17 File Processing

I read part of it all the way through.

—Samuel Goldwyn

I can only assume that a "Do Not File" document is filed in a "Do Not File" file.

—Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975

A great memory does not make a philosopher, any more than a dictionary can be called grammar.

—John Henry, Cardinal Newman

OBJECTIVES

In this chapter you will learn:

- To create, read, write and update files.
- Sequential file processing.
- Random-access file processing.
- To use high-performance unformatted I/O operations.
- The differences between formatted-data and raw-data file processing.
- To build a transaction-processing program using random-access file processing.

Outline

17.1 Introduction

- **Files**
  - **Used for data persistence**
    - **Permanent retention of large amounts of data**
  - **Stored on secondary storage devices**
    - **Magnetic disks**
    - **Optical disks**
    - **Tapes**

17.2 The Data Hierarchy

- Bits (“binary digits”)
  - Can assume one of two values, `0` or `1`
  - Smallest data item that computers support
    - Computer circuitry performs simple bit manipulations
    - Ultimately all data items are composed of bits
- Characters
  - Include decimal digits, letters and special symbols
    - Composed of bits (`1`s and `0`s)
  - A character set is the set of all characters used on a particular computer
  - `char`s are stored in bytes (8 bits)
  - `wchar_t`s are stored in more than one byte

17.2 The Data Hierarchy (Cont.)

- Fields
  - Composed of characters
  - Conveys some meaning
  - Example
    - Uppercase and lowercase letters can represent a name
- Records
  - Composed of several fields
  - Represented as a class in C++
  - Example
    - An employee’s record might include id#, name, address, etc.
  - A record key is a field unique to each record

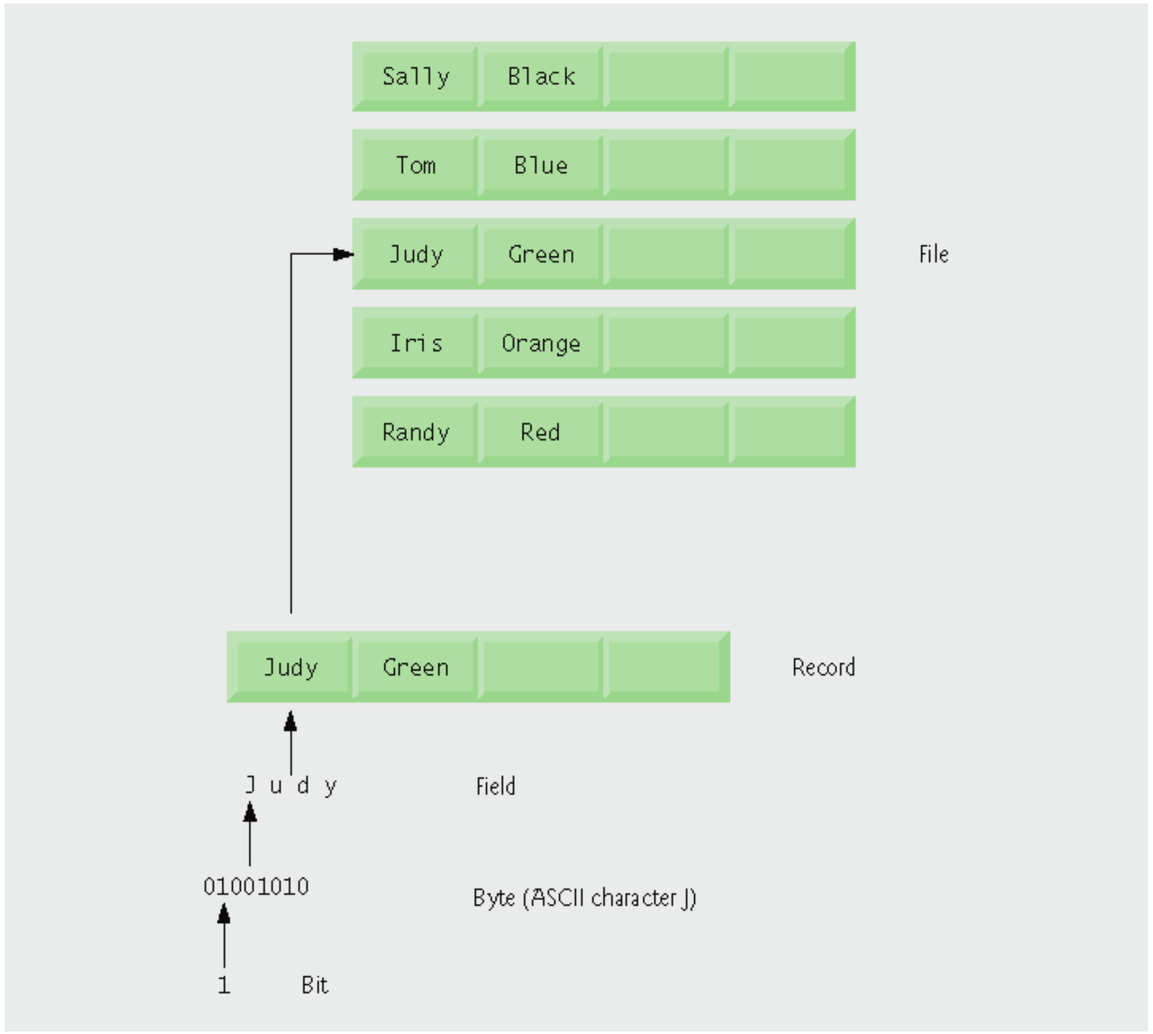

Fig. 17.1 | **Data hierarchy.**

17.2 The Data Hierarchy (Cont.)

- File
  - Composed of a group of related records
  - Example
    - A payroll file containing one record for each employee
  - Many ways to organize records in a file
    - Such as a sequential file
      - Records are stored in order by a record-key field
- Database
  - Composed of a group of related files
  - Managed by a collection of programs called a database management system (DBMS)

17.3 Files and Streams

- **Files**
  - Viewed by C++ as a sequence of bytes
  - Ends either with an end-of-file marker or at a system-recorded byte number
  - Communication between a program and a file is performed through stream objects
    - `<fstream>` header file
      - Stream class templates
        - `basic_ifstream` – for file input
        - `basic_ofstream` – for file output
        - `basic_fstream` – for file input and output
      - Files are opened by creating objects of stream template specializations

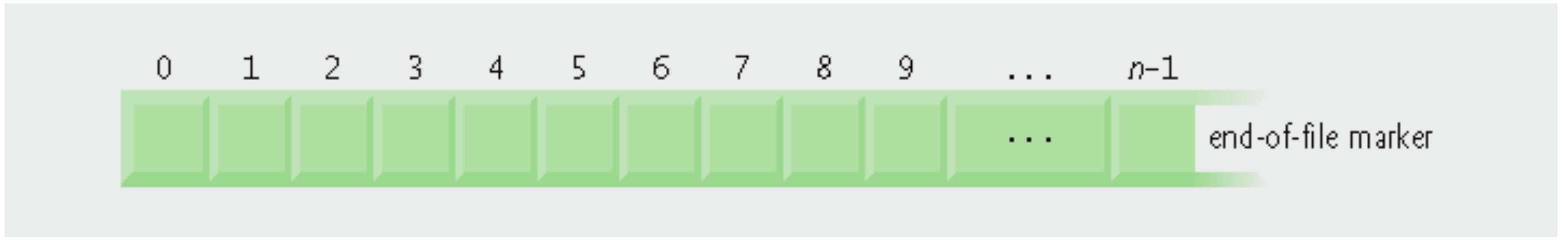

Fig. 17.2 | C++' s view of a file of *n* bytes.

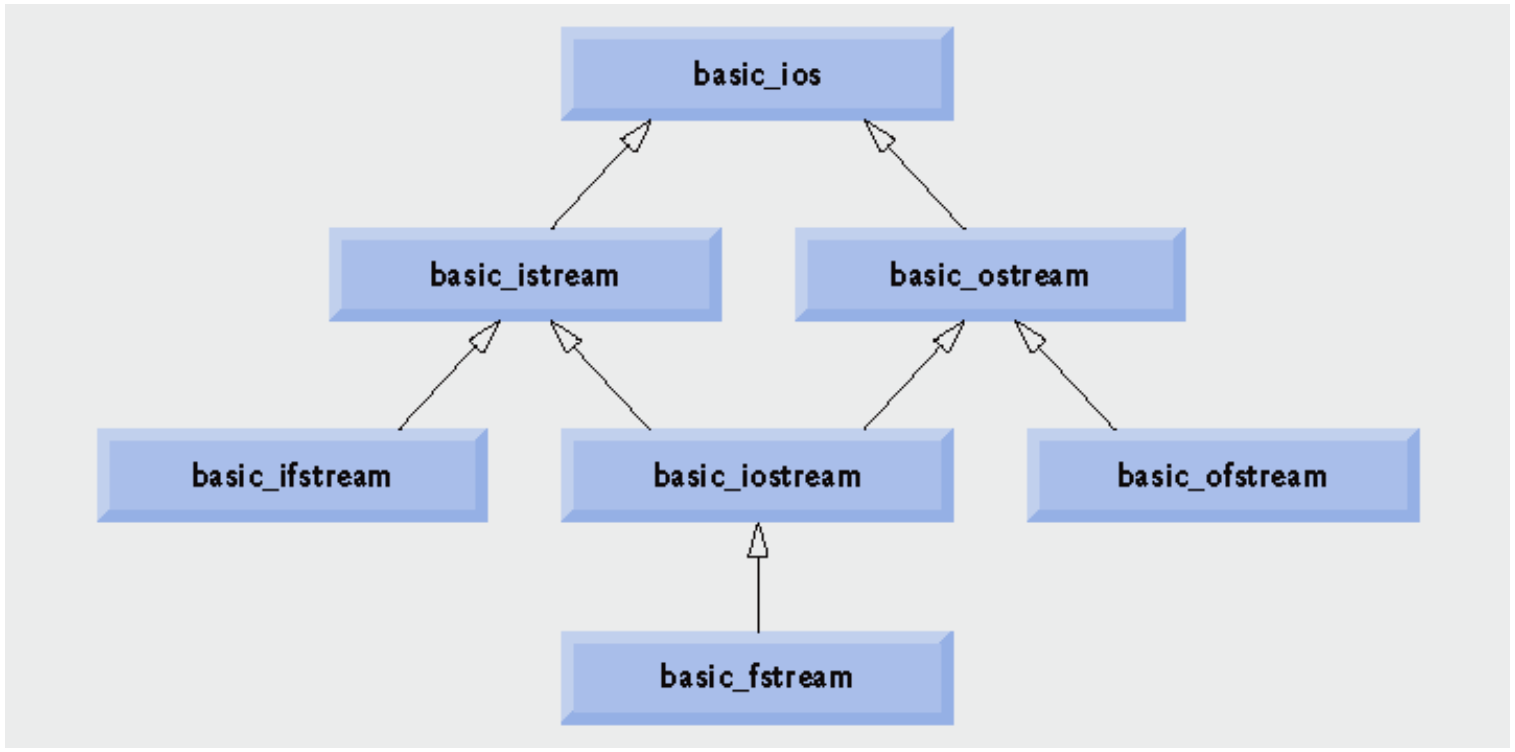

Fig. 17.3 | Portion of stream I/O template hierarchy.

17.4 Creating a Sequential File

- **File structure**
  - **The programmer must structure files**
    - **C++ does not impose structures on files**

Outline

```
// Fig. 17.4: Fig17_04.cpp
// Create a sequential file.
#include <iostream>
using std::cerr;
using std::cin;
using std::cout;
using std::endl;
using std::ios;

#include <fstream> // file stream
using std::ofstream; // output file stream

#include <cstdlib>
using std::exit; // exit function prototype

int main()
{
   // ofstream constructor opens file
   ofstream outClientFile( "clients.dat", 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? ";
```

Open file **`client.dat`** for output

Overloaded **`operator!`** will return **`true`** if the file did not open successfully

`Fig17_04.cpp`

(1 of 2)

Outline

```cpp

   int account;
   char name[ 30 ];
   double balance;

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

   return 0; // ofstream destructor closes file
} // end main
```

Overloaded **operator void *** will return the null pointer **0** (**false**) when the user enters the end-of-file indicator

Write data to **client.dat** using the stream insertion operator

ofstream destructor implicitly closes the file

```
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
```

17.4 Creating a Sequential File (Cont.)

- Creating an `ofstream` object
  - Opens a file for output
  - Constructor takes two arguments
    - A filename
      - If the file doe not exist, it is first created
    - A file-open mode
      - `ios::out` – the default mode
        - Overwrites preexisting data in the file
      - `ios::app`
        - Appends data to the end of the file
  - Can also use member function `open` on existing object
    - Takes same arguments as the constructor

Common Programming Error 17.1

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 if they exist (this also is the default action for `ios::out`).
`ios::binary`	Open a file for binary (i.e., nontext) input or output.

Fig. 17.5 | **File open modes.**

Common Programming Error 17.2

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

17.4 Creating a Sequential File (Cont.)

- Using an `ofstream` object
  - Writing to the file
    - Use the stream insertion operator
  - Member function `close`
    - Releases the file resource
    - Implicitly performed by `ofstream`'s destructor
- `ios` operators (usable with `ofstream`)
  - Operator member function `operator!`
    - Returns `true` if either the `failbit` or `badbit` is set
  - Operator member function `operator void *`
    - Converts the stream to a pointer
      - The null pointer if either the `failbit` or `badbit` is set

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. 17.6 | End-of-file key combinations for various popular computer systems.

Performance Tip 17.1

Closing files explicitly when the program no longer needs to reference them can reduce resource usage (especially if the program continues execution after closing the files).

17.5 Reading Data from a Sequential File

- Creating an `ifstream` object
  - Opens a file for input
  - Constructor takes two arguments
    - A filename
    - A file-open mode
      - `ios::in` – the default mode
        - Can only read from the file
  - Can also use member function `open` on an existing object
    - Takes same arguments as the constructor

Good Programming Practice 17.1

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.

Outline

Fig17_07.cpp

(1 of 3)

```cpp
// Fig. 17.7: Fig17_07.cpp
// Reading and printing a sequential file.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;
using std::fixed;
using std::ios;
using std::left;
using std::right;
using std::showpoint;

#include <fstream> // file stream
using std::ifstream; // input file stream

#include <iomanip>
using std::setw;
using std::setprecision;

#include <string>
using std::string;

#include <cstdlib>
using std::exit; // exit function prototype

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

Outline

Fig17_07.cpp

(2 of 3)

```

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

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

   int account;
   char name[ 30 ];
   double balance;

   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 );

   return 0; // ifstream destructor closes the file
} // end main
```

Open **clients.dat** for input

Overloaded **operator!** returns **true** if **clients.dat** was not opened successfully

Overloaded **operator void *** returns a null pointer **0** (**false**) when the end of **clients.dat** is reached

ifstream destructor implicitly closes the file

Outline

```

// 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
```

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

Fig17_07.cpp

(3 of 3)

17.5 Reading Data from a Sequential File (Cont.)

- File-position pointer
  - The byte number of the next byte to be read or written
  - Member functions `seekg` and `seekp` (of `istream` and `ostream`, respectively)
    - Repositions the file-position pointer to the specified location
      - Takes desired offset argument as a `long`
    - A second argument can specify the seek direction
      - `ios::beg` – the default
        - Positioning relative to the beginning
      - `ios::cur`
        - Positioning relative to the current position
      - `ios::end`
        - Positioning relative to the end

17.5 Reading Data from a Sequential File (Cont.)

- Member functions `seekg` and `seekp` (Cont.)
  - Examples
    - `fileObject.seekg( n );`
      - Position to the *n*th byte of `fileObject`
    - `fileObject.seekg( n, ios::cur );`
      - Position *n* bytes forward in `fileobject`
    - `fileObject.seekg( n, ios::end );`
      - Position *n* bytes back from end of `fileObject`
    - `fileObject.seekg( 0, ios::end );`
      - Position at end of `fileObject`

17.5 Reading Data from a Sequential File (Cont.)

- **File-position pointer (Cont.)**
  - Member functions `tellg` and `tellp` (of `istream` and `ostream`, respectively)
    - Returns current position of the file-position pointer as type `long`
    - Example
      - `Location = fileObject.tellg();`

Outline

```cpp
// Fig. 17.8: Fig17_08.cpp
// Credit inquiry program.
#include <iostream>
using std::cerr;
using std::cin;
using std::cout;
using std::endl;
using std::fixed;
using std::ios;
using std::left;
using std::right;
using std::showpoint;

#include <fstream>
using std::ifstream;

#include <iomanip>
using std::setw;
using std::setprecision;

#include <string>
using std::string;

#include <cstdlib>
using std::exit; // exit function prototype

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

Fig17_08.cpp

(1 of 6)

Outline

Fig17_08.cpp

(2 of 6)

```

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

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

   int request;
   int account;
   char name[ 30 ];
   double balance;

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

```

Outline

```
   // 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

      // 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

```

Fig17_08.cpp

(3 of 6)

Outline

.cpp

(4 of 6)

```
      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;
   return 0; // ifstream destructor closes the file
} // end main

// 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

```

Use `ostream` member function **`seekg`** to reposition the file-position pointer to the beginning

```
// 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

// 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
```

Fig17_08.cpp

(5 of 6)

Outline

Fig17_08.cpp

(6 of 6)

```
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

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.
```

17.6 Updating Sequential Files

- Updating a record in a sequential file
  - The new record could be longer than the old record
    - If it is, it could overwrite the next sequential record
    - You would have to rewrite every record into another file
      - Copy over all records before this one
      - Write new version of this record
      - Copy over all records after this one
    - This might be acceptable if you are updating many records

17.7 Random-Access Files

- **Random-access files**
  - Necessary for instant-access applications
    - Such as transaction-processing systems
  - A record can be inserted, deleted or modified without affecting other records
  - Various techniques can be used
    - Require that all records be of the same length, arranged in the order of the record keys
      - Program can calculate the exact location of any record
        - Base on the record size and record key

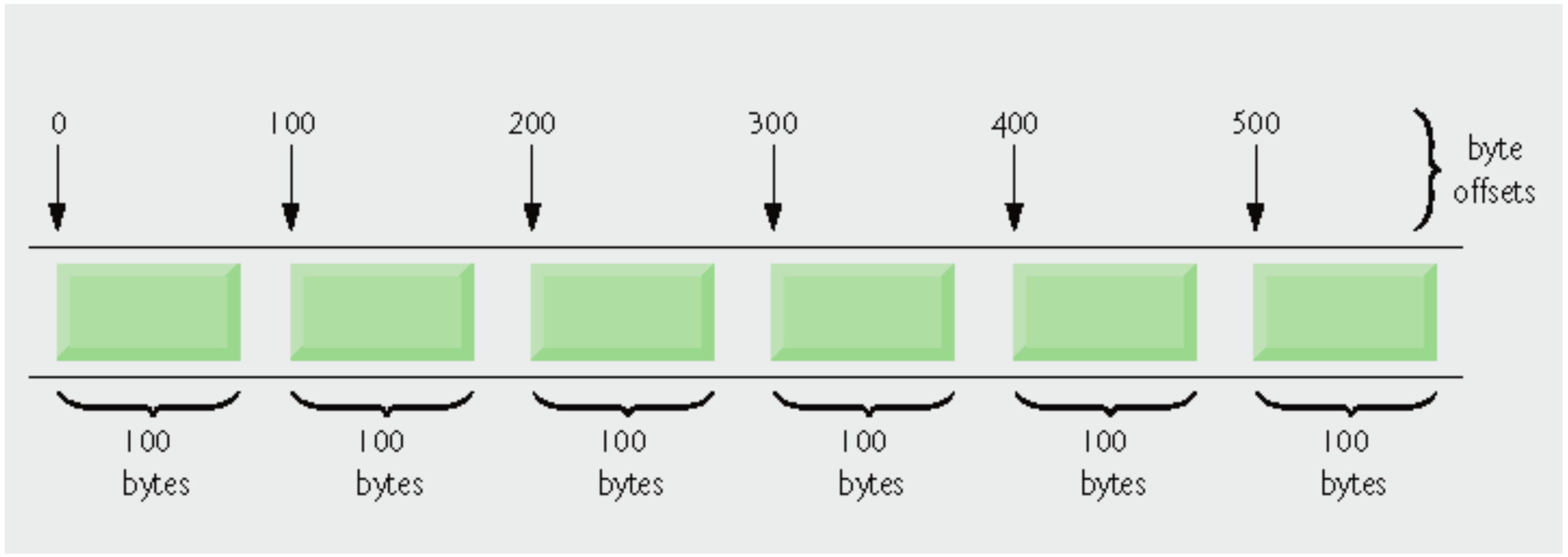

Fig. 17.9 | C++ view of a random-access file.

17.8 Creating a Random-Access File

- `ostream` member function `write`
  - Writes a number of bytes from a location in memory to the stream
    - If the stream is associated with a file, the writing is at the "put" file-position pointer
  - First argument
    - A `const char *` pointing to bytes in memory
  - Second argument
    - A `size_t` specifying the number of bytes to write
  - Example
    - `outFile.write( reinterpret_cast< const char * >( &number ), sizeof( number ) );`

17.8 Creating a Random-Access File (Cont.)

- Operator `reinterpret_cast`
  - Casts a pointer of one type to an unrelated type
  - Also converts between pointer and integer types
  - Is performed at compile time
    - Does not change the value of the object pointed to

Error-Prevention Tip 17.1

It is easy to use `reinterpret_cast` to perform dangerous manipulations that could lead to serious execution-time errors.

Portability Tip 17.1

Using `reinterpret_cast` is compiler-dependent and can cause programs to behave differently on different platforms. The `reinterpret_cast` operator should not be used unless absolute necessary.

Portability Tip 17.2

A program that reads unformatted data (written by `write`) must be compiled and executed on a system compatible with the program that wrote the data, because different systems may represent internal data differently.

Outline

ClientData.h

(1 of 2)

```
// Fig. 17.10: ClientData.h
// Class ClientData definition used in Fig. 17.12-Fig. 17.15.
#ifndef CLIENTDATA_H
#define CLIENTDATA_H

#include <string>
using std::string;

class ClientData
{
public:
   // default ClientData constructor
   ClientData( int = 0, string = "", string = "", double = 0.0 );

   // accessor functions for accountNumber
   void setAccountNumber( int );
   int getAccountNumber() const;

   // accessor functions for lastName
   void setLastName( string );
   string getLastName() const;

   // accessor functions for firstName
   void setFirstName( string );
   string getFirstName() const;

   // accessor functions for balance
   void setBalance( double );
   double getBalance() const;
private:
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   double balance;
}; // end class ClientData

#endif
```

Store the first and last name in fixed-length **`char`** arrays – we cannot use **`string`**s because they do not have uniform length

Outline

`ClientData.h`

(2 of 2)

```cpp
// Fig. 17.11: ClientData.cpp
// Class ClientData stores customer's credit information.
#include <string>
using std::string;

#include "ClientData.h"

// default ClientData constructor
ClientData::ClientData( int accountNumberValue,
   string lastNameValue, string firstNameValue, double balanceValue )
{
   setAccountNumber( accountNumberValue );
   setLastName( lastNameValue );
   setFirstName( firstNameValue );
   setBalance( balanceValue );
} // end ClientData constructor

// get account-number value
int ClientData::getAccountNumber() const
{
   return accountNumber;
} // end function getAccountNumber

// set account-number value
void ClientData::setAccountNumber( int accountNumberValue )
{
   accountNumber = accountNumberValue; // should validate
} // end function setAccountNumber
```

Outline

ClientData.cpp

(1 of 3)

Outline

```

// get last-name value
string ClientData::getLastName() const
{
   return lastName;
} // end function getLastName

// set last-name value
void ClientData::setLastName( string lastNameString )
{
   // copy at most 15 characters from string to lastName
   const char *lastNameValue = lastNameString.data();
   int length = lastNameString.size();
   length = ( length < 15 ? length : 14 );
   strncpy( lastName, lastNameValue, length );
   lastName[ length ] = '\0'; // append null character to lastName
} // end function setLastName

// get first-name value
string ClientData::getFirstName() const
{
   return firstName;
} // end function getFirstName
```

`ClientData.cpp`

(2 of 3)

`string` member function **`data`** returns an array containing the characters of the string (not guaranteed to be null terminated)

`string` member function **`size`** returns the length of **`lastNameString`**

Outline

```

// set first-name value
void ClientData::setFirstName( string firstNameString )
{
   // copy at most 10 characters from string to firstName
   const char *firstNameValue = firstNameString.data();
   int length = firstNameString.size();
   length = ( length < 10 ? length : 9 );
   strncpy( firstName, firstNameValue, length );
   firstName[ length ] = '\0'; // append null character to firstName
} // end function setFirstName

// get balance value
double ClientData::getBalance() const
{
   return balance;
} // end function getBalance

// set balance value
void ClientData::setBalance( double balanceValue )
{
   balance = balanceValue;
} // end function setBalance
```

ClientData.cpp

(3 of 3)

Outline

```
// Fig. 17.12: Fig17_12.cpp
// Creating a randomly accessed file.
#include <iostream>
using std::cerr;
using std::endl;
using std::ios;

#include <fstream>
using std::ofstream;

#include <cstdlib>
using std::exit; // exit function prototype

#include "ClientData.h" // ClientData class definition

```

Fig17_12.cpp

(1 of 2)

Outline

.cpp

(2 of 2)

```
int main()
{
   ofstream outCredit( "credit.dat", ios::binary );

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

   ClientData blankClient; // constructor zeros out each data member

   // output 100 blank records to file
   for ( int i = 0; i < 100; i++ )
      outCredit.write( reinterpret_cast< const char * >( &blankClient ),
         sizeof( ClientData ) );

   return 0;
} // end main
```

Open **credit.dat** in binary mode, which is required to write fixed-length records

Write the data in **blankClient** to **credit.dat** as bytes

17.9 Writing Data Randomly to a Random-Access File

- **Writing data randomly**
  - **Opening for input and output in binary mode**
    - **Use an `fstream` object**
    - **Combine file-open modes `ios::in`, `ios::out` and `ios::binary`**
      - **Separate each open mode from the next with the bitwise inclusive OR operator (`|`)**
  - **Use function `seekp` to set the "put" file-position pointer to the specific position**
    - **Example calculation**
      - `( n - 1 ) * sizeof( ClientData )`
        - **Byte location for *n*th `ClientData` record**
  - **Use function `write` to output the data**

Outline

Fig17_13.cpp

(1 of 4)

```
// Fig. 17.13: Fig17_13.cpp
// Writing to a random-access file.
#include <iostream>
using std::cerr;
using std::cin;
using std::cout;
using std::endl;
using std::ios;

#include <iomanip>
using std::setw;

#include <fstream>
using std::fstream;

#include <cstdlib>
using std::exit; // exit function prototype

#include "ClientData.h" // ClientData class definition

int main()
{
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   double balance;

   fstream outCredit( "credit.dat", ios::in | ios::out | ios::binary );

```

Create `fstream outCredit` to open `credit.dat` for input and output in binary mode

Outline

Fig17_13.cpp

(2 of 4)

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

   cout << "Enter account number (1 to 100, 0 to end input)\n? ";

   // require user to specify account number
   ClientData client;
   cin >> accountNumber;

   // user enters information, which is copied into file
   while ( accountNumber > 0 && accountNumber <= 100 )
   {
      // user enters last name, first name and balance
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> setw( 15 ) >> lastName;
      cin >> setw( 10 ) >> firstName;
      cin >> balance;

      // set record accountNumber, lastName, firstName and balance values
      client.setAccountNumber( accountNumber );
      client.setLastName( lastName );
      client.setFirstName( firstName );
      client.setBalance( balance );

      // seek position in file of user-specified record
      outCredit.seekp( ( client.getAccountNumber() - 1 ) *
         sizeof( ClientData ) );

      // write user-specified information in file
      outCredit.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );

      // enable user to enter another account
      cout << "Enter account number\n? ";
      cin >> accountNumber;
   } // end while

   return 0;
} // end main
```

Position the "put" file-position pointer to the desired byte location

Fig17_13.cpp

(3 of 4)

Write the **ClientData** record to the correct position in the file

Outline

Fig17_13.cpp

(4 of 4)

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

17.10 Reading from a Random-Access File Sequentially

- Sequentially reading a random-access file
  - `ifstream` member function `read`
    - Inputs a number of bytes from the current file position in the stream into an object
    - First argument
      - A `char *` pointing to the object in memory
    - Second argument
      - A `size_t` specifying the number of bytes to input
  - Additional benefit
    - Sequentially read-in records are sorted in order of ascending record keys
      - Space-time trade off: a fast sorting algorithm, but space-consuming

Outline

Fig17_14.cpp

(1 of 3)

```
// Fig. 17.14: Fig17_14.cpp
// Reading a random access file sequentially.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;
using std::fixed;
using std::ios;
using std::left;
using std::right;
using std::showpoint;

#include <iomanip>
using std::setprecision;
using std::setw;

#include <fstream>
using std::ifstream;
using std::ostream;

#include <cstdlib>
using std::exit; // exit function prototype

#include "ClientData.h" // ClientData class definition

void outputLine( ostream&, const ClientData & ); // prototype

```

Outline

Fig17_14.cpp

(2 of 3)

```
int main()
{
   ifstream inCredit( "credit.dat", ios::in );

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

   cout << left << setw( 10 ) << "Account" << setw( 16 )
      << "Last Name" << setw( 11 ) << "First Name" << left
      << setw( 10 ) << right << "Balance" << endl;

   ClientData client; // create record

   // read first record from file
   inCredit.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // read all records from file
   while ( inCredit && !inCredit.eof() )
   {
      // display record
      if ( client.getAccountNumber() != 0 )
         outputLine( cout, client );

```

This loop-continuation condition evaluates to `false` if an error occurs when reading from the file or if the end of file is reached

```
      // read next from file
      inCredit.read( reinterpret_cast< char * >( &client ),
         sizeof( ClientData ) );
   } // end while

   return 0;
} // end main

// display single record
void outputLine( ostream &output, const ClientData &record )
{
   output << left << setw( 10 ) << record.getAccountNumber()
      << setw( 16 ) << record.getLastName()
      << setw( 11 ) << record.getFirstName()
      << setw( 10 ) << setprecision( 2 ) << right << fixed
      << showpoint << record.getBalance() << endl;
} // end function outputLine
```

Because **`outputLine`** takes an **`ostream`** reference as argument, it can be used with **`cout`** (an **`ostream`** object) or an **`ofstream`** object (derived from **`ostream`**) to output to the screen or to a file

```
Account    Last Name        First Name    Balance
29         Brown            Nancy          -24.54
33         Dunn             Stacey         314.33
37         Barker           Doug             0.00
88         Smith            Dave           258.34
96         Stone            Sam             34.98
```

```cpp
// Fig. 17.15: Fig17_15.cpp
// This program reads a random access file sequentially, updates
// data previously written to the file, creates data to be placed
// in the file, and deletes data previously in the file.
#include <iostream>
using std::cerr;
using std::cin;
using std::cout;
using std::endl;
using std::fixed;
using std::ios;
using std::left;
using std::right;
using std::showpoint;

#include <fstream>
using std::ofstream;
using std::ostream;
using std::fstream;

#include <iomanip>
using std::setw;
using std::setprecision;

#include <cstdlib>
using std::exit; // exit function prototype

#include "ClientData.h" // ClientData class definition

```

Outline

Fig17_15.cpp

(1 of 10)

Outline

Fig17_15.cpp

(2 of 10)

```
int enterChoice();
void createTextFile( fstream& );
void updateRecord( fstream& );
void newRecord( fstream& );
void deleteRecord( fstream& );
void outputLine( ostream&, const ClientData & );
int getAccount( const char * const );

enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };

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

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

   int choice; // store user choice

```

"Or" together modes **ios::in** and **ios::out** for both reading and writing capabilities

Outline

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

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

   return 0;
} // end main

```

Fig17_15.cpp

(3 of 10)

Outline

Fig17_15.cpp

(4 of 10)

```
// enable user to input menu choice
int enterChoice()
{
   // display available options
   cout << "\nEnter your choice" << endl
      << "1 - store a formatted text file of accounts" << endl
      << "    called \"print.txt\" for printing" << endl
      << "2 - update an account" << endl
      << "3 - add a new account" << endl
      << "4 - delete an account" << endl
      << "5 - end program\n? ";

   int menuChoice;
   cin >> menuChoice; // input menu selection from user
   return menuChoice;
} // end function enterChoice

// create formatted text file for printing
void createTextFile( fstream &readFromFile )
{
   // create text file
   ofstream outPrintFile( "print.txt", ios::out );

   // exit program if ofstream cannot create file
   if ( !outPrintFile )
   {
      cerr << "File could not be created." << endl;
      exit( 1 );
   } // end if

```

fstream object argument for inputting data from **credit.dat**

Outline

pp

```
   outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
      << "Last Name" << setw( 11 ) << "First Name" << right
      << setw( 10 ) << "Balance" << endl;

   // set file-position pointer to beginning of readFromFile
   readFromFile.seekg( 0 );

   // read first record from record file
   ClientData client;
   readFromFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // copy all records from record file into text file
   while ( !readFromFile.eof() )
   {
      // write single record to text file
      if ( client.getAccountNumber() != 0 ) // skip empty records
         outputLine( outPrintFile, client );

      // read next record from record file
      readFromFile.read( reinterpret_cast< char * >( &client ),
         sizeof( ClientData ) );
   } // end while
} // end function createTextFile

// update balance in record
void updateRecord( fstream &updateFile )
{
   // obtain number of account to update
   int accountNumber = getAccount( "Enter account to update" );
```

Use **`istream`** member function **`seekg`** to ensure that the file-position pointer is at the beginning of the file

Outline

Fig17_15.cpp

```
   // move file-position pointer to correct record in file
   updateFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read first record from file
   ClientData client;
   updateFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // update record
   if ( client.getAccountNumber() != 0 )
   {
      outputLine( cout, client ); // display the record

      // request user to specify transaction
      cout << "\nEnter charge (+) or payment (-): ";
      double transaction; // charge or payment
      cin >> transaction;

      // update record balance
      double oldBalance = client.getBalance();
      client.setBalance( oldBalance + transaction );
      outputLine( cout, client ); // display the record

      // move file-position pointer to correct record in file
      updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );

```

Read data into object **`client`**, using **`istream`** member function **`read`**

Determine whether the record contains information

Use function **`outputLine`** with the **`cout ostream`** object to display the record

Outline

```
      // write updated record over old record in file
      updateFile.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );
   } // end if
   else // display error if account does not exist
      cerr << "Account #" << accountNumber
         << " has no information." << endl;
} // end function updateRecord

// create and insert record
void newRecord( fstream &insertInFile )
{
   // obtain number of account to create
   int accountNumber = getAccount( "Enter new account number" );

   // move file-position pointer to correct record in file
   insertInFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read record from file
   ClientData client;
   insertInFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // create record, if record does not previously exist
   if ( client.getAccountNumber() == 0 )
   {
      char lastName[ 15 ];
      char firstName[ 10 ];
      double balance;

```

Fig17_15.cpp

(7 of 10)

Outline

Fig17_15.cpp

(8 of 10)

```
      // user enters last name, first name and balance
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> setw( 15 ) >> lastName;
      cin >> setw( 10 ) >> firstName;
      cin >> balance;

      // use values to populate account values
      client.setLastName( lastName );
      client.setFirstName( firstName );
      client.setBalance( balance );
      client.setAccountNumber( accountNumber );

      // move file-position pointer to correct record in file
      insertInFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );

      // insert record in file
      insertInFile.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );
   } // end if
   else // display error if account already exists
      cerr << "Account #" << accountNumber
         << " already contains information." << endl;
} // end function newRecord

// delete an existing record
void deleteRecord( fstream &deleteFromFile )
{
   // obtain number of account to delete
   int accountNumber = getAccount( "Enter account to delete" );

```

Display an error message indicating that the account exists

Outline

```
   // move file-position pointer to correct record in file
   deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read record from file
   ClientData client;
   deleteFromFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // delete record, if record exists in file
   if ( client.getAccountNumber() != 0 )
   {
      ClientData blankClient; // create blank record

      // move file-position pointer to correct record in file
      deleteFromFile.seekp( ( accountNumber - 1 ) *
         sizeof( ClientData ) );

      // replace existing record with blank record
      deleteFromFile.write(
         reinterpret_cast< const char * >( &blankClient ),
         sizeof( ClientData ) );

      cout << "Account #" << accountNumber << " deleted.\n";
   } // end if
   else // display error if record does not exist
      cerr << "Account #" << accountNumber << " is empty.\n";
} // end deleteRecord

```

Fig17_15.cpp

(9 of 10)

Copy an empty record into the file to reinitialize that account

If the specified account is empty, display an error message

Outline

Fig17_15.cpp

(10 of 10)

```
// display single record
void outputLine( ostream &output, const ClientData &record )
{
   output << left << setw( 10 ) << record.getAccountNumber()
      << setw( 16 ) << record.getLastName()
      << setw( 11 ) << record.getFirstName()
      << setw( 10 ) << setprecision( 2 ) << right << fixed
      << showpoint << record.getBalance() << endl;
} // end function outputLine

// obtain account-number value from user
int getAccount( const char * const prompt )
{
   int accountNumber;

   // obtain account-number value
   do
   {
      cout << prompt << " (1 - 100): ";
      cin >> accountNumber;
   } while ( accountNumber < 1 || accountNumber > 100 );

   return accountNumber;
} // end function getAccount
```

17.12 Input/Output of Objects

- Inputting/outputting objects to disk files
  - Usually done by inputting/outputting the object’s data members
    - We overloaded the stream extraction operator >> and stream insertion operator << for this
    - Loses the object’s type information
      - If the program knows the object type, the program can read the correct type
      - If not, we would have to output a type code preceding each collection of data members representing an object
        - A `switch` statement can then be used to invoke the proper overloaded function