Chapter 8
Sequential-Access Files

C++ How to Program,
Late Objects Version, 7/e

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.

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.
  - It's simple and economical to build electronic devices that can assume two stable states—one state represents 0 and the other represents 1.
- The smallest data item that computers support is called a bit
  - Short for "binary digit"—a digit that can assume one of two values
  - Each data item, or bit, can assume either the value 0 or the value 1.
- Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from 1 to 0 or from 0 to 1).

8.2 Data Hierarchy (cont.)

- Programming with data in the low-level form of bits is cumbersome.
- It's preferable to program with data in forms such as decimal digits (0–9), letters (A–Z and a–z) and special symbols (e.g., $, @, %, &, * and many others).
- Digits, letters and special symbols are referred to as characters.
- The set of all characters used to write programs and represent data items on a particular computer is called that computer's character set.
- Every character in a computer's character set is represented as a pattern of 1s and 0s.
- Bytes are composed of eight bits.

8.2 Data Hierarchy (cont.)

- You create programs and data items with characters; computers manipulate and process these characters as patterns of bits.
- Each `char` typically occupies one byte.
- C++ also provides data type `wchar_t`, which can occupy more than one byte
  - to support larger character sets, such as the Unicode® character set; for more information on Unicode®, visit `www.unicode.org`

8.2 Data Hierarchy (cont.)

- 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 consisting of uppercase and lowercase letters can represent a person's name.
- Data items processed by computers form a data hierarchy (Fig. 8.1), in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and to larger data aggregates.

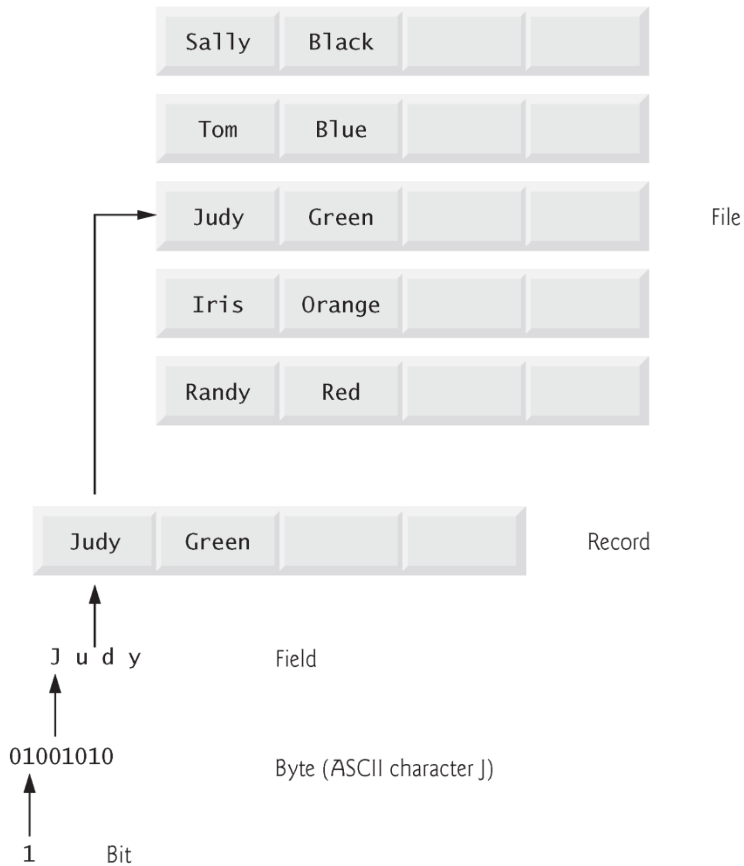

Fig. 8.1 | Data hierarchy.

8.2 Data Hierarchy (cont.)

- Typically, a record (which can be represented as a `class` in C++) is composed of several fields (called data members in C++).
  - Thus, a record is a group of related fields.
- 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.
- A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all others.

8.2 Data Hierarchy (cont.)

- 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 (Fig. 8.2).
- Each file ends either with an end-of-file marker or at a specific byte number recorded in an operating-system-maintained, administrative data structure.
- When a file is opened, an object is created, and a stream is associated with the object.
- 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.

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

8.3 Files and Streams (cont.)

- To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included.

8.4 Creating a Sequential File

- C++ imposes no structure on a file.
- Thus, a concept like that of a “record” does not exist in a C++ file.
- You must structure files to meet the application’s requirements.
- Figure 8.3 creates a sequential file that might be used in an accounts-receivable system to help manage the money owed by a company’s credit clients.
- For each client, the program obtains the client’s account number, name and balance (i.e., the amount the client owes the company for goods and services received in the past).
- The data obtained for each client constitutes a record for that client.
- The account number serves as the record key.
- This program assumes the user enters the records in account number order.
  - In a comprehensive accounts receivable system, a sorting capability would be provided to eliminate this restriction.

```cpp
// 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? ";

```

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

```
    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.)

8.4 Creating a Sequential File (cont.)

- In Fig. 8.3, the file is to be opened for output, so an `ofstream` object is created.
- Two arguments are passed to the object's constructor—the filename and the file-open mode (line 12).
- For an `ofstream` object, the file-open mode can be either `ios::out` to output data to a file or `ios::app` to append data to the end of a file (without modifying any data already in the file).
- Existing files opened with mode `ios::out` are truncated—all data in the file is discarded.
- If the specified file does not yet exist, then the `ofstream` object creates the file, using that filename.
- The `ofstream` constructor opens the file—this establishes a "line of communication" with the file.
- By default, `ofstream` objects are opened for output, so the open mode is not required in the constructor call.
- Figure 8.4 lists the 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.*

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.

8.4 Creating a Sequential File (cont.)

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

- After creating an `ofstream` object and attempting to open it, the program tests whether the open operation was successful.
- 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, and
  - opening a file for writing when no disk space is available.

8.4 Creating a Sequential File (cont.)

- Function `exit` terminates a program.
  - The argument to `exit` is returned to the environment from which the program was invoked.
  - Argument `0` indicates that the program terminated normally; any other value indicates that the program terminated due to an error.
  - The calling environment (most likely the operating system) uses the value returned by `exit` to respond appropriately to the error.

8.4 Creating a Sequential File (cont.)

- 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 data—this sets the “end-of-file indicator” in the `cin` object.
- When the end-of-file indicator is set, the `while` condition becomes false terminating the `while` statement.
- Figure 8.5 lists the keyboard combinations for entering end-of-file for various computer systems.
- Later in the chapter, we’ll use the eof member function to test for end-of-file in an input file.

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 (cont.)

- Line 31 writes a set of data to the file `clients.txt`, using the stream insertion operator `<<` and the `outClientFile` object associated with the file at the beginning of the program.
- The data may be retrieved by a program designed to read the file (see Section 8.5).
- The file created in Fig. 8.3 is simply a text file, so it can be viewed by any text editor.

8.4 Creating a Sequential File (cont.)

- Once the user enters the end-of-file indicator, `main` terminates.
- This implicitly invokes `outClientFile`'s destructor, which closes the `clients.txt` file.
- You also can close the `ofstream` object explicitly, using member function `close` in the statement

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.*

8.5 Reading Data from a Sequential File

- Files store data so it may be retrieved for processing when needed.
- In this section, we discuss how to read data sequentially from a file.
- Figure 8.6 reads records from the `clients.txt` file that we created using the program of Fig. 8.3 and displays the contents of these records.

8.5 Reading Data from a Sequential File (cont.)

- Creating an `ifstream` object opens a file for input.
- The `ifstream` constructor can receive the filename and the file open mode as arguments.
- Line 15 creates an `ifstream` object called `inClientFile` and associates it with the `clients.txt` file.
- The arguments in parentheses are passed to the `ifstream` constructor function, which opens the file and establishes a "line of communication" with the file.

```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.)

```
   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.)

8.5 Reading Data from a Sequential File (cont.)

- Objects of class `ifstream` are 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 without opening a specific file, because 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`.
- 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 run, closing the file.

8.5 Reading Data from a Sequential File (cont.)

- To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found.
- It might be necessary to process the file sequentially several times (from the beginning of the file) during 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 (cont.)

- Each `istream` object has a "get pointer," which indicates the byte number in the file from which the next input is to occur, and each `ostream` object has a "put pointer," which indicates the byte number in the file at which the next output should be placed.
- The statement
  - `inClientFile.seekg( 0 );`

  repositions the file-position pointer to the beginning of the file (location `0`) attached to `inClientFile`.
- The argument to `seekg` normally is a `long` integer.

8.5 Reading Data from a Sequential File (cont.)

- A second argument can be specified to indicate the seek direction, which can be
  - `ios::beg` (the default) for positioning relative to the beginning of a stream,
  - `ios::cur` for positioning relative to the current position in a stream or
  - `ios::end` for positioning relative to the end of a stream
- The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location (this is also referred to as the offset from the beginning of the file).

8.5 Reading Data from a Sequential File (cont.)

- 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 (cont.)

- 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)

```
// 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.)

```
      // read account, name and balance from file
      inClientFile >> account >> name >> balance;

      // display file contents (until eof)
      while ( !inClientFile.eof() )
      {
         // display record
         if ( shouldDisplay( request, balance ) )
            outputLine( account, name, balance );

         // read account, name and balance from file
         inClientFile >> account >> name >> balance;
      } // end inner while

      inClientFile.clear(); // reset eof for next input
      inClientFile.seekg( 0 ); // reposition to beginning of file
      request = getRequest(); // get additional request from user
   } // end outer while

   cout << "End of run." << endl;
} // end main
```

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.)

```cpp
// 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.

8.6 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).
  - However, these integers become different-sized fields when output as formatted text (character sequences).
  - Therefore, the formatted input/output model usually is not used to update records in place.

8.6 Updating Sequential Files (cont.)

- 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.