EE1003
Introduction to Computer I

# Chapter 8
# Sequential-Access Files

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw



## Learning Objectives

In this chapter you'll learn:

- The data hierarchy from bits, to files to databases.
- To create, read, write and update sequential files.
- Some of the key streams that are associated with file processing.

# Outline





# 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.
- In this chapter, we explain how to build C++ programs that create, update and process sequential files.
- We examine techniques for input of data from, and output of data to, `string` streams rather than files in Chapter 18, Class `string` and String Stream Processing.

# 8.2 Data Hierarchy

- Ultimately, all data items that digital computers process are reduced to combinations of zeros and ones.
- The smallest data item that computers support is called a bit
  - Each data item, or bit, can assume either the value 0 or the value 1.
- The decimal digits (0–9), letters (A–Z and a–z) and special symbols (e.g., $, @, %, &, *, …) are referred to as characters.
  - People create programs and data items with characters.
- Every character in a computer's character set is represented as a pattern of 1s and 0s.
  - Computers manipulate and process these characters as patterns of bits.
- Bytes are composed of eight bits.



# 8.2 Data Hierarchy

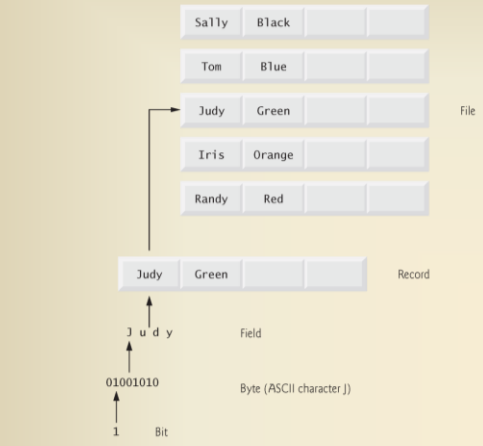


**Fig. 8.1** | Data hierarchy.

## 8.2 Data Hierarchy

- Just as characters are composed of bits, fields are composed of characters.
- A field is a group of characters that conveys some meaning.
  - For example, a field can represent a person's name.
- Typically, a record is composed of several fields
  - Thus, a record is a group of related fields.
  - Can be represented as a `class` in C++.
- A file is a group of related records.
- To facilitate retrieving specific records from a file, at least one field in each record is chosen as a record key.



## 8.2 Data Hierarchy

- There are many ways of organizing records in a file.
- A common type of organization is called a sequential file, in which records typically are stored in order by a record-key field.
- Most businesses use many different files to store data.
- A group of related files often are stored in a database.
- A collection of programs designed to create and manage databases is called a database management system (DBMS).

# 8.3 Files and Streams

- C++ views each file as a sequence of bytes.
- Each file ends either with an end-of-file marker or at a specific byte number recorded in operating.
- When a file is opened, an object is created, and a stream is associated with the object.
- The streams associated with these objects provide communication channels between a program and a particular file or device.


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



# 8.4 Creating a Sequential File

- C++ imposes no structure on a file.
  - You must structure files to meet the application's requirements.
- Figure 8.3 creates a sequential file for an accounts-receivable system to manage the money owed by a company's clients.
- To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included.
- For each client, the program obtains the client's account number, name and balance (i.e., the amount the client).
- Each obtained data constitutes a record for that client.
- The account number serves as the record key.
- This program assumes that the records are in account no. order.
  - In a comprehensive accounts receivable system, a sorting capability would be provided to eliminate this restriction.

# 8.4 Creating a Sequential File

```
1  // Fig. 8.3: Fig08_03.cpp
2  // Create a sequential file.
3  #include <iostream>
4  #include <string>
5  #include <fstream> // file stream
6  #include <cstdlib>
7  using namespace std;
8
9  int main()
10 {
11    // ofstream constructor opens file
12    ofstream outClientFile( "clients.txt", ios::out );
13
14    // exit program if unable to create file
15    if ( !outClientFile ) // overloaded ! operator
16    {
17       cerr << "File could not be opened" << endl;
18       exit( 1 );
19    } // end if
20
21    cout << "Enter the account, name, and balance." << endl
22       << "Enter end-of-file to end input.\n? ";
23
```

**Fig. 8.3** | Creating a sequential file. (Part 1 of 2.)



# 8.4 Creating a Sequential File

```
24    int account; // customer's account number
25    string name; //customer's name
26    double balance; // amount of money customer owes company
27
28    // read account, name and balance from cin, then place in file
29    while ( cin >> account >> name >> balance )
30    {
31       outClientFile << account << ' ' << name << ' ' << balance << endl;
32       cout << "? ";
33    } // end while
34 } // end main
```

```
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.)

# 8.4 Creating a Sequential File

- In Fig. 8.3, the file is to be opened for output, so an `ofstream` object is created.
  - This establishes a "line of communication" with the file.
  - By default, `ofstream` objects are opened for output.
- 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
- If the specified file does not yet exist, then the `ofstream` object creates the file, using that filename.
- Existing files opened with mode `ios::out` are truncated
  - All data in the file is discarded without warning.



# 8.4 Creating a Sequential File

| 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.

**Common Programming Error 8.1**

*Use caution when opening an existing file for output (`ios::out`), especially when you want to preserve the file's contents, which will be discarded without warning.*

**Good Programming Practice 8.1**

*Open a file for input only (using `ios::in`) if the file's contents should not be modified. This prevents unintentional modification of the file's contents and is an example of the principle of least privilege.*

## 8.4 Creating a Sequential File

- An `ofstream` object can be created without opening a specific file—a file can be attached to the object later.
- For example, the statement
  - `ofstream outClientFile;`
- creates an `ofstream` object named `outClientFile`.
- The `ofstream` member function `open` opens a file and attaches it to an existing `ofstream` object as follows:
  - `outClientFile.open("clients.txt", ios::out);`

**Common Programming Error 8.2**

*Not opening a file before attempting to reference it in a program will result in an error.*



## 8.4 Creating a Sequential File

- After creating an `ofstream` object, the program tests whether the open operation was successful before using it.
- The condition in the `if` statement in lines 15–19 returns true if the `open` operation failed.
- Some possible errors are
  - attempting to open a nonexistent file for reading,
  - attempting to open a file for reading or writing without permission,
  - opening a file for writing when no disk space is available.
- Function `exit` terminates a program.
  - The argument to `exit` is returned to the environment from which the program was invoked. (0: normally, others: error)
  - The calling environment (most likely the operating system) uses the value returned by `exit` to respond appropriately to the error.

# 8.4 Creating a Sequential File

- The `while` statement of lines 29–33 inputs each set of data from the keyboard.
- The user enters the end-of-file key combination to inform the program to process no additional.
- When the end-of-file indicator is set, the `while` condition becomes false terminating the `while` statement.

| Computer system | Keyboard combination |
|---|---|
| UNIX/Linux/Mac OS X | <*Ctrl-d*> (on a line by itself) |
| Microsoft Windows | <*Ctrl-z*> (sometimes followed by pressing *Enter*) |
| VAX (VMS) | <*Ctrl-z*> |

**Fig. 8.5** | End-of-file key combinations for various popular computer systems.



# 8.4 Creating a Sequential File

- Line 31 writes a set of data to the file `clients.txt`, using the stream insertion operator << and the `outClientFile` object associated with the file.
  - Similar to using `cout`, but replace `cout` by `outClientFile`.
  - The data may be retrieved by a program designed to read the file.
- The file created in Fig. 8.3 is simply a text file, so it can be viewed by any text editor.
- Once the user enters the end-of-file indicator, `main` terminates.
- This implicitly invokes `outClientFile`'s destructor, which closes the `clients.txt` file.
- You also can close the `ofstream` object explicitly, using member function close in the statement.

# 8.5 Reading Data from a Sequential File

- Files store data so it may be retrieved for processing when needed.
- Figure 8.6 reads records from the `clients.txt` file and displays the contents of these records.
- Creating an `ifstream` object opens a file for input.
  - It can receive the filename and the file open mode as arguments.
- Line 15 creates an `ifstream` object called `inClientFile` and associates it with the `clients.txt` file.
- The arguments in parentheses are passed to the `ifstream` constructor function, which opens the file and establishes a "line of communication" with the file.



# 8.5 Reading Data from a Sequential File

```
1  // Fig. 8.6: Fig08_06.cpp
2  // Reading and printing a sequential file.
3  #include <iostream>
4  #include <fstream> // file stream
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 void outputLine( int, const string, double ); // prototype
11
12 int main()
13 {
14    // ifstream constructor opens the file
15    ifstream inClientFile( "clients.txt", ios::in );
16
17    // exit program if ifstream could not open file
18    if ( !inClientFile )
19    {
20       cerr << "File could not be opened" << endl;
21       exit( 1 );
22    } // end if
23
```

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

# 8.5 Reading Data from a Sequential File

```
24    int account; // customer's account number
25    string name; // customer's name
26    double balance; //amount of money customer owes company
27
28    cout << left << setw( 10 ) << "Account" << setw( 13 )
29       << "Name" << "Balance" << endl << fixed << showpoint;
30
31    // display each record in file
32    while ( inClientFile >> account >> name >> balance )
33       outputLine( account, name, balance );
34 } // end main
35
36 // display single record from file
37 void outputLine( int account, const string name, double balance )
38 {
39    cout << left << setw( 10 ) << account << setw( 13 ) << name
40       << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
41 } // end function outputLine
```

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



# 8.5 Reading Data from a Sequential File

- Objects of class `ifstream` are opened for input by default, so we could use the statement
  - `ifstream inClientFile( "clients.txt" );`
- An `ifstream` object can also be created without opening a specific file
  - A file can be attached to it later.
- Each time line 32 executes, it reads another record from the file into the variables `account`, `name` and `balance`.
  - Similar to using `cin`, but replace `cin` by `inClientFile`.
- When the end of file has been reached, the `while` condition returns `false` and terminates the `while` statement and the program.

## 8.5 Reading Data from a Sequential File

- To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found.
- It might be necessary to process the file sequentially several times (from the beginning of the file) during program execution.
- Both `istream` and `ostream` provide member functions for repositioning the file-position pointer (the byte number of the next byte in the file to be read or written).
  - seekg (" seek get") for `istream`
  - seekp (" seek put") for `ostream`



## 8.5 Reading Data from a Sequential File

- Each `istream/ostream` object has a "pointer" to indicate the location (in byte) of next access in the file.
  - The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location.
- The statement `inClientFile.seekg( 0 );` repositions the file-position pointer to the beginning of the file (location 0) attached to `inClientFile`.
- A second argument can be used to indicate the seek direction
  - 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,
  - ios::end for positioning relative to the end of a stream

## 8.5 Reading Data from a Sequential File

- Some examples of positioning the "get" file-position pointer are

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );
// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );
// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

- The same operations can be performed using `ostream` member function `seekp`.



## 8.5 Reading Data from a Sequential File

- Member functions tellg and tellp are provided to return the current locations of the "get" and "put" pointers, respectively.
- 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)

# 8.5 Reading Data from a Sequential File

```
1  // Fig. 8.7: Fig08_08.cpp
2  // Credit inquiry program.
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
11 int getRequest();
12 bool shouldDisplay( int, double );
13 void outputLine( int, const string, double );
14
15 int main()
16 {
17    // ifstream constructor opens the file
18    ifstream inClientFile( "clients.txt", ios::in );
19
20    // exit program if ifstream could not open file
21    if ( !inClientFile )
22    {
23       cerr << "File could not be opened" << endl;
24       exit( 1 );
25    } // end if
```

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



# 8.5 Reading Data from a Sequential File

```
26
27    int request; // request type: zero, credit or debit balance
28    int account; // customer's account number
29    string name; // customer's name
30    double balance; // amount of money customer owes company
31
32    // get user's request (e.g., zero, credit or debit balance)
33    request = getRequest();
34
35    // process user's request
36    while ( request != END )
37    {
38       switch ( request )
39       {
40          case ZERO_BALANCE:
41             cout << "\nAccounts with zero balances:\n";
42             break;
43          case CREDIT_BALANCE:
44             cout << "\nAccounts with credit balances:\n";
45             break;
46          case DEBIT_BALANCE:
47             cout << "\nAccounts with debit balances:\n";
48             break;
49       } // end switch
```

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

# 8.5 Reading Data from a Sequential File

```
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.)



# 8.5 Reading Data from a Sequential File

```
73 // obtain request from user
74 int getRequest()
75 {
76    int request; // request from user
77
78    // display request options
79    cout << "\nEnter request" << endl
80       << " 1 - List accounts with zero balances" << endl
81       << " 2 - List accounts with credit balances" << endl
82       << " 3 - List accounts with debit balances" << endl
83       << " 4 - End of run" << fixed << showpoint;
84
85    do // input user request
86    {
87       cout << "\n? ";
88       cin >> request;
89    } while ( request < ZERO_BALANCE && request > END );
90
91    return request;
92 } // end function getRequest
93
```

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

# 8.5 Reading Data from a Sequential File

```
94  // determine whether to display given record
95  bool shouldDisplay( int type, double balance )
96  {
97     // determine whether to display zero balances
98     if ( type == ZERO_BALANCE && balance == 0 )
99        return true;
100
101    // determine whether to display credit balances
102    if ( type == CREDIT_BALANCE && balance < 0 )
103       return true;
104
105    // determine whether to display debit balances
106    if ( type == DEBIT_BALANCE && balance > 0 )
107       return true;
108
109    return false;
110 } // end function shouldDisplay
111
```

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



# 8.5 Reading Data from a Sequential

```
112 // display single record from file
113 void outputLine( int account, const string name, double balance )
114 {
115    cout << left << setw( 10 ) << account << setw( 13 ) << name
116       << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
117 } // end function outputLine
```

```
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.)

# 8.5 Reading Data from a Sequential File

```
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 written to a sequential file as shown in Section 8.4 cannot be modified without the risk of destroying other data.
- For example, 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` (contains six more characters)
- The characters beyond the second "`o`" in "`Worthington`" would overwrite the beginning of the next sequential record.
- The problem is that fields—and hence records—can vary in size.
  - For example, values 7, 14, –117, 2074, and 27383 are all `int`s.
  - However, these integers become different-sized fields when output as formatted text (character sequences).

# 8.6 Updating Sequential Files

- Such updating can be done, but a bit awkwardly.
- For example, to make the preceding name change
  - the records before `300 White 0.00` in a sequential file could be copied to a new file
  - the updated record then written to the new file
  - and the records after `300 White 0.00` copied to the new file.
- This requires processing *every* record in the file to update *only* one record.
- If many records are being updated in one pass of the file, though, this technique can be acceptable.



# 8.6 Updating Sequential Files

- Another approach is fixing the field width in advance to leave enough space for future updating.
- For example, we can set the maximum length of *name* is 11
  - `outClientFile << account << ' ' << setw(11) << name << ' ' << balance <<endl;`
  - Original File: `300 White 0.00`
  - Updated File: `300 Worthington 0.00`
  - Since the total length of this record (including space) is not changed, this update will not affect the following records.
- Although this approach does not damage other data in the same file, more redundant spaces will make the file larger.
  - Length checking is also required on the input data.

# Summary

- Data Hierarchy
- Creating a Sequential File
- Reading Data from a Sequential File
- Updating Sequential Files





**Andy, Yu-Guang Chen**

**Assistant Professor, Department of EE, NCU**

**Email: andyygchen@ee.ncu.edu.tw**