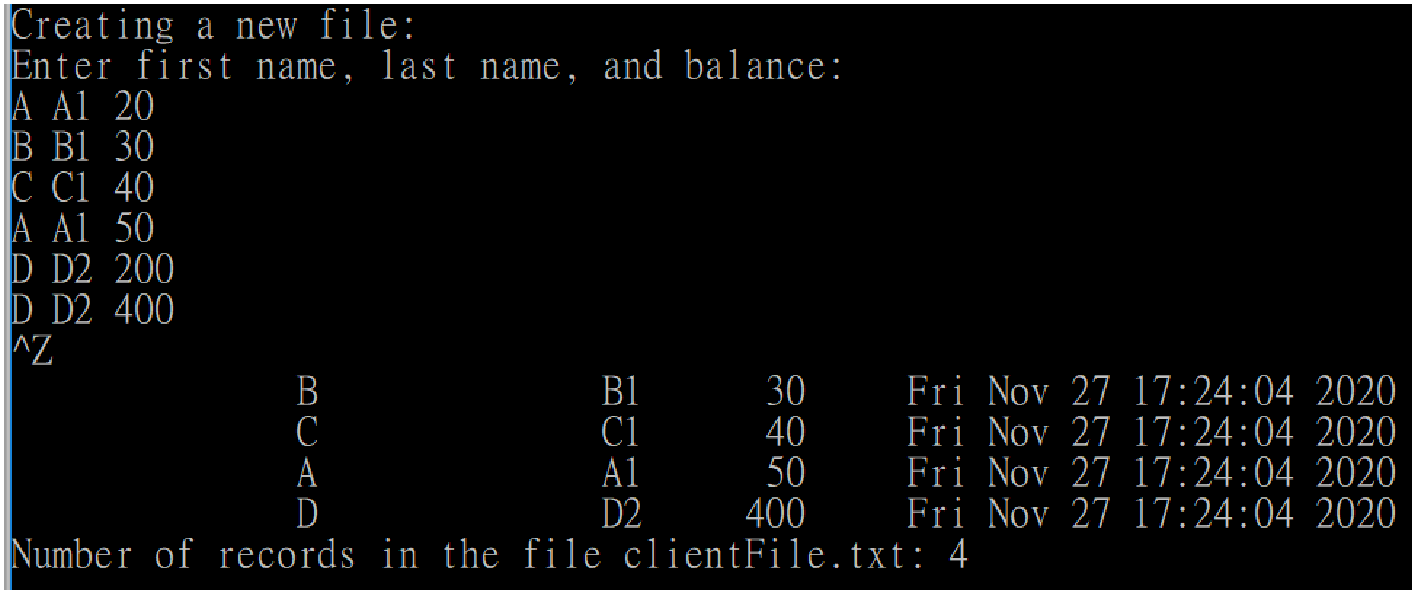
Hints

- You may have to close a file object and then re-open the file object with different file-open mode when removing a record in the file.
- Before checking whether a record for the given first name and last name already exists in the file, you may have to reposition the file-position pointer to the beginning of the file before you do search again.
- You may copy part of the content in clientFile.txt to a temporary file and then copy it back.
- You may use arrays for removing duplicates.
- Note that a file object of fstream has **two** file-position pointers, one for reading and the other for writing.
- A file is opened once. It should have a corresponding close. That is, if fileObject.open(…) is called, there should be a fileObject.close(…).

Input & Output (First Run)

- Each line consists of a record as shown below.

Sample input	Sample output
A A1 20 B B1 30 C C1 40 A A1 50 D D2 200 D D2 400	B B1 30 Fri Nov 27 16:40:33 2020 C C1 40 Fri Nov 27 16:40:38 2020 A A1 50 Fri Nov 27 16:40:44 2020 D D2 400 Fri Nov 27 16:41:09 2020 Number of records in the file clientFile.txt: 4



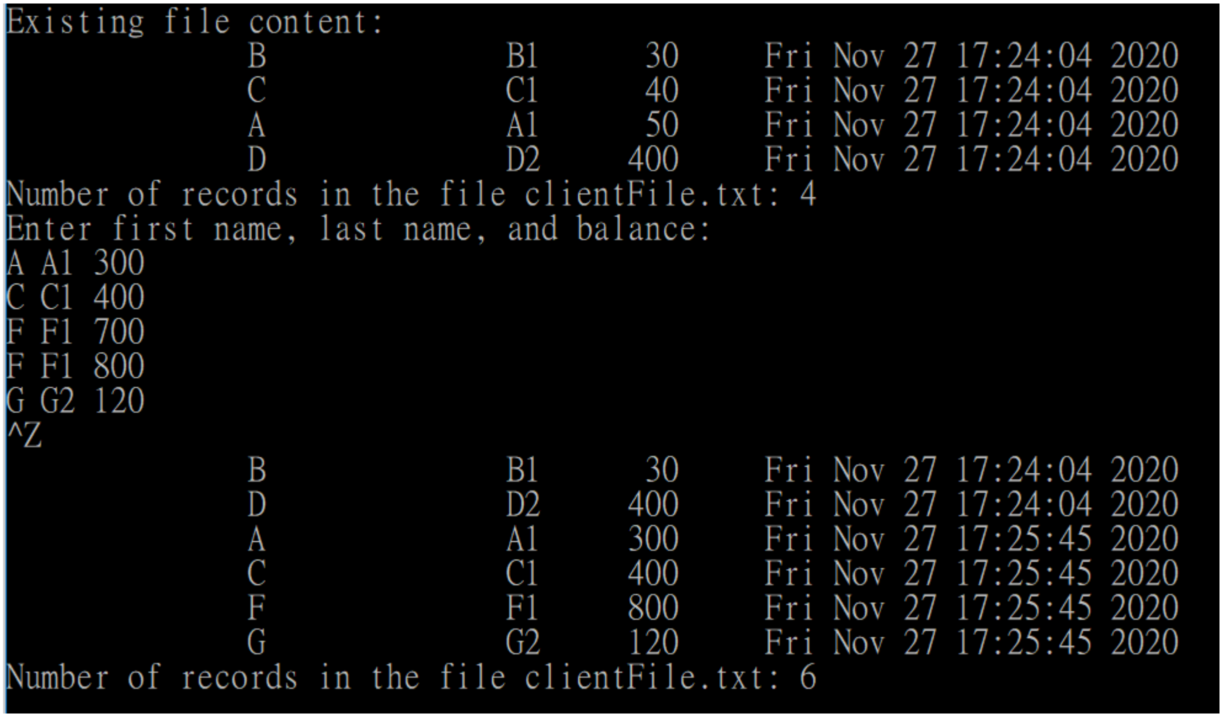
```
Creating a new file:
Enter first name, last name, and balance:
A A1 20
B B1 30
C C1 40
A A1 50
D D2 200
D D2 400
^Z
              B               B1        30      Fri Nov 27 17:24:04 2020
              C               C1        40      Fri Nov 27 17:24:04 2020
              A               A1        50      Fri Nov 27 17:24:04 2020
              D               D2       400      Fri Nov 27 17:24:04 2020
Number of records in the file clientFile.txt: 4
```

(The first two lines are prompt messages. You should print out the same prompt messages.)

Input & Output (Second Run)

This run is conducted with an existing file clientFile.txt.

Sample input	Sample output
A A1 300 C C1 400 F F1 700 F F1 800 G G2 120	B B1 30 Fri Nov 27 16:40:33 2020 D D2 400 Fri Nov 27 16:41:09 2020 A A1 300 Fri Nov 27 17:09:55 2020 C C1 400 Fri Nov 27 17:10:08 2020 F F1 800 Fri Nov 27 17:10:27 2020 G G2 120 Fri Nov 27 17:10:39 2020 Number of records in the file clientFile.txt: 6



```
Existing file content:
              B              B1      30     Fri Nov 27 17:24:04 2020
              C              C1      40     Fri Nov 27 17:24:04 2020
              A              A1      50     Fri Nov 27 17:24:04 2020
              D              D2     400     Fri Nov 27 17:24:04 2020
Number of records in the file clientFile.txt: 4
Enter first name, last name, and balance:
A A1 300
C C1 400
F F1 700
F F1 800
G G2 120
^Z
              B              B1      30     Fri Nov 27 17:24:04 2020
              D              D2     400     Fri Nov 27 17:24:04 2020
              A              A1     300     Fri Nov 27 17:25:45 2020
              C              C1     400     Fri Nov 27 17:25:45 2020
              F              F1     800     Fri Nov 27 17:25:45 2020
              G              G2     120     Fri Nov 27 17:25:45 2020
Number of records in the file clientFile.txt: 6
```

TA Grading Notes (1)

- **To test a program, ask students to run the program twice. Before running the program, remove clientFile.txt from work library.**
  - The first time runs the program with the first part of public test cases. This will create clientFile.txt.
  - The second time runs the program with the file created in the first time and the second part of public test cases.
- **Should have the following statement in the main() function:**

  createRecord(ofstream &, const char *, string, string, double);

  This function should be contained in a loop.

TA Grading Notes (2)

- Should also have the following statement in the main() function:
  removeDuplicateRecords(fstream&, const string)
  This function should NOT be contained in a loop.
- Ask students to explain how removing duplicate records is done.
- Should have at least a call to displayFile(). This function should not be modified.
- Should make sure that the duplicate records are correctly removed.