Stream Input/Output

C How to Program, 6/e

23.1 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 (>>).

Software Engineering Observation 23.1

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

Error-Prevention Tip 23.1

C++ I/O is type safe.

Software Engineering Observation 23.2

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

23.2 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.

23.3 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 23.1

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

Portability Tip 23.1

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

23.3.1 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.

23.3.1 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.

23.3.2 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.

23.3.3 `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.

23.3.4 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.

23.3.4 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.

23.3.4 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. 23.1 summarizes these inheritance relationships.

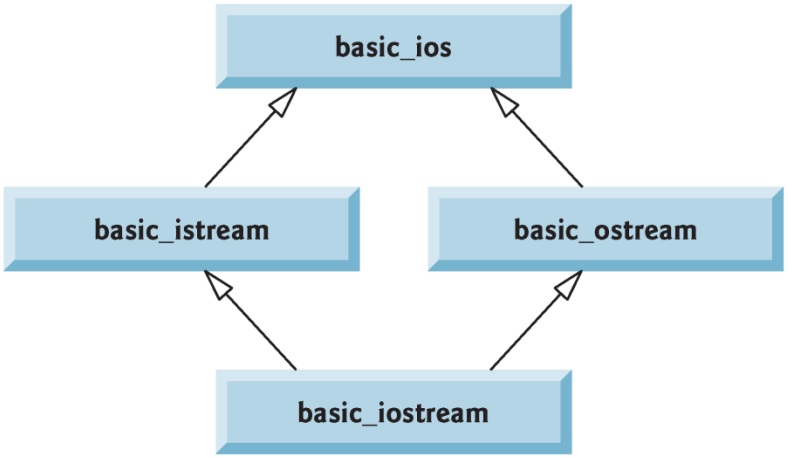

Fig. 23.1 | Stream-I/O template hierarchy portion.

23.3.4 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.

23.3.4 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.

23.3.4 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.

23.3.4 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. 23.2 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.

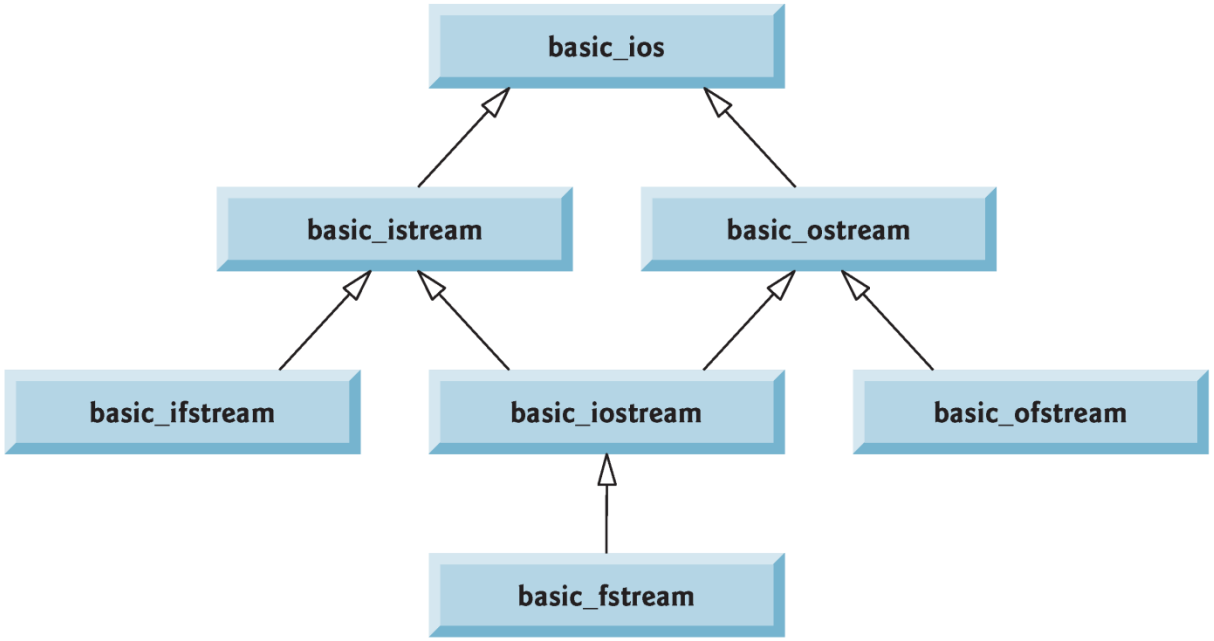

Fig. 23.2 | Stream-I/O template hierarchy portion showing the main file-processing templates.

23.4 Stream Output

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

23.4.1 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 23.3 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.

```
1  // Fig. 23.3: Fig23_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. 23.3 | Printing the address stored in a `char *` variable.

23.4.2 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`.

23.5 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.

23.6 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.

23.6.1 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 23.4 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.

```
1  // Fig. 23.4: Fig23_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. 23.4 | `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. 23.4 | `get`, `put` and `eof` member functions. (Part 2 of 2.)

23.6.1 `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).

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

- Figure 23.5 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.

```
1  // Fig. 23.5: Fig23_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. 23.5 | Input of a string using `cin` with stream extraction contrasted with input using `cin.get`. (Part 1 of 2.)

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

23.6.1 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. 23.6 demonstrates the use of the `getline` member function to input a line of text (line 13).

```
1  // Fig. 23.6: Fig23_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. 23.6 | 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. 23.6 | Inputting character data with `cin` member function `getline`. (Part 2 of 2.)

23.6.2 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.

23.6.3 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.”

23.7 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 23.8 shows how to determine whether `failbit` has been set.
- Member function `gcount` reports the number of characters read by the last input operation.

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

- Figure 23.7 demonstrates `istream` member functions `read` and `gcount`, and `ostream` member function `write`.

```
1  // Fig. 23.7: Fig23_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. 23.7 | 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. 23.7 | Unformatted I/O using the `read`, `gcount` and `write` member functions. (Part 2 of 2.)

23.8 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.

23.8.1 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 23.8 demonstrates stream manipulators `hex`, `oct`, `dec` and `setbase`.

```
1  // Fig. 23.8: Fig23_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. 23.8 | Stream manipulators `hex`, `oct`, `dec` and `setbase`. (Part 1 of 2.)

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

Fig. 23.8 | Stream manipulators `hex`, `oct`, `dec` and `setbase`. (Part 2 of 2.)

23.8.2 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. 23.9 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`.

```
1  // Fig. 23.9: Fig23_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. 23.9 | Precision of floating-point values. (Part 1 of 3.)

```
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. 23.9 | 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. 23.9 | Precision of floating-point values. (Part 3 of 3.)

23.8.3 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 23.10 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 23.1

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 23.2

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

```
1  // Fig. 23.10: Fig23_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. 23.10 | `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. 23.10 | `width` member function of class `ios_base`. (Part 2 of 2.)

Input/Output in C++

- A basic introduction to FILE I/O in C++ is available at http://www.cplusplus.com/doc/tutorial/files