Lecture 22:
Stream Input/Output & File Processing

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 4th, 2010

15.1 Introduction

- The C++ standard libraries provide an extensive set of input/output capabilities.
- C++ uses *type-safe I/O*.
- Each I/O operation is executed in a manner sensitive to the data type.
- If an I/O member function has been de-fined to handle a particular data type, then that member function is called to handle that data type.
- If there is no match between the type of the actual data and a function for handling that data type, the compiler generates an error.
- Thus, improper data cannot "sneak" through the system.
- Users can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (<<) and the stream extraction operator (>>).

Error-Prevention Tip 15.1

C++ I/O is type safe.

15.2 Streams

- C++ I/O occurs in streams, which are sequences of bytes.
- In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory.
- In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).
- An application associates meaning with bytes.
- The system I/O mechanisms should transfer bytes from devices to memory (and vice versa) consistently and reliably.
- Such transfers often involve some mechanical motion, such as the rotation of a disk or a tape, or the typing of keystrokes at a keyboard.
- The time these transfers take is typically much greater than the time the processor requires to manipulate data internally.
- Thus, I/O operations require careful planning and tuning to ensure optimal performance.

15.3 Streams

- C++ provides both “low-level” and “high-level” I/O capabilities.
- Low-level I/O capabilities (i.e., unformatted I/O) specify that some number of bytes should be transferred device-to-memory or memory-to-device.
- In such transfers, the individual byte is the item of interest.
- Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient.
- Programmers generally prefer a higher-level view of I/O (i.e., formatted I/O*)*, in which bytes are grouped into meaningful units, such as integers, floating-point numbers, characters, strings and user-defined types.
- These type-oriented capabilities are satisfactory for most I/O other than high-volume file processing.

Performance Tip 15.1

Use unformatted I/O for the best performance in high-volume file processing.

Portability Tip 15.1

Using unformatted I/O can lead to portability problems, because unformatted data is not portable across all platforms.

15.3.2 `iostream` Library Header Files

- The C++ `iostream` library provides hundreds of I/O capabilities.
- Several header files contain portions of the library interface.
- Most C++ programs include the `<iostream>` header file, which declares basic services required for all stream-I/O operations.
- The `<iostream>` header file defines the `cin`, `cout`, `cerr` and `clog` objects, which correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively.
- Both unformatted- and formatted-I/O services are provided.

15.3.3 iostream Library Header Files (cont.)

- The `<iomanip>` header declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as `setw` and `setprecision`.
- The `<fstream>` header declares services for user-controlled file processing.
- C++ implementations generally contain other I/O-related libraries that provide sys-tem-specific capabilities, such as the controlling of special-purpose devices for audio and video I/O.

15.3.4 Stream Input/Output Classes and Objects

- The `iostream` library provides many templates for handling common I/O operations.
- Class template `basic_istream` supports stream-input operations, class template `basic_ostream` supports stream-output operations, and class template `basic_iostream` supports both stream-input and stream-output operations.
  - Each template has a predefined template specialization that enables `char` I/O.
  - In addition, the `iostream` library provides a set of `typedef`s that provide aliases for these template specializations.
  - The `typedef` specifier declares synonyms (aliases) for previously defined data types.
  - Creating a name using `typedef` does not create a data type; `typedef` creates only a type name that may be used in the program.

15.3.4 Stream Input/Output Classes and Objects (cont.)

- The `typedef` istream represents a specialization of `basic_istream` that enables `char` input.
- The `typedef` ostream represents a specialization of `basic_ostream` that enables `char` output.
- The `typedef` iostream represents a specialization of `basic_iostream` that enables both `char` input and output.
- We use these `typedef`s throughout this chapter.

15.3.4 Stream Input/Output Classes and Objects (cont.)

- Predefined object `cin` is an `istream` instance and is said to be "connected to" (or attached to) the standard input device, which usually is the keyboard.
- The `>>` operator is overloaded to input data items of fundamental types, strings and pointer values.
- The predefined object `cout` is an `ostream` instance and is said to be "connected to" the standard out-put device, which usually is the display screen.
- The `<<` operator is overloaded to output data items of fundamental types, strings and pointer values.

15.3.4 Stream Input/Output Classes and Objects (cont.)

- The predefined object `cerr` is an `ostream` instance and is said to be "connected to" the standard error device, normally the screen.
- Outputs to object `cerr` are unbuffered, implying that each stream insertion to `cerr` causes its output to appear immediately—this is appropriate for notifying a user promptly about errors.
- The predefined object `clog` is an instance of the `ostream` class and is said to be "connected to" the standard error device.
- Outputs to `clog` are buffered.
- This means that each insertion to `clog` could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.
- Buffering is an I/O performance-enhancement technique discussed in operating-systems courses.

15.4 Stream Output

- Formatted and unformatted output capabilities are provided by `ostream`.

15.4.1 Output of char * Variables

- The << operator has been overloaded to output a char * as a null-terminated string.
- To output the address, you can cast the char * to a void * (this can be done to any pointer variable).
- Figure 15.3 demonstrates printing a char * variable in both string and address formats.
- The address prints as a hexadecimal (base-16) number, which might differ among computers.
- To learn more about hexadecimal numbers, read Appendix D.

```cpp
// Fig. 15.3: Fig15_03.cpp
// Printing the address stored in a char * variable.
#include <iostream>
using namespace std;

int main()
{
   const char *const word = "again";

   // display value of char *, then display value of char *
   // static_cast to void *
   cout << "Value of word is: " << word << endl
      << "Value of static_cast< void * >( word ) is: "
      << static_cast< void * >( word ) << endl;
} // end main
```

```
Value of word is: again
Value of static_cast< void * >( word ) is: 00428300
```

Fig. 15.3 | Printing the address stored in a `char *` variable.

15.4.2 Character Output Using Member Function put

- We can use the `put` member function to output characters.
- For example, the statement
    - `cout.put( 'A' );`
- displays a single character `A`.
- Calls to `put` may be cascaded, as in the statement
    - `cout.put( 'A' ).put( '\n' );`
- which outputs the letter `A` followed by a newline character.
- As with `<<`, the preceding statement executes in this manner, because the dot operator (`.`) associates from left to right, and the `put` member function returns a reference to the `ostream` object (`cout`) that received the `put` call.
- The `put` function also may be called with a numeric expression that represents an ASCII value, as in the following statement
    - `cout.put( 65 );`
- which also out-puts `A`.

15.5 Stream Input

- Formatted and unformatted input capabilities are provided by `istream`.
- The stream extraction operator (`>>`) normally skips white-space characters (such as blanks, tabs and newlines) in the input stream; later we'll see how to change this behavior.
- After each input, the stream extraction operator returns a reference to the stream object that received the extraction message (e.g., `cin` in the expression `cin >> grade`).
- If that reference is used as a condition, the stream's overloaded `void *` cast operator function is implicitly invoked to convert the reference into a non-null pointer value or the null pointer based on the success or failure of the last input operation.
  – A non-null pointer converts to the `bool` value `true` to indicate success and the null pointer converts to the `bool` value `false` to indicate failure.
- When an attempt is made to read past the end of a stream, the stream's overloaded `void *` cast operator returns the null pointer to indicate end-of-file.

15.6.1 `get` and `getline` Member Functions

- The get member function with no arguments inputs one character from the desig-nated stream (including white-space characters and other nongraphic characters, such as the key sequence that represents end-of-file) and returns it as the value of the function call.
- This version of `get` returns `EOF` when end-of-file is encoun-tered on the stream.
- Figure 15.4 demonstrates the use of member functions `eof` and `get` on input stream `cin` and member function `put` on output stream `cout`.
- The user enters a line of text and presses Enter followed by end-of-file (*<Ctrl>-z* on Microsoft Windows systems, *<Ctrl>-d* on UNIX and Macintosh systems).
- This program uses the version of `istream` member function `get` that takes no arguments and returns the character being input (line 15).
- Function `eof` returns `true` only after the program attempts to read past the last character in the stream.

```cpp
// Fig. 15.4: Fig15_04.cpp
// Using member functions get, put and eof.
#include <iostream>
using namespace std;

int main()
{
   int character; // use int, because char cannot represent EOF

   // prompt user to enter line of text
   cout << "Before input, cin.eof() is " << cin.eof() << endl
      << "Enter a sentence followed by end-of-file:" << endl;

   // use get to read each character; use put to display it
   while ( ( character = cin.get() ) != EOF )
      cout.put( character );

   // display end-of-file character
   cout << "\nEOF in this system is: " << character << endl;
   cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
} // end main
```

Fig. 15.4 | get, put and eof member functions. (Part 1 of 2.)

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

EOF in this system is: -1
After input of EOF, cin.eof() is 1
```

Fig. 15.4 | `get`, `put` and `eof` member functions. (Part 2 of 2.)

15.6.1 get and getline Member Functions (cont.)

- The `get` member function with a character-reference argument inputs the next character from the input stream (even if this is a white-space character) and stores it in the character ar-gument.
- This version of `get` returns a reference to the `istream` object for which the `get` member function is being invoked.
- A third version of `get` takes three arguments—a character array, a size limit and a delimiter (with default value `'\n'`).
- This version reads characters from the input stream.
- It either reads one fewer than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read.
- A null character is inserted to terminate the input string in the character array used as a buffer by the pro-gram.
- The delimiter is not placed in the character array but does remain in the input stream (the delimiter will be the next character read).

15.6.1 `get` and `getline` Member Functions (cont.)

- Figure 15.5 compares input using stream extraction with `cin` (which reads characters until a white-space character is encountered) and input using `cin.get`.
- The call to `cin.get` (line 22) does not specify a delimiter, so the default `'\n'` character is used.

```cpp
// Fig. 15.5: Fig15_05.cpp
// Contrasting input of a string via cin and cin.get.
#include <iostream>
using namespace std;

int main()
{
   // create two char arrays, each with 80 elements
   const int SIZE = 80;
   char buffer1[ SIZE ];
   char buffer2[ SIZE ];

   // use cin to input characters into buffer1
   cout << "Enter a sentence:" << endl;
   cin >> buffer1;

   // display buffer1 contents
   cout << "\nThe string read with cin was:" << endl
      << buffer1 << endl << endl;

   // use cin.get to input characters into buffer2
   cin.get( buffer2, SIZE );
```

Fig. 15.5 | Input of a string using `cin` with stream extraction contrasted with input using `cin.get`. (Part 1 of 2.)

```

   // display buffer2 contents
   cout << "The string read with cin.get was:" << endl
      << buffer2 << endl;
} // end main
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Fig. 15.5 | Input of a string using `cin` with stream extraction contrasted with input using `cin.get`. (Part 2 of 2.)

15.6.1 get and getline Member Functions (cont.)

- Member function getline operates similarly to the third version of the get member function and inserts a null character after the line in the character array.
- The getline function removes the delimiter from the stream (i.e., reads the character and discards it), but does not store it in the character ar-ray.
- The program of Fig. 15.6 demonstrates the use of the getline member function to input a line of text (line 13).

```cpp
// Fig. 15.6: Fig15_06.cpp
// Inputting characters using cin member function getline.
#include <iostream>
using namespace std;

int main()
{
   const int SIZE = 80;
   char buffer[ SIZE ]; // create array of 80 characters

   // input characters in buffer via cin function getline
   cout << "Enter a sentence:" << endl;
   cin.getline( buffer, SIZE );

   // display buffer contents
   cout << "\nThe sentence entered is:" << endl << buffer << endl;
} // end main
```

Fig. 15.6 | Inputting character data with `cin` member function `getline`. (Part 1 of 2.)

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Fig. 15.6 | Inputting character data with `cin` member function `getline`. (Part 2 of 2.)

15.6.2 `istream` Member Functions `peek`, `putback` and `ignore`

- The ignore member function of `istream` reads and discards a designated number of characters (the default is one) or terminates upon encountering a designated delimiter (the default is `EOF`, which causes `ignore` to skip to the end of the file when reading from a file).
- The putback member function places the previous character obtained by a `get` from an input stream back into that stream.
  - This function is useful for applications that scan an input stream looking for a field beginning with a specific character.
  - When that character is input, the application returns the character to the stream, so the character can be included in the input data.
- The peek member function returns the next character from an input stream but does not remove the character from the stream.

15.6.3 Type-Safe I/O

- C++ offers type-safe I/O.
- The **<<** and **>>** operators are overloaded to accept data items of specific types.
- If unexpected data is processed, various error bits are set, which the user may test to determine whether an I/O operation succeeded or failed.
- If operator << has not been overloaded for a user-defined type and you attempt to input into or output the contents of an object of that user-defined type, the compiler reports an error.
- This enables the program to "stay in control."

15.7 Unformatted I/O Using read, write and gcount

- Unformatted input/output is performed using the read and write member functions of `istream` and `ostream`, respectively.
- Member function `read` inputs bytes to a character array in memory; member function `write` outputs bytes from a character array.
- These bytes are not formatted in any way.
- They're input or output as raw bytes.
- The `read` member function inputs a designated number of characters into a character array.
- If fewer than the designated number of characters are read, `failbit` is set.
- Section 15.8 shows how to determine whether `failbit` has been set.
- Member function gcount reports the number of characters read by the last input operation.

15.7 Unformatted I/O Using read, write and gcount (cont.)

- Figure 15.7 demonstrates `istream` member functions `read` and `gcount`, and `ostream` member function `write`.

```cpp
// Fig. 15.7: Fig15_07.cpp
// Unformatted I/O using read, gcount and write.
#include <iostream>
using namespace std;

int main()
{
   const int SIZE = 80;
   char buffer[ SIZE ]; // create array of 80 characters

   // use function read to input characters into buffer
   cout << "Enter a sentence:" << endl;
   cin.read( buffer, 20 );

   // use functions write and gcount to display buffer characters
   cout << endl << "The sentence entered was:" << endl;
   cout.write( buffer, cin.gcount() );
   cout << endl;
} // end main
```

Fig. 15.7 | Unformatted I/O using the `read`, `gcount` and `write` member functions. (Part 1 of 2.)

```
Enter a sentence:
Using the read, write, and gcount member functions
The sentence entered was:
Using the read, writ
```

Fig. 15.7 | Unformatted I/O using the `read`, `gcount` and `write` member functions. (Part 2 of 2.)

15.8.2 Floating-Point Precision (precision, setprecision)

- We can control the precision of floating-point numbers (i.e., the number of digits to the right of the decimal point) by using either the `setprecision` stream manipulator or the `precision` member function of `ios_base`.
- A call to either of these sets the precision for all subse-quent output operations until the next precision-setting call.
- A call to member function `precision` with no argument returns the current precision setting (this is what you need to use so that you can restore the original precision eventually after a "sticky" setting is no longer needed).
- The program of Fig. 15.9 uses both member function `precision` (line 22) and the `setprecision` manipula-tor (line 31) to print a table that shows the square root of `2`, with precision varying from `0` to `9`.

```cpp
// Fig. 15.9: Fig15_09.cpp
// Controlling precision of floating-point values.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
   double root2 = sqrt( 2.0 ); // calculate square root of 2
   int places; // precision, vary from 0-9

   cout << "Square root of 2 with precisions 0-9." << endl
      << "Precision set by ios_base member function "
      << "precision:" << endl;

   cout << fixed; // use fixed-point notation

   // display square root using ios_base function precision
   for ( places = 0; places <= 9; places++ )
   {
      cout.precision( places );
      cout << root2 << endl;
   } // end for
```

Fig. 15.9 | Precision of floating-point values. (Part 1 of 3.)

```
    cout << "\nPrecision set by stream manipulator "
       << "setprecision:" << endl;

    // set precision for each digit, then display square root
    for ( places = 0; places <= 9; places++ )
       cout << setprecision( places ) << root2 << endl;
 } // end main
```

```
Square root of 2 with precisions 0-9.
Precision set by ios_base member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 15.9 | Precision of floating-point values. (Part 2 of 3.)

```
Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 15.9 | Precision of floating-point values. (Part 3 of 3.)

15.8.3 Field Width (width, setw)

- The `width` member function (of base class `ios_base`) sets the field width (i.e., the number of character positions in which a value should be output or the maximum number of characters that should be input) and returns the previous width.
- If values output are narrower than the field width, fill characters are inserted as padding.
- A value wider than the designated width will not be truncated—the full number will be printed.
- The `width` function with no argument returns the current setting.
- Figure 15.10 demonstrates the use of the `width` member function on both input and output.
- On input into a `char` array, a maximum of one fewer characters than the width will be read.
- Remem-ber that stream extraction terminates when nonleading white space is encountered.
- The `setw` stream manipulator also may be used to set the field width.

Common Programming Error 15.1

The width setting applies only for the next insertion or extraction (i.e., the width setting is not "sticky"); afterward, the width is set implicitly to 0 (i.e., input and output will be performed with default settings). Assuming that the width setting applies to all subsequent outputs is a logic error.

Common Programming Error 15.2

When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs.

```cpp
// Fig. 15.10: Fig15_10.cpp
// Demonstrating member function width.
#include <iostream>
using namespace std;

int main()
{
   int widthValue = 4;
   char sentence[ 10 ];

   cout << "Enter a sentence:" << endl;
   cin.width( 5 ); // input only 5 characters from sentence

   // set field width, then display characters based on that width
   while ( cin >> sentence )
   {
      cout.width( widthValue++ );
      cout << sentence << endl;
      cin.width( 5 ); // input 5 more characters from sentence
   } // end while
} // end main
```

Fig. 15.10 | width member function of class ios_base. (Part 1 of 2.)

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

Fig. 15.10 | `width` member function of class `ios_base`. (Part 2 of 2.)

15.9.7 Specifying Boolean Format (`boolalpha`)

- C++ provides data type `bool`, whose values may be `false` or `true`, as a preferred alternative to the old style of using `0` to indicate `false` and nonzero to indicate `true`.
- A `bool` variable outputs as `0` or `1` by default.
- However, we can use stream manipulator boolalpha to set the output stream to display `bool` values as the strings `"true"` and `"false"`.
- Use stream manipulator noboolalpha to set the output stream to display `bool` values as integers (i.e., the default setting).
- The program of Fig. 15.20 demonstrates these stream manipulators.

Good Programming Practice 15.1

Displaying `bool` *values as* `true` *or* `false`*, rather than nonzero or 0, respectively, makes program outputs clearer.*

```cpp
// Fig. 15.20: Fig15_20.cpp
// Demonstrating stream manipulators boolalpha and noboolalpha.
#include <iostream>
using namespace std;

int main()
{
   bool booleanValue = true;

   // display default true booleanValue
   cout << "booleanValue is " << booleanValue << endl;

   // display booleanValue after using boolalpha
   cout << "booleanValue (after using boolalpha) is "
      << boolalpha << booleanValue << endl << endl;

   cout << "switch booleanValue and use noboolalpha" << endl;
   booleanValue = false; // change booleanValue
   cout << noboolalpha << endl; // use noboolalpha

   // display default false booleanValue after using noboolalpha
   cout << "booleanValue is " << booleanValue << endl;

```

Fig. 15.20 | Stream manipulators `boolalpha` and `noboolalpha`. (Part 1 of 2.)

```
    // display booleanValue after using boolalpha again
    cout << "booleanValue (after using boolalpha) is "
       << boolalpha << booleanValue << endl;
} // end main
```

```
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false
```

Fig. 15.20 | Stream manipulators `boolalpha` and `noboolalpha`. (Part 2 of 2.)

15.10 Stream Error States

- The state of a stream may be tested through bits in class `ios_base`.
- The `eofbit` is set for an input stream after end-of-file is encountered.
- A program can use member function `eof` to determine whether end-of-file has been encountered on a stream after an attempt to extract data beyond the end of the stream.
- The `failbit` is set for a stream when a format error occurs on the stream and no characters are input (e.g., when you attempt to read a number and the user enters a string).
  - When such an error occurs, the characters are not lost.
- The `fail` member function reports whether a stream operation has failed.
- Usually, recovering from such errors is possible.

```
// Fig. 15.22: Fig15_22.cpp
// Testing error states.
#include <iostream>
using namespace std;

int main()
{
   int integerValue;

   // display results of cin functions
   cout << "Before a bad input operation:"
      << "\ncin.rdstate(): " << cin.rdstate()
      << "\n    cin.eof(): " << cin.eof()
      << "\n   cin.fail(): " << cin.fail()
      << "\n    cin.bad(): " << cin.bad()
      << "\n   cin.good(): " << cin.good()
      << "\n\nExpects an integer, but enter a character: ";

   cin >> integerValue; // enter character value
   cout << endl;

```

Fig. 15.22 | Testing error states. (Part 1 of 3.)

```
   // display results of cin functions after bad input
   cout << "After a bad input operation:"
      << "\ncin.rdstate(): " << cin.rdstate()
      << "\n    cin.eof(): " << cin.eof()
      << "\n   cin.fail(): " << cin.fail()
      << "\n    cin.bad(): " << cin.bad()
      << "\n   cin.good(): " << cin.good() << endl << endl;

   cin.clear(); // clear stream

   // display results of cin functions after clearing cin
   cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
      << "\ncin.good(): " << cin.good() << endl;
} // end main
```

Fig. 15.22 | Testing error states. (Part 2 of 3.)

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 1
    cin.bad(): 0
   cin.good(): 0

After cin.clear()
cin.fail(): 0
cin.good(): 1
```

Fig. 15.22 | Testing error states. (Part 3 of 3.)

15.11 Stream Error States

- The `badbit` is set for a stream when an error occurs that results in the loss of data.
- The bad member function reports whether a stream operation failed.
  - Generally, such serious fail-ures are nonrecoverable.
- The goodbit is set for a stream if none of the bits `eofbit`, `failbit` or `bad-bit` is set for the stream.
- The good member function returns `true` if the `bad`, `fail` and `eof` functions would all return `false`.
- I/O operations should be performed only on "good" streams.
- The `rdstate` member function returns the stream's error state.
- The preferred means of testing the state of a stream is to use member functions `eof`, `bad`, `fail` and `good`—using these func-tions does not require you to be familiar with particular status bits.
- The `clear` member function is used to restore a stream's state to "good," so that I/O may proceed on that stream.

15.12 Stream Error States

- The program of Fig. 15.22 demonstrates member functions `rdstate`, `eof`, `fail`, `bad`, `good` and `clear`.
- The `operator!` member function of `basic_ios` returns `true` if the `badbit` is set, the `failbit` is set or both are set.
- The `operator void *` member function returns `false` (`0`) if the `badbit` is set, the `failbit` is set or both are set.
- These functions are use-ful in file processing when a `true`/`false` condition is being tested under the control of a se-lection statement or repetition statement.

17.1 Introduction

- Storage of data in memory is temporary.
- Files are used for data persistence—permanent reten-tion 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 data files.
- We consider both sequential files and random-access files.
- We compare formatted-data file processing and raw-data file processing.
- 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.

17.2 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).

17.2 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.

17.2 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`

17.2 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. 17.1), 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.

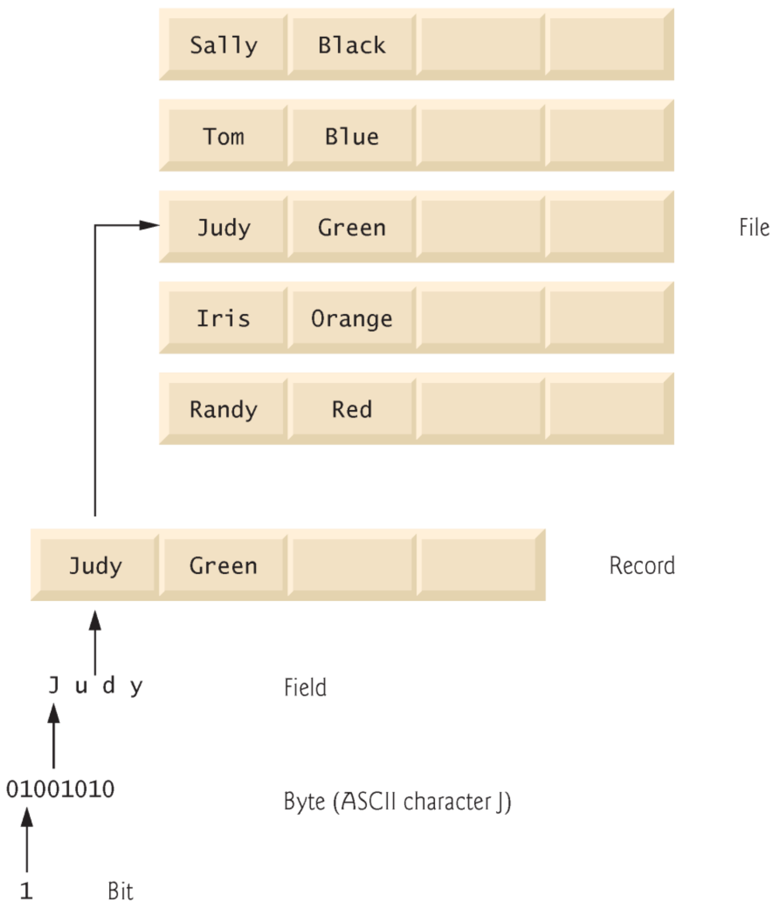

Fig. 17.1 | Data hierarchy.

17.2 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.

17.2 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).

17.3 Files and Streams

- C++ views each file as a sequence of bytes (Fig. 17.2).
- Each file ends ei-ther with an end-of-file marker or at a specific byte number recorded in an operating-system-main-tained, 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.

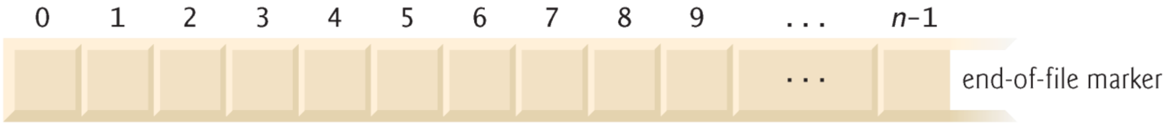

Fig. 17.2 | C++'s view of a file of *n* bytes.

17.3 Files and Streams (cont.)

- To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included.
- Header `<fstream>` includes the definitions for the stream class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input and output).
- Each class template has a predefined template specialization that enables `char` I/O.

17.3 Files and Streams (cont.)

- The `<fstream>` library provides `typedef` aliases for these template specializations.
- The `typedef` `ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file.
- The `typedef` `ofstream` represents a specialization of `basic_ofstream` that enables `char` output to files.
- The `typedef` `fstream` represents a specialization of `basic_fstream` that enables `char` input from, and output to, files.
- Files are opened by creating objects of these stream template specializations.

17.3 Files and Streams (cont.)

- These templates "derive" from class templates `basic_istream`, `basic_ostream` and `basic_iostream`, respectively.
- Thus, all member func-tions, operators and manipulators that belong to these templates (which we described in Chapter 15) also can be applied to file streams.
- Figure 17.3 summarizes the inheritance relationships of the I/O classes that we've discussed to this point.

17.4 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 17.4 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 ac-count 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.

```
// Fig. 17.4: Fig17_04.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.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? ";

```

Fig. 17.4 | Creating a sequential file. (Part 1 of 2.)

```
    int account;
    string name;
    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
 } // 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. 17.4 | Creating a sequential file. (Part 2 of 2.)

17.4 Creating a Sequential File (cont.)

- In Fig. 17.4, 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 estab-lishes 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 17.5 lists the file-open modes.

Questions

?