Performance Tip 15.1

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

Portability Tip 15.1

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

15.2.1 Classic Streams vs. Standard Streams

- **C++ classic stream libraries**
  - **Enable input and output of `chars` (single bytes)**
- **ASCII character set**
  - **Uses single bytes**
  - **Represents only a limited set of characters**
- **Unicode character set**
  - **Represents most of the world's commercially viable languages, mathematical symbols and more**
  - **www.unicode.org**

15.2.1 Classic Streams vs. Standard Streams (Cont.)

- **C++ standard stream libraries**
  - **Enables I/O operations with Unicode characters**
  - **Class template versions of classic C++ stream classes**
    - **Specializations for processing characters of types `char` and `wchar_t`**
      - **`wchar_t`s can store Unicode characters**

15.2.3 Stream Input/Output Classes and Objects (Cont.)

- typedefs in <iostream> library
  - istream
    - Represents a specialization of basic_istream
    - Enables char input
  - ostream
    - Represents a specialization of basic_ostream
    - Enables char output
  - iostream
    - Represents a specialization of basic_iostream
    - Enables char input and output

15.2.3 Stream Input/Output Classes and Objects (Cont.)

- **Stream-I/O template hierarchy**
  - `basic_istream` and `basic_ostream` derive from `basic_ios`
  - `basic_iostream` derives from `basic_istream` and `basic_ostream`
    - Uses multiple inheritance
- **Stream operator overloading**
  - Stream insertion operator
    - Left-shift operator (<<) is overloaded for stream output
  - Stream extraction operator
    - Right-shift operator(>>) is overloaded for stream input

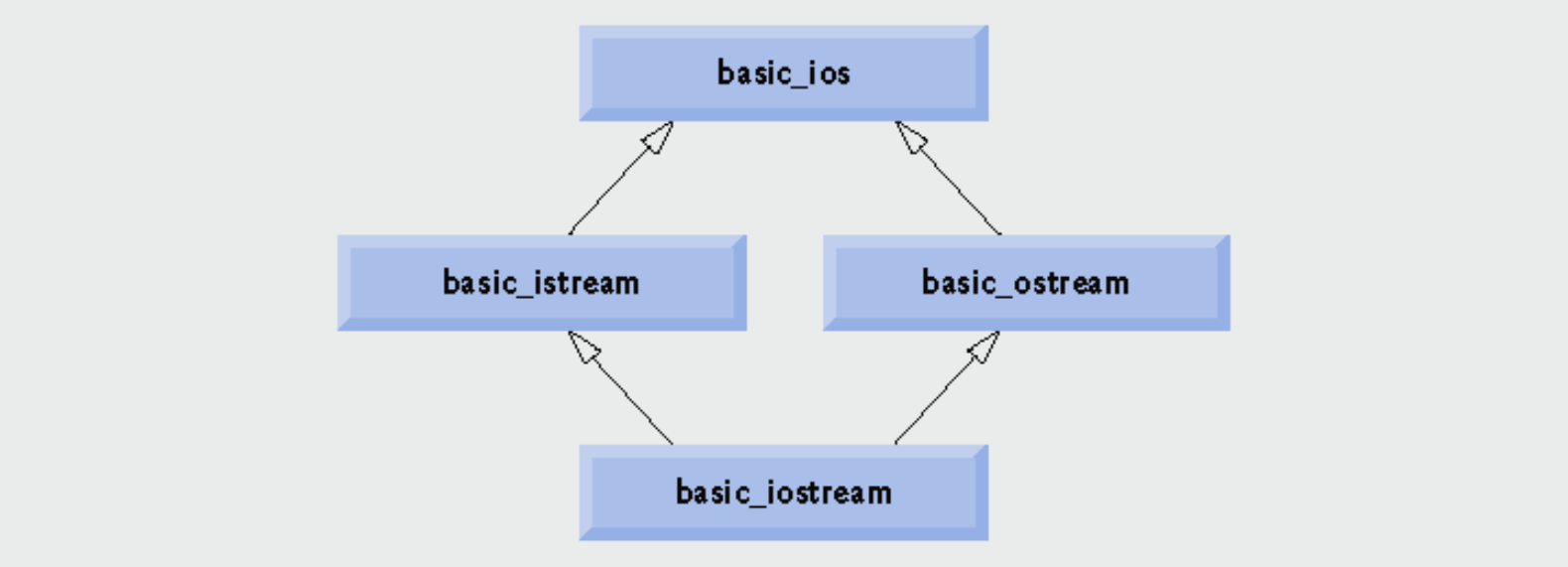

Fig. 15.1 | Stream-I/O template hierarchy portion.

15.2.3 Stream Input/Output Classes and Objects (Cont.)

- **Standard stream objects**
  - `istream` **instance**
    - `cin`
      - **Connected to the standard input device, usually the keyboard**
  - `ostream` **instances**
    - `cout`
      - **Connected to the standard output device, usually the display screen**
    - `cerr`
      - **Connected to the standard error device**
      - **Unbuffered - output appears immediately**
    - `clog`
      - **Connected to the standard error device**
      - **Buffered - output is held until the buffer is filled or flushed**

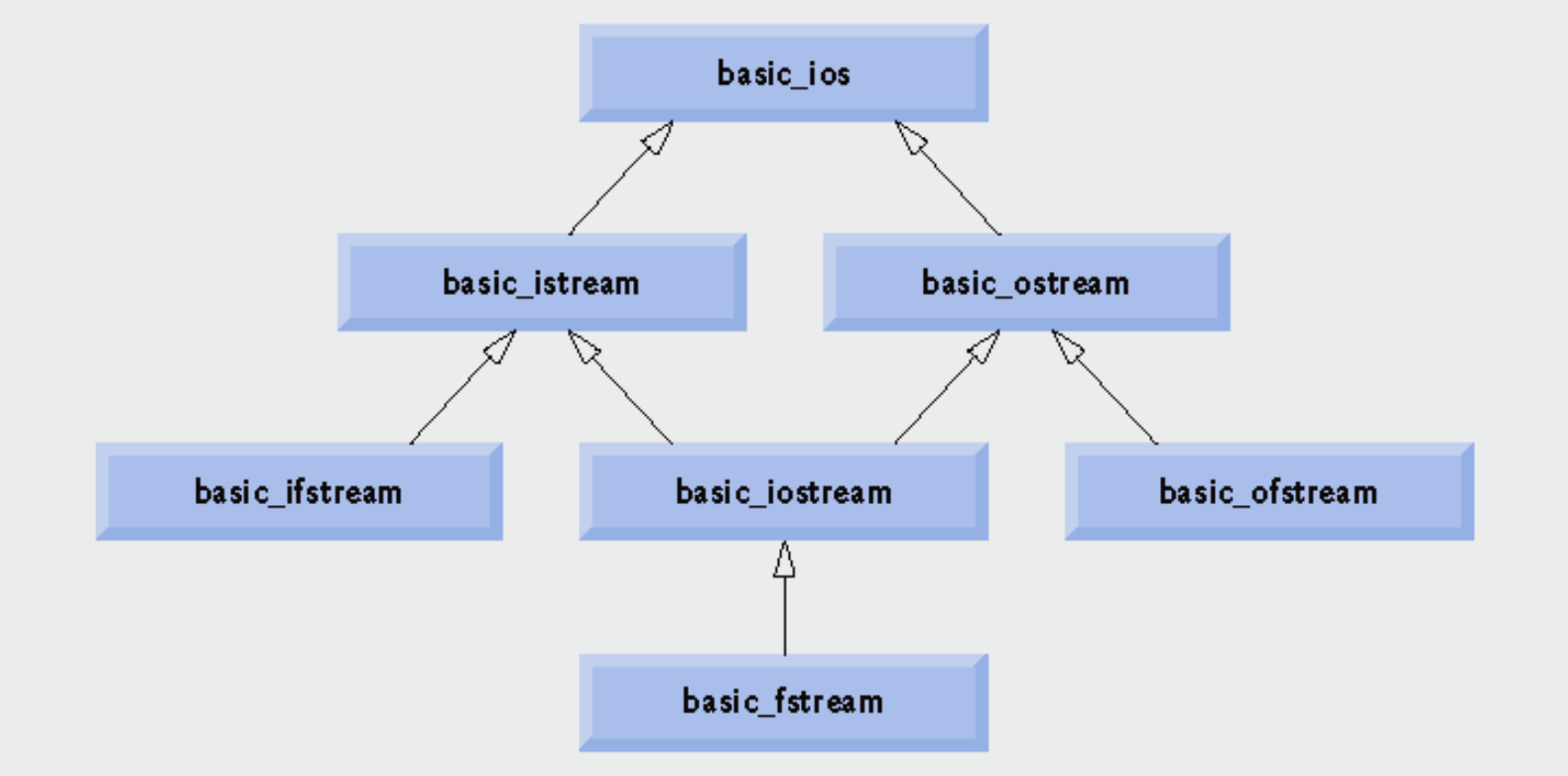

Fig. 15.2 | Stream-I/O template hierarchy portion showing the main file-processing templates.

15.3 Stream Output

- `ostream` output capabilities
  - Can output
    - Standard data types
    - Characters
    - Unformatted data
    - Integers
    - Floating-point values
    - Values in fields

15.3.1 Output of `char *` Variables

- **Outputting `char *` (memory address of a `char`)**
  - **Cannot use << operator**
    - **Has been overloaded to print `char *` as a null-terminated string**
  - **Solution**
    - **Cast the `char *` to a `void *`**
  - **Address is printed as a hexadecimal (base-16) number**

Outline

```
1  // Fig. 15.3: Fig15_03.cpp
2  // Printing the address stored in a char * variable.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char *word = "again";
10
11    // display value of char *, then display value of char *
12    // static_cast to void *
13    cout << "Value of word is: " << word << endl
14       << "Value of static_cast< void * >( word ) is: "
15       << static_cast< void * >( word ) << endl;
16    return 0;
17 } // end main
```

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

Fig15_03.cpp

(1 of 1)

Cast the **char *** to a **void ***

Address prints as a hexadecimal (base-16) number

15.3.2 Character Output using Member Function `put`

- `ostream` member function `put`
  - Outputs a character
  - Returns a reference to the same `ostream` object
    - Can be cascaded
  - Can be called with a numeric expression that represents an ASCII value
  - Examples
    - `cout.put( 'A' );`
    - `cout.put( 'A' ).put( '\n' );`
    - `cout.put( 65 );`

15.4 Stream Input

- `istream` input capabilities
  - Stream extraction operator (overloaded >> operator)
    - Skips over white-space characters
    - Returns a reference to the `istream` object
  - When used as a condition, `void *` cast operator is implicitly invoked
    - Converts to non-null pointer (`true`) or null pointer (`false`)
      - Based on success or failure of last input operation
        - An attempt to read past end of stream is one such failure

15.4 Stream Input (Cont.)

- `istream` input capabilities (Cont.)
  - State bits
    - Control the state of the stream
    - `failbit`
      - Set if input data is of wrong type
    - `badbit`
      - Set if stream extraction operation fails

15.4.1 `get` and `getline` Member Functions

- `istream` member function `get`
  - With no arguments
    - Returns one character input from the stream
      - Any character, including white-space and non-graphic characters
    - Returns `EOF` when end-of-file is encountered
  - With a character-reference argument
    - Stores input character in the character-reference argument
    - Returns a reference to the `istream` object

15.4.1 `get` and `getline` Member Functions (Cont.)

- `istream` member function `get` (Cont.)
  - With three arguments: a character array, a size limit and a delimiter (default delimiter is `'\n'`)
    - Reads and stores characters in the character array
    - Terminates at one fewer characters than the size limit or upon reading the delimiter
      - Delimiter is left in the stream, not placed in array
    - Null character is inserted after end of input in array
- `istream` member function `eof`
  - Returns `false` when end-of-file has not occurred
  - Returns `true` when end-of-file has occurred

Outline

Fig15_04.cpp

(1 of 2)

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

Call **eof** member function before end-of-file is reached

while loop terminates when **get** member function returns **EOF**

Outline

```
19
20    // display end-of-file character
21    cout << "\nEOF in this system is: " << character << endl;
22    cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
23    return 0;
24 } // end main
```

Fig15_04.cpp

```
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
```

Display **character**, which currently contains the value of **EOF**

Call **eof** member function after end-of-file is reached

End-of-file is represented by <*ctrl*>-*z* on Microsoft Windows systems, <*ctrl*>-*d* on UNIX and Macintosh systems.

Outline

Fig15_05.cpp

(1 of 2)

```
1  // Fig. 15.5: Fig15_05.cpp
2  // Contrasting input of a string via cin and cin.get.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    // create two char arrays, each with 80 elements
11    const int SIZE = 80;
12    char buffer1[ SIZE ];
13    char buffer2[ SIZE ];
14
15    // use cin to input characters into buffer1
16    cout << "Enter a sentence:" << endl;
17    cin >> buffer1;
18
19    // display buffer1 contents
20    cout << "\nThe string read with cin was:" << endl
21       << buffer1 << endl << endl;
22
23    // use cin.get to input characters into buffer2
24    cin.get( buffer2, SIZE );
25
26    // display buffer2 contents
27    cout << "The string read with cin.get was:" << endl
28       << buffer2 << endl;
29    return 0;
30 } // end main
```

Use stream extraction with **cin**

Call three-argument version of member function **get** (third argument is default value **'\n'**)

```
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
```

Stream extraction operation reads up to first white-space character

get member function reads up to the delimiter character **'\n'**

Outline

Fig15_05.cpp

(1 of 2)

15.4.1 `get` and `getline` Member Functions (Cont.)

- `istream` member function `getline`
  - (Similar to the three-argument version of `get`
    - Except the delimiter *is* removed from the stream)
  - Three arguments: a character array, a size limit and a delimiter (default delimiter is `'\n'`)
    - Reads and stores characters in the character array
    - Terminates at one fewer characters than the size limit or upon reading the delimiter
      - Delimiter is removed from the stream, but not placed in the array
    - Null character is inserted after end of input in array

Outline

Fig15_06.cpp

(1 of 1)

```
1  // Fig. 15.6: Fig15_06.cpp
2  // Inputting characters using cin member function getline.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    const int SIZE = 80;
11    char buffer[ SIZE ]; // create array of 80 characters
12
13    // input characters in buffer via cin function getline
14    cout << "Enter a sentence:" << endl;
15    cin.getline( buffer, SIZE );
16
17    // display buffer contents
18    cout << "\nThe sentence entered is:" << endl << buffer << endl;
19    return 0;
20 } // end main
```

Call member function **getline**

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

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

15.4.2 istream Member Functions peek, putback and ignore

- istream member function ignore
  - Reads and discards a designated number of characters or terminates upon encountering a designated delimiter
    - Default number of characters is one
    - Default delimiter is EOF
- istream member function putback
  - Places previous character obtained by a get from the input stream back into the stream
- istream member function peek
  - Returns the next character in the input stream, but does not remove it from the stream

15.4.3 Type-Safe I/O

- **C++ offers type-safe I/O**
  - **<< and >> operators are overloaded to accept data of specific types**
    - **Attempts to input or output a user-defined type that << and >> have not been overloaded for result in compiler errors**
  - **If unexpected data is processed, error bits are set**
    - **User may test the error bits to determine I/O operation success or failure**
  - **The program is able to "stay in control"**

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

- `istream` member function `read`
  - Inputs some number of bytes to a character array
  - If fewer characters are read than the designated number, `failbit` is set
- `istream` member function `gcount`
  - Reports number of characters read by last input operation
- `ostream` member function `write`
  - Outputs some number of bytes from a character array

Outline

Fig15_07.cpp

(1 of 1)

```
1  // Fig. 15.7: Fig15_07.cpp
2  // Unformatted I/O using read, gcount and write.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    const int SIZE = 80;
11    char buffer[ SIZE ]; // create array of 80 characters
12
13    // use function read to input characters into buffer
14    cout << "Enter a sentence:" << endl;
15    cin.read( buffer, 20 );
16
17    // use functions write and gcount to display buffer characters
18    cout << endl << "The sentence entered was:" << endl;
19    cout.write( buffer, cin.gcount() );
20    cout << endl;
21    return 0;
22 } // end main
```

`read` **`20`** bytes from the input stream to **`buffer`**

`write` out as many characters as were read by the last input operation from **`buffer`** to the output stream

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

15.6 Introduction to Stream Manipulators

- **Stream manipulators perform formatting tasks**
  - **Setting field widths**
  - **Setting precision**
  - **Setting and unsetting format state**
  - **Setting fill characters in fields**
  - **Flushing streams**
  - **Inserting a newline and flushing the output stream**
  - **Inserting a null character and skipping white space in the input stream**

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

- **Change a stream's integer base by inserting manipulators**
  - `hex` **manipulator**
    - **Sets the base to hexadecimal (base 16)**
  - `oct` **manipulator**
    - **Sets the base to octal (base 8)**
  - `dec` **manipulator**
    - **Resets the base to decimal**
  - `setbase` **parameterized stream manipulator**
    - **Takes one integer argument:** `10`, `8` **or** `16`
    - **Sets the base to decimal, octal or hexadecimal**
    - **Requires the inclusion of the** `<iomanip>` **header file**
  - **Stream base values are sticky**
    - **Remain until explicitly changed to another base value**

Outline

Fig15_08.cpp

(1 of 2)

```
1  // Fig. 15.8: Fig15_08.cpp
2  // Using stream manipulators hex, oct, dec and setbase.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::dec;
7  using std::endl;
8  using std::hex;
9  using std::oct;
10
11 #include <iomanip>
12 using std::setbase;
13
```

Parameterized stream manipulator **setbase** is in header file **<iomanip>**

Outline

Fig15_08.cpp

(2 of 2)

```
14 int main()
15 {
16    int number;
17
18    cout << "Enter a decimal number: ";
19    cin >> number; // input number
20
21    // use hex stream manipulator to show hexadecimal number
22    cout << number << " in hexadecimal is: " << hex
23       << number << endl;
24
25    // use oct stream manipulator to show octal number
26    cout << dec << number << " in octal is: "
27       << oct << number << endl;
28
29    // use setbase stream manipulator to show decimal number
30    cout << setbase( 10 ) << number << " in decimal is: "
31       << number << endl;
32    return 0;
33 } // end main
```

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

Set base to hexadecimal

Set base to octal

Reset base to decimal

15.6.2 Floating-Point Precision (`precision`, `setprecision`)

- **Precision of floating-point numbers**
  - **Number of digits displayed to the right of the decimal point**
  - **`setprecision` parameterized stream manipulator**
  - **`precision` member function**
    - **When called with no arguments, returns the current precision setting**
  - **Precision settings are sticky**

Outline

Fig15_09.cpp

(1 of 2)

```
1  // Fig. 15.9: Fig15_09.cpp
2  // Controlling precision of floating-point values.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include <cmath>
12 using std::sqrt; // sqrt prototype
13
14 int main()
15 {
16    double root2 = sqrt( 2.0 ); // calculate square root of 2
17    int places; // precision, vary from 0-9
18
19    cout << "Square root of 2 with precisions 0-9." << endl
20       << "Precision set by ios_base member function "
21       << "precision:" << endl;
22
23    cout << fixed; // use fixed-point notation
24
25    // display square root using ios_base function precision
26    for ( places = 0; places <= 9; places++ )
27    {
28       cout.precision( places );
29       cout << root2 << endl;
30    } // end for
```

Use member function **precision** to set **cout** to display **places** digits to the right of the decimal point

Outline

Fig15_09.cpp

(2 of 2)

```
31
32    cout << "\nPrecision set by stream manipulator "
33       << "setprecision:" << endl;
34
35    // set precision for each digit, then display square root
36    for ( places = 0; places <= 9; places++ )
37       cout << setprecision( places ) << root2 << endl;
38
39    return 0;
40 } // end main
```

Use parameterized stream manipulator **setprecision** to set **cout** to display **places** digits to the right of the decimal point

```
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

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
```

15.6.3 Field Width (`width`, `setw`)

- **Field width**
  - **(for `ostream`) Number of character positions in which value is outputted**
    - **Fill characters are inserted as padding**
    - **Values wider than the field are not truncated**
  - **(for `istream`) Maximum number of characters inputted**
    - **For `char` array, maximum of one fewer characters than the width will be read (to accommodate null character)**

15.6.3 Field Width (`width`, `setw`) (Cont.)

- **Field width (Cont.)**
  - **Member function `width` of base class `ios_base`**
    - **Sets the field width**
    - **Returns the previous width**
      - **`width` function call with no arguments just returns the current setting**
  - **Parameterized stream manipulator `setw`**
    - **Sets the field width**
  - **Field width settings are not sticky**

Common Programming Error 15.1

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

Common Programming Error 15.2

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

Outline

Fig15_10.cpp

(1 of 2)

```
1  // Fig. 15.10: Fig15_10.cpp
2  // Demonstrating member function width.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int widthValue = 4;
11    char sentence[ 10 ];
12
13    cout << "Enter a sentence:" << endl;
14    cin.width( 5 ); // input only 5 characters from sentence
15
16    // set field width, then display characters based on that width
17    while ( cin >> sentence )
18    {
19       cout.width( widthValue++ );
20       cout << sentence << endl;
21       cin.width( 5 ); // input 5 more characters from sentence
22    } // end while
23
24    return 0;
25 } // end main
```

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

Outline

Fig15_10.cpp

(2 of 2)

15.6.4 Use-Defined Output Stream Manipulators

- **Programmers can create their own stream manipulators**
  - **Output stream manipulators**
    - **Must have return type and parameter type `ostream &`**

Outline

Fig15_11.cpp

(1 of 2)

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

Outline

Fig15_11.cpp

(2 of 2)

```
26
27 // endLine manipulator (using escape sequence \n and member
28 // function flush)
29 ostream& endLine( ostream& output )
30 {
31    return output << '\n' << flush; // issue endl-like end of line
32 } // end endLine manipulator
33
34 int main()
35 {
36    // use tab and endLine manipulators
37    cout << "Testing the tab manipulator:" << endLine
38       << 'a' << tab << 'b' << tab << 'c' << endLine;
39
40    cout << "Testing the carriageReturn and bell manipulators:"
41       << endLine << "..........";
42
43    cout << bell; // use bell manipulator
44
45    // use carriageReturn and endLine manipulators
46    cout << carriageReturn << "-----" << endLine;
47    return 0;
48 } // end main
```

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

15.7 Stream Format States and Stream Manipulators

- **Stream manipulators specify stream-I/O formatting**
  - **All these manipulators belong to class `ios_base`**

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.

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

Stream Manipulator	Description
`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`.
`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 are 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)

15.7.1 Trailing Zeros and Decimal Points (`showpoint`)

- **Stream manipulator `showpoint`**
  - **Floating-point numbers are output with decimal point and trailing zeros**
    - **Example**
      - **`79.0` prints as `79.0000` instead of `79`**
  - **Reset `showpoint` setting with `noshowpoint`**

Outline

Fig15_13.cpp

(1 of 2)

```
1  // Fig. 15.13: Fig15_13.cpp
2  // Using showpoint to control the printing of
3  // trailing zeros and decimal points for doubles.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::showpoint;
8
9  int main()
10 {
11    // display double values with default stream format
12    cout << "Before using showpoint" << endl
13       << "9.9900 prints as: " << 9.9900 << endl
14       << "9.9000 prints as: " << 9.9000 << endl
15       << "9.0000 prints as: " << 9.0000 << endl << endl;
```

```
16
17    // display double value after showpoint
18    cout << showpoint
19       << "After using showpoint" << endl
20       << "9.9900 prints as: " << 9.9900 << endl
21       << "9.9000 prints as: " << 9.9000 << endl
22       << "9.0000 prints as: " << 9.0000 << endl;
23    return 0;
24 } // end main
```

```
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
```

Outline

Fig15_13.cpp

(2 of 2)

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

- **Justification in a field**
  - **Manipulator `left`**
    - **fields are left-justified**
    - **padding characters to the right**
  - **Manipulator `right`**
    - **fields are right-justified**
    - **padding characters to the left**
  - **Manipulator `internal`**
    - **signs or bases on the left**
      - **`showpos` forces the plus sign to print**
    - **magnitudes on the right**
    - **padding characters in the middle**

Outline

Fig15_14.cpp

(1 of 2)

```
1  // Fig. 15.14: Fig15_14.cpp
2  // Demonstrating left justification and right justification.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14    int x = 12345;
15
16    // display x right justified (default)
17    cout << "Default is right justified:" << endl
18       << setw( 10 ) << x;
19
20    // use left manipulator to display x left justified
21    cout << "\n\nUse std::left to left justify x:\n"
22       << left << setw( 10 ) << x;
23
24    // use right manipulator to display x right justified
25    cout << "\n\nUse std::right to right justify x:\n"
26       << right << setw( 10 ) << x << endl;
27    return 0;
28 } // end main
```

```
Default is right justified:
     12345

Use std::left to left justify x:
12345

Use std::right to right justify x:
     12345
```

Outline

Fig15_14.cpp

(2 of 2)

Outline

Fig15_15.cpp

(1 of 1)

```
1  // Fig. 15.15: Fig15_15.cpp
2  // Printing an integer with internal spacing and plus sign.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::internal;
7  using std::showpos;
8
9  #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14    // display value with internal spacing and plus sign
15    cout << internal << showpos << setw( 10 ) << 123 << endl;
16    return 0;
17 } // end main
```

```
+       123
```

15.7.3 Padding (`fill`, `setfill`)

- **Padding in a field**
  - **Fill characters are used to pad a field**
    - **Member function `fill`**
      - **Specifies the fill character**
        - **Spaces are used if no value is specified**
      - **Returns the prior fill character**
    - **Stream manipulator `setfill`**
      - **Specifies the fill character**