Chapter 15 Stream Input/Output

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

©1992-2011 by Pearson Education, Inc. All Rights Reserved.

OBJECTIVES

In this chapter you'll learn:

- To use C++ object-oriented stream input/output.
- To format input and output.
- The stream-I/O class hierarchy.
- To use stream manipulators.
- To control justification and padding.
- To determine the success or failure of input/output operations.
- To tie output streams to input streams.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.1 Introduction

15.2 Streams

15.2.1 Classic Streams vs. Standard Streams
15.2.2 `iostream` Library Header Files
15.2.3 Stream Input/Output Classes and Objects

15.3 Stream Output

15.3.1 Output of `char *` Variables
15.3.2 Character Output Using Member Function `put`

15.4 Stream Input

15.4.1 `get` and `getline` Member Functions
15.4.2 `istream` Member Functions `peek`, `putback` and `ignore`
15.4.3 Type-Safe I/O

15.5 Unformatted I/O Using `read`, `write` and `gcount`

15.6 Introduction to Stream Manipulators

15.6.1 Integral Stream Base: `dec`, `oct`, `hex` and `setbase`
15.6.2 Floating-Point Precision (`precision`, `setprecision`)
15.6.3 Field Width (`width`, `setw`)
15.6.4 User-Defined Output Stream Manipulators

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.7 Stream Format States and Stream Manipulators

15.7.1 Trailing Zeros and Decimal Points (`showpoint`)
15.7.2 Justification (`left`, `right` and `internal`)
15.7.3 Padding (`fill`, `setfill`)
15.7.4 Integral Stream Base (`dec`, `oct`, `hex`, `showbase`)
15.7.5 Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)
15.7.6 Uppercase/Lowercase Control (`uppercase`)
15.7.7 Specifying Boolean Format (`boolalpha`)
15.7.8 Setting and Resetting the Format State via Member-Function `flags`

15.8 Stream Error States

15.9 Tying an Output Stream to an Input Stream

15.10 Wrap-Up

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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 (>>).

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Software Engineering Observation 15.1

Use the C++-style I/O exclusively in C++ programs, even though C-style I/O is available to C++ programmers.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Error-Prevention Tip 15.1

C++ I/O is type safe.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Software Engineering Observation 15.2

C++ enables a common treatment of I/O for predefined types and user-defined types. This commonality facilitates software development and reuse.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Performance Tip 15.1

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Portability Tip 15.1

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.3.1 Classic Streams vs. Standard Streams

- In the past, the C++ classic stream libraries enabled input and output of `char`s.
- Because a `char` normally occupies one byte, it can represent only a limited set of characters (such as those in the ASCII character set).
- However, many languages use alphabets that contain more characters than a single-byte `char` can represent.
- The ASCII character set does not provide these characters; the Unicode® character set does.
- Unicode is an extensive international character set that represents the majority of the world's "commercially viable" languages, mathematical symbols and much more.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.3.1 Classic Streams vs. Standard Streams (cont.)

- C++ includes the standard stream libraries, which enable developers to build systems capable of performing I/O operations with Unicode characters.
- For this purpose, C++ includes an additional character type called `wchar_t`, which can store 2-byte Unicode characters.
- The C++ standard also redesigned the classic C++ stream classes, which processed only `char`s, as class templates with separate specializations for processing characters of types `char` and `wchar_t`, respectively.
- We use the `char` type of class templates throughout this book.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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

- Templates `basic_istream` and `basic_ostream` both derive through single inheritance from base template `basic_ios`.Template `basic_iostream` derives through multiple inheritance from templates `basic_istream` and `basic_ostream`.
- The UML class diagram of Fig. 15.1 summarizes these inheritance relationships.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

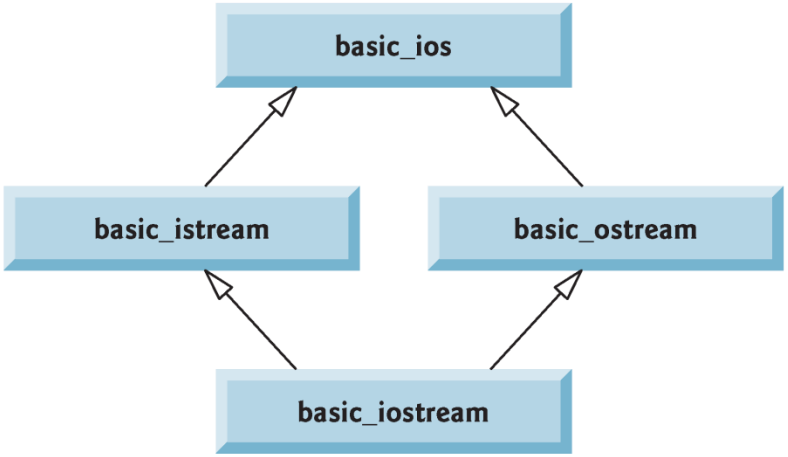

Fig. 15.1 | Stream-I/O template hierarchy portion.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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

- C++ file processing uses 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.
- C++ provides a set of `typedef`s that provide 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 a file.
- The `typedef` `fstream` represents a specialization of `basic_fstream` that enables `char` input from, and output to, a file.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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

- Template `basic_ifstream` inherits from `basic_istream`, `basic_ofstream` inherits from `basic_ostream` and `basic_fstream` inherits from `basic_iostream`.
- The UML class diagram of Fig. 15.2 summarizes the various inheritance relationships of the I/O-related classes.
- The full stream-I/O class hierarchy provides most of the capabilities that you need.
- Consult the class-library reference for your C++ system for additional file-processing information.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

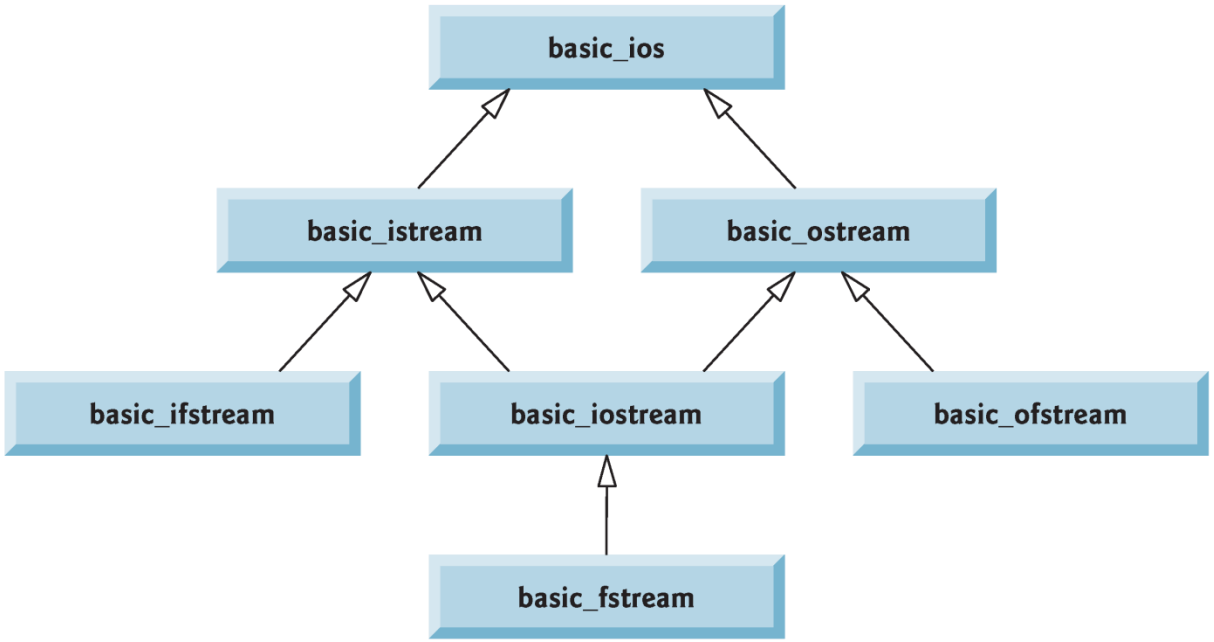

Fig. 15.2 | Stream-I/O template hierarchy portion showing the main file-processing templates.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.4 Stream Output

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.3: Fig15_03.cpp
2  // Printing the address stored in a char * variable.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const char *const word = "again";
9
10    // display value of char *, then display value of char *
11    // static_cast to void *
12    cout << "Value of word is: " << word << endl
13       << "Value of static_cast< void * >( word ) is: "
14       << static_cast< void * >( word ) << endl;
15 } // 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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.6 Stream Input

- Each stream object contains a set of state bits used to control the stream's state (i.e., formatting, setting error states, etc.).
- These bits are used by the stream's overloaded `void *` cast operator to determine whether to return a non-null pointer or the null pointer.
- Stream extraction causes the stream's `failbit` to be set if data of the wrong type is input and causes the stream's `badbit` to be set if the operation fails.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.4: Fig15_04.cpp
2  // Using member functions get, put and eof.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int character; // use int, because char cannot represent EOF
9
10    // prompt user to enter line of text
11    cout << "Before input, cin.eof() is " << cin.eof() << endl
12       << "Enter a sentence followed by end-of-file:" << endl;
13
14    // use get to read each character; use put to display it
15    while ( ( character = cin.get() ) != EOF )
16       cout.put( character );
17
18    // display end-of-file character
19    cout << "\nEOF in this system is: " << character << endl;
20    cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
21 } // end main
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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).

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.5: Fig15_05.cpp
2  // Contrasting input of a string via cin and cin.get.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // create two char arrays, each with 80 elements
9     const int SIZE = 80;
10    char buffer1[ SIZE ];
11    char buffer2[ SIZE ];
12
13    // use cin to input characters into buffer1
14    cout << "Enter a sentence:" << endl;
15    cin >> buffer1;
16
17    // display buffer1 contents
18    cout << "\nThe string read with cin was:" << endl
19       << buffer1 << endl << endl;
20
21    // use cin.get to input characters into buffer2
22    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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
23
24      // display buffer2 contents
25      cout << "The string read with cin.get was:" << endl
26         << buffer2 << endl;
27   } // 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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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).

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.6: Fig15_06.cpp
2  // Inputting characters using cin member function getline.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 80;
9     char buffer[ SIZE ]; // create array of 80 characters
10
11    // input characters in buffer via cin function getline
12    cout << "Enter a sentence:" << endl;
13    cin.getline( buffer, SIZE );
14
15    // display buffer contents
16    cout << "\nThe sentence entered is:" << endl << buffer << endl;
17 } // end main
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.”

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.7: Fig15_07.cpp
2  // Unformatted I/O using read, gcount and write.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 80;
9     char buffer[ SIZE ]; // create array of 80 characters
10
11    // use function read to input characters into buffer
12    cout << "Enter a sentence:" << endl;
13    cin.read( buffer, 20 );
14
15    // use functions write and gcount to display buffer characters
16    cout << endl << "The sentence entered was:" << endl;
17    cout.write( buffer, cin.gcount() );
18    cout << endl;
19 } // end main
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.8 Introduction to Stream Manipulators

- C++ provides various stream manipulators that perform formatting tasks.
- The stream manipulators provide capabilities such as setting field widths, setting precision, setting and unsetting format state, setting the fill character in fields, flushing streams, inserting a newline into the output stream (and flushing the stream), inserting a null character into the output stream and skipping white space in the input stream.
- These features are described in the following sections.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.8.1 Integral Stream Base: dec, oct, hex and setbase

- Integers are interpreted normally as decimal (base-10) values.
- To change the base in which integers are interpreted on a stream, insert the `hex` manipulator to set the base to hexadecimal (base 16) or insert the `oct` manipulator to set the base to octal (base 8).
- Insert the `dec` manipulator to reset the stream base to decimal.
- These are all sticky manipulators.
- The base of a stream also may be changed by the `setbase` stream manipulator, which takes one integer argument of 10, 8, or 16 to set the base to decimal, octal or hexadecimal, respectively.
- Because `setbase` takes an argument, it's called a parameterized stream manipulator.
- Figure 15.8 demonstrates stream manipulators `hex`, `oct`, `dec` and `setbase`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.8: Fig15_08.cpp
2  // Using stream manipulators hex, oct, dec and setbase.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int number;
10
11    cout << "Enter a decimal number: ";
12    cin >> number; // input number
13
14    // use hex stream manipulator to show hexadecimal number
15    cout << number << " in hexadecimal is: " << hex
16       << number << endl;
17
18    // use oct stream manipulator to show octal number
19    cout << dec << number << " in octal is: "
20       << oct << number << endl;
21
22    // use setbase stream manipulator to show decimal number
23    cout << setbase( 10 ) << number << " in decimal is: "
24       << number << endl;
25 } // end main
```

Fig. 15.8 | Stream manipulators `hex`, `oct`, `dec` and `setbase`. (Part 1 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 15.8 | Stream manipulators `hex`, `oct`, `dec` and `setbase`. (Part 2 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.9: Fig15_09.cpp
2  // Controlling precision of floating-point values.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath>
6  using namespace std;
7
8  int main()
9  {
10    double root2 = sqrt( 2.0 ); // calculate square root of 2
11    int places; // precision, vary from 0-9
12
13    cout << "Square root of 2 with precisions 0-9." << endl
14       << "Precision set by ios_base member function "
15       << "precision:" << endl;
16
17    cout << fixed; // use fixed-point notation
18
19    // display square root using ios_base function precision
20    for ( places = 0; places <= 9; places++ )
21    {
22       cout.precision( places );
23       cout << root2 << endl;
24    } // end for
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
25
26      cout << "\nPrecision set by stream manipulator "
27         << "setprecision:" << endl;
28
29      // set precision for each digit, then display square root
30      for ( places = 0; places <= 9; places++ )
31         cout << setprecision( places ) << root2 << endl;
32   } // 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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.10: Fig15_10.cpp
2  // Demonstrating member function width.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int widthValue = 4;
9     char sentence[ 10 ];
10
11    cout << "Enter a sentence:" << endl;
12    cin.width( 5 ); // input only 5 characters from sentence
13
14    // set field width, then display characters based on that width
15    while ( cin >> sentence )
16    {
17       cout.width( widthValue++ );
18       cout << sentence << endl;
19       cin.width( 5 ); // input 5 more characters from sentence
20    } // end while
21 } // end main
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.8.4 User-Defined Output Stream Manipulators

- You can create your own stream manipulators.
- Figure 15.11 shows the creation and use of new nonparameterized stream manipulators `bell` (lines 8–11), `carriageReturn` (lines 14–17), `tab` (lines 20–23) and `endLine` (lines 27–30).
- For output stream manipulators, the return type and parameter must be of type `ostream &`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.11: Fig15_11.cpp
2  // Creating and testing user-defined, nonparameterized
3  // stream manipulators.
4  #include <iostream>
5  using namespace std;
6
7  // bell manipulator (using escape sequence \a)
8  ostream& bell( ostream& output )
9  {
10    return output << '\a'; // issue system beep
11 } // end bell manipulator
12
13 // carriageReturn manipulator (using escape sequence \r)
14 ostream& carriageReturn( ostream& output )
15 {
16    return output << '\r'; // issue carriage return
17 } // end carriageReturn manipulator
18
19 // tab manipulator (using escape sequence \t)
20 ostream& tab( ostream& output )
21 {
22    return output << '\t'; // issue tab
23 } // end tab manipulator
```

Fig. 15.11 | User-defined, nonparameterized stream manipulators. (Part 1 of 3.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
24
25 // endLine manipulator (using escape sequence \n and member
26 // function flush)
27 ostream& endLine( ostream& output )
28 {
29    return output << '\n' << flush; // issue endl-like end of line
30 } // end endLine manipulator
31
32 int main()
33 {
34    // use tab and endLine manipulators
35    cout << "Testing the tab manipulator:" << endLine
36       << 'a' << tab << 'b' << tab << 'c' << endLine;
37
38    cout << "Testing the carriageReturn and bell manipulators:"
39       << endLine << ".........";
40
41    cout << bell; // use bell manipulator
42
43    // use carriageReturn and endLine manipulators
44    cout << carriageReturn << "-----" << endLine;
45 } // end main
```

Fig. 15.11 | User-defined, nonparameterized stream manipulators. (Part 2 of 3.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
Testing the tab manipulator:
a       b       c
Testing the carriageReturn and bell manipulators:
-----.....
```

Fig. 15.11 | User-defined, nonparameterized stream manipulators. (Part 3 of 3.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9 Stream Format States and Stream Manipulators

- Various stream manipulators can be used to specify the kinds of formatting to be performed during stream-I/O operations.
- Stream manipulators control the output's format settings.
- Figure 15.12 lists each stream manipulator that controls a given stream's format state.
- All these manipulators belong to class `ios_base`.
- We show examples of most of these stream manipulators in the next several sections.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Stream manipulator	Description
`skipws`	Skip white-space characters on an input stream. This setting is reset with stream manipulator `noskipws`.
`left`	Left justify output in a field. Padding characters appear to the right if necessary.
`right`	Right justify output in a field. Padding characters appear to the left if necessary.
`internal`	Indicate that a number's sign should be left justified in a field and a number's magnitude should be right justified in that same field (i.e., padding characters appear between the sign and the number).
`dec`	Specify that integers should be treated as decimal (base 10) values.
`oct`	Specify that integers should be treated as octal (base 8) values.
`hex`	Specify that integers should be treated as hexadecimal (base 16) values.
`showbase`	Specify that the base of a number is to be output ahead of the number (a leading `0` for octals; a leading `0x` or `0X` for hexadecimals). This setting is reset with stream manipulator `noshowbase`.

Fig. 15.12 | Format state stream manipulators from `<iostream>`. (Part 1 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Stream manipulator	Description
`showpoint`	Specify that floating-point numbers should be output with a decimal point. This is used normally with `fixed` to guarantee a certain number of digits to the right of the decimal point, even if they're zeros. This setting is reset with stream manipulator `noshowpoint`.
`uppercase`	Specify that uppercase letters (i.e., `X` and `A` through `F`) should be used in a hexadecimal integer and that uppercase `E` should be used when representing a floating-point value in scientific notation. This setting is reset with stream manipulator `nouppercase`.
`showpos`	Specify that positive numbers should be preceded by a plus sign (`+`). This setting is reset with stream manipulator `noshowpos`.
`scientific`	Specify output of a floating-point value in scientific notation.
`fixed`	Specify output of a floating-point value in fixed-point notation with a specific number of digits to the right of the decimal point.

Fig. 15.12 | Format state stream manipulators from `<iostream>`. (Part 2 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.1 Trailing Zeros and Decimal Points (`showpoint`)

- Stream manipulator `showpoint` forces a floating-point number to be output with its decimal point and trailing zeros.
- To reset the `showpoint` setting, output the stream manipulator `noshowpoint`.
- The program in Fig. 15.13 shows how to use stream manipulator `showpoint` to control the printing of trailing zeros and decimal points for floating-point values.
- Recall that the default precision of a floating-point number is 6.
- When neither the `fixed` nor the `scientific` stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after decimal point.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.13: Fig15_13.cpp
2  // Controlling the printing of trailing zeros and
3  // decimal points in floating-point values.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     // display double values with default stream format
10    cout << "Before using showpoint" << endl
11       << "9.9900 prints as: " << 9.9900 << endl
12       << "9.9000 prints as: " << 9.9000 << endl
13       << "9.0000 prints as: " << 9.0000 << endl << endl;
14
15    // display double value after showpoint
16    cout << showpoint
17       << "After using showpoint" << endl
18       << "9.9900 prints as: " << 9.9900 << endl
19       << "9.9000 prints as: " << 9.9000 << endl
20       << "9.0000 prints as: " << 9.0000 << endl;
21 } // end main
```

Fig. 15.13 | Controlling the printing of trailing zeros and decimal points in floating-point values. (Part 1 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Fig. 15.13 | Controlling the printing of trailing zeros and decimal points in floating-point values. (Part 2 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.2 Justification (`left`, `right` and `internal`)

- Stream manipulators `left` and `right` enable fields to be left justified with padding characters to the right or right justified with padding characters to the left, respectively.
- The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 15.7.3).
- Figure 15.14 uses the `setw`, `left` and `right` manipulators to left justify and right justify integer data in a field.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.14: Fig15_14.cpp
2  // Left and right justification with stream manipulators left and right.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int x = 12345;
10
11    // display x right justified (default)
12    cout << "Default is right justified:" << endl
13       << setw( 10 ) << x;
14
15    // use left manipulator to display x left justified
16    cout << "\n\nUse std::left to left justify x:\n"
17       << left << setw( 10 ) << x;
18
19    // use right manipulator to display x right justified
20    cout << "\n\nUse std::right to right justify x:\n"
21       << right << setw( 10 ) << x << endl;
22 } // end main
```

Fig. 15.14 | Left justification and right justification with stream manipulators `left` and `right`. (Part 1 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

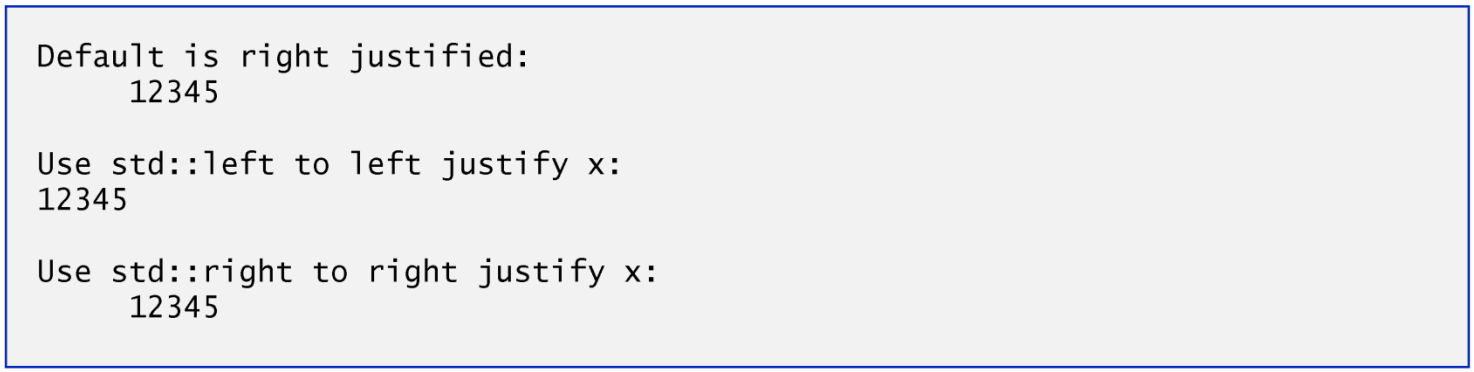

Fig. 15.14 | Left justification and right justification with stream manipulators `left` and `right`. (Part 2 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.2 Justification (`left`, `right` and `internal`) (cont.)

- Stream manipulator `internal` indicates that a number's sign (or base when using stream manipulator `showbase`) should be left justified within a field, that the number's magnitude should be right justified and that intervening spaces should be padded with the fill character.
- Figure 15.15 shows the `internal` stream manipulator specifying internal spacing (line 10).
- Note that `showpos` forces the plus sign to print (line 10).
- To reset the `showpos` setting, output the stream manipulator `noshowpos`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.15: Fig15_15.cpp
2  // Printing an integer with internal spacing and plus sign.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // display value with internal spacing and plus sign
10    cout << internal << showpos << setw( 10 ) << 123 << endl;
11 } // end main
```

```
+       123
```

Fig. 15.15 | Printing an integer with internal spacing and plus sign.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.3 Padding (`fill`, `setfill`)

- The `fill` member function specifies the fill character to be used with justified fields; if no value is specified, spaces are used for padding.
- The `fill` function returns the prior padding character.
- The `setfill` manipulator also sets the padding character.
- Figure 15.16 demonstrates using member function `fill` (line 30) and stream manipulator `setfill` (lines 34 and 37) to set the fill character.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.16: Fig15_16.cpp
2  // Using member function fill and stream manipulator setfill to change
3  // the padding character for fields larger than the printed value.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    int x = 10000;
11
12    // display x
13    cout << x << " printed as int right and left justified\n"
14       << "and as hex with internal justification.\n"
15       << "Using the default pad character (space):" << endl;
16
17    // display x with base
18    cout << showbase << setw( 10 ) << x << endl;
19
20    // display x with left justification
21    cout << left << setw( 10 ) << x << endl;
```

Fig. 15.16 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed. (Part 1 of 3.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
22
23      // display x as hex with internal justification
24      cout << internal << setw( 10 ) << hex << x << endl << endl;
25
26      cout << "Using various padding characters:" << endl;
27
28      // display x using padded characters (right justification)
29      cout << right;
30      cout.fill( '*' );
31      cout << setw( 10 ) << dec << x << endl;
32
33      // display x using padded characters (left justification)
34      cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
35
36      // display x using padded characters (internal justification)
37      cout << internal << setw( 10 ) << setfill( '^' ) << hex
38         << x << endl;
39   } // end main
```

Fig. 15.16 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed. (Part 2 of 3.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

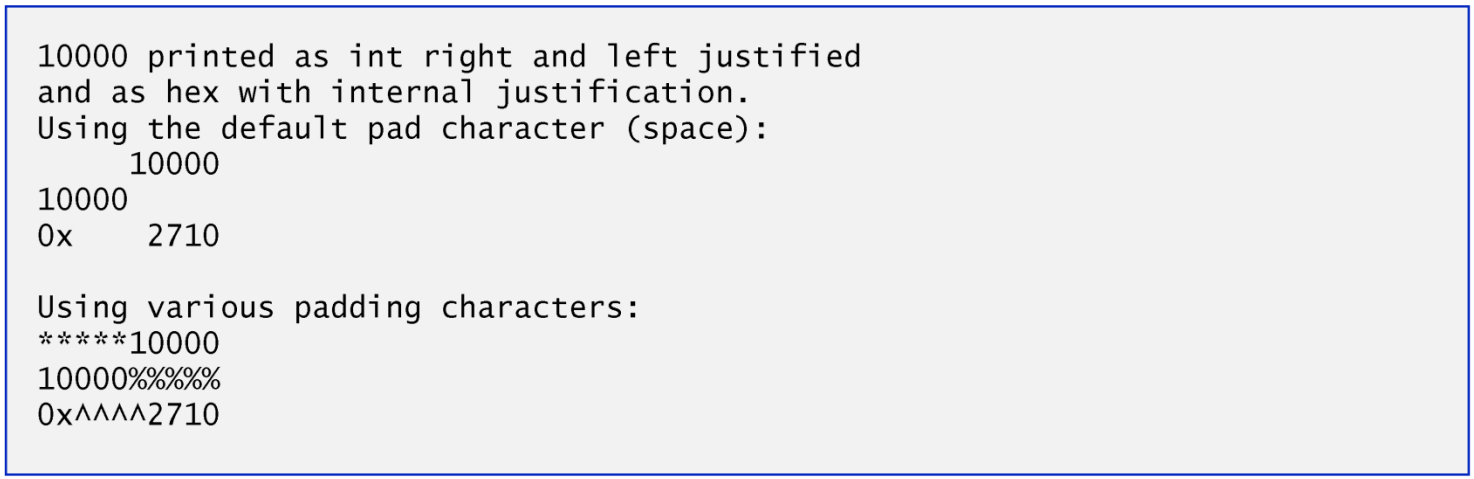

Fig. 15.16 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed. (Part 3 of 3.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.4 Integral Stream Base (dec, oct, hex, showbase)

- C++ provides stream manipulators `dec`, `hex` and `oct` to specify that integers are to be displayed as decimal, hexadecimal and octal values, respectively.
- Stream insertions default to decimal if none of these manipulators is used.
- With stream extraction, integers prefixed with 0 (zero) are treated as octal values, integers prefixed with 0x or 0X are treated as hexadecimal values, and all other integers are treated as decimal values.
- Once a particular base is specified for a stream, all integers on that stream are processed using that base until a different base is specified or until the program terminates.
- Stream manipulator `showbase` forces the base of an integral value to be output.
- Decimal numbers are output by default, octal numbers are output with a leading 0, and hexadecimal numbers are output with either a leading 0x or a leading 0X.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.4 Integral Stream Base (`dec`, `oct`, `hex`, `showbase`) (cont.)

- Figure 15.17 demonstrates the use of stream manipulator `showbase` to force an integer to print in decimal, octal and hexadecimal formats.
- To reset the `showbase` setting, output the stream manipulator `noshowbase`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.17: Fig15_17.cpp
2  // Using stream manipulator showbase.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 100;
9
10    // use showbase to show number base
11    cout << "Printing integers preceded by their base:" << endl
12       << showbase;
13
14    cout << x << endl; // print decimal value
15    cout << oct << x << endl; // print octal value
16    cout << hex << x << endl; // print hexadecimal value
17 } // end main
```

```
Printing integers preceded by their base:
100
0144
0x64
```

Fig. 15.17 | Stream manipulator `showbase`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.5 Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)

- Stream manipulators `scientific` and `fixed` control the output format of floating-point numbers.
- Stream manipulator `scientific` forces the output of a floating-point number to display in scientific format.
- Stream manipulator `fixed` forces a floating-point number to dis-play a specific number of digits (as specified by member function `precision` or stream manipulator `setprecision`) to the right of the decimal point.
- Without using another manipulator, the floating-pointnumber value determines the output format.
- Figure 15.18 demonstrates displaying floating-point numbers in fixed and scientific formats using stream manipulators `scientific` (line 18) and `fixed` (line 22).

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.18: Fig15_18.cpp
2  // Displaying floating-point values in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     double x = 0.001234567;
10    double y = 1.946e9;
11
12    // display x and y in default format
13    cout << "Displayed in default format:" << endl
14       << x << '\t' << y << endl;
15
16    // display x and y in scientific format
17    cout << "\nDisplayed in scientific format:" << endl
18       << scientific << x << '\t' << y << endl;
19
20    // display x and y in fixed format
21    cout << "\nDisplayed in fixed format:" << endl
22       << fixed << x << '\t' << y << endl;
23 } // end main
```

Fig. 15.18 | Floating-point values displayed in default, scientific and fixed formats. (Part 1 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

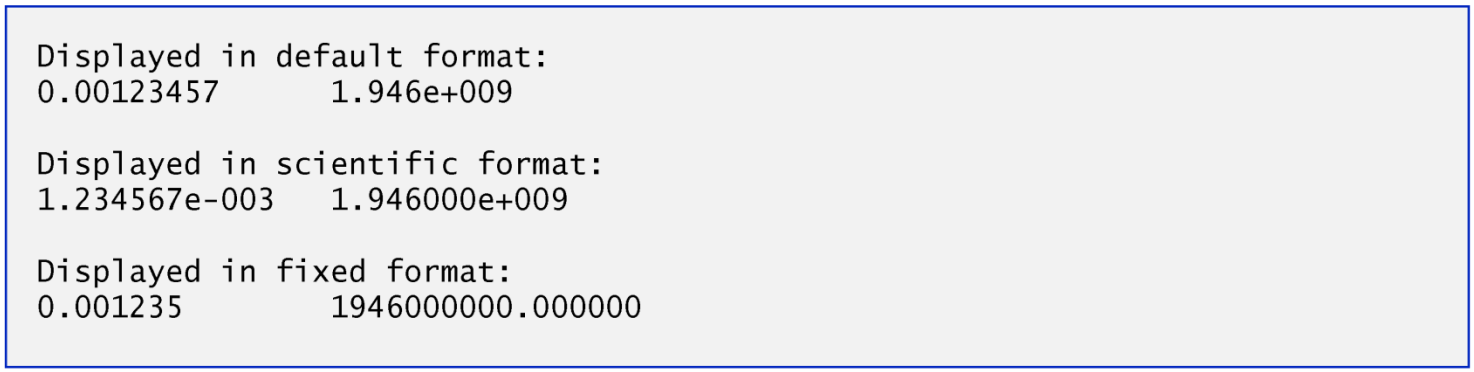

Fig. 15.18 | Floating-point values displayed in default, scientific and fixed formats. (Part 2 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.6 Uppercase/Lowercase Control (`uppercase`)

- Stream manipulator `uppercase` outputs an uppercase `X` or `E` with hexadecimal-integer values or with scientific notation floating-point values, respectively (Fig. 15.19).
- Using stream manipulator `uppercase` also causes all letters in a hexadecimal value to be uppercase.
- By default, the letters for hexadecimal values and the exponents in scientific notation floating-point values appear in lowercase.
- To reset the `uppercase` setting, output the stream manipulator `nouppercase`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.19: Fig15_19.cpp
2  // Stream manipulator uppercase.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     cout << "Printing uppercase letters in scientific" << endl
9        << "notation exponents and hexadecimal values:" << endl;
10
11    // use std:uppercase to display uppercase letters; use std::hex and
12    // std::showbase to display hexadecimal value and its base
13    cout << uppercase << 4.345e10 << endl
14       << hex << showbase << 123456789 << endl;
15 } // end main
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
0X75BCD15
```

Fig. 15.19 | Stream manipulator `uppercase`.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

Good Programming Practice 15.1

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.20: Fig15_20.cpp
2  // Demonstrating stream manipulators boolalpha and noboolalpha.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     bool booleanValue = true;
9
10    // display default true booleanValue
11    cout << "booleanValue is " << booleanValue << endl;
12
13    // display booleanValue after using boolalpha
14    cout << "booleanValue (after using boolalpha) is "
15       << boolalpha << booleanValue << endl << endl;
16
17    cout << "switch booleanValue and use noboolalpha" << endl;
18    booleanValue = false; // change booleanValue
19    cout << noboolalpha << endl; // use noboolalpha
20
21    // display default false booleanValue after using noboolalpha
22    cout << "booleanValue is " << booleanValue << endl;
23
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
24      // display booleanValue after using boolalpha again
25      cout << "booleanValue (after using boolalpha) is "
26         << boolalpha << booleanValue << endl;
27   } // 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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.9.8 Setting and Resetting the Format State via Member Function `flags`

- We now discuss how to return an output stream's format to its default state after having applied several manipulations.
- Member function `flags` without an argument returns the current format settings as a `fmtflags` data type (of class `ios_base`), which represents the format state.
- Member function `flags` with a `fmtflags` argument sets the format state as specified by the argument and returns the prior state settings.
- The initial set-tings of the value that `flags` returns might differ across several systems.
- The program of Fig. 15.21 uses member function `flags` to save the stream's original format state (line 17), then restore the original format settings (line 25).

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.21: Fig15_21.cpp
2  // Demonstrating the flags member function.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int integerValue = 1000;
9     double doubleValue = 0.0947628;
10
11    // display flags value, int and double values (original format)
12    cout << "The value of the flags variable is: " << cout.flags()
13       << "\nPrint int and double in original format:\n"
14       << integerValue << '\t' << doubleValue << endl << endl;
15
16    // use cout flags function to save original format
17    ios_base::fmtflags originalFormat = cout.flags();
18    cout << showbase << oct << scientific; // change format
19
20    // display flags value, int and double values (new format)
21    cout << "The value of the flags variable is: " << cout.flags()
22       << "\nPrint int and double in a new format:\n"
23       << integerValue << '\t' << doubleValue << endl << endl;
```

Fig. 15.21 | `flags` member function. (Part 1 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
24
25      cout.flags( originalFormat ); // restore format
26
27      // display flags value, int and double values (original format)
28      cout << "The restored value of the flags variable is: "
29         << cout.flags()
30         << "\nPrint values in original format again:\n"
31         << integerValue << '\t' << doubleValue << endl;
32   } // end main
```

```
The value of the flags variable is: 513
Print int and double in original format:
1000    0.0947628

The value of the flags variable is: 012011
Print int and double in a new format:
01750   9.476280e-002

The restored value of the flags variable is: 513
Print values in original format again:
1000    0.0947628
```

Fig. 15.21 | `flags` member function. (Part 2 of 2.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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 encoun-tered.
- 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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 15.22: Fig15_22.cpp
2  // Testing error states.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int integerValue;
9
10    // display results of cin functions
11    cout << "Before a bad input operation:"
12       << "\ncin.rdstate(): " << cin.rdstate()
13       << "\n    cin.eof(): " << cin.eof()
14       << "\n   cin.fail(): " << cin.fail()
15       << "\n    cin.bad(): " << cin.bad()
16       << "\n   cin.good(): " << cin.good()
17       << "\n\nExpects an integer, but enter a character: ";
18
19    cin >> integerValue; // enter character value
20    cout << endl;
21
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
22      // display results of cin functions after bad input
23      cout << "After a bad input operation:"
24         << "\ncin.rdstate(): " << cin.rdstate()
25         << "\n    cin.eof(): " << cin.eof()
26         << "\n   cin.fail(): " << cin.fail()
27         << "\n    cin.bad(): " << cin.bad()
28         << "\n   cin.good(): " << cin.good() << endl << endl;
29
30      cin.clear(); // clear stream
31
32      // display results of cin functions after clearing cin
33      cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
34         << "\ncin.good(): " << cin.good() << endl;
35   } // end main
```

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

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

```
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.)

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

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.

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.

15.13 Tying an Output Stream to an Input Stream

- Interactive applications generally involve an `istream` for input and an `ostream` for output.
- When a prompting message appears on the screen, the user responds by entering the appropriate data.
- Obviously, the prompt needs to appear before the input operation proceeds.
- With output buffering, outputs appear only when the buffer fills, when outputs are flushed explicitly by the program or automatically at the end of the program.
- C++ provides member function `tie` to synchronize (i.e., "tie together") the operation of an `istream` and an `ostream` to ensure that outputs appear before their subsequent inputs.
- The call
  - `cin.tie( &cout );`
- ties `cout` (an `ostream`) to `cin` (an `istream`).

©1992–2011 by Pearson Education, Inc.
All Rights Reserved.