

# 25

# Stream Input/Output



*Consciousness... does not appear to itself chopped up in bits... A “river” or a “stream” are the metaphors by which it is most naturally described.*

— William James

*All the news that’s fit to print.*

— Adolph S. Ochs

*Remove not the landmark on the boundary of the fields.*

— Amenehope



# OBJECTIVES

In this chapter you will 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.



- 25.1 Introduction
- 25.2 Streams
  - 25.2.1 Classic Streams vs. Standard Streams
  - 25.2.2 `iostream` Library Header Files
  - 25.2.3 Stream Input/Output Classes and Objects
- 25.3 Stream Output
  - 25.3.1 Output of `char*` Variables
  - 25.3.2 Character Output using Member Function `put`
- 25.4 Stream Input
  - 25.4.1 `get` and `getline` Member Functions
  - 25.4.2 `istream` Member Functions `peek`, `putback` and `ignore`
  - 25.4.3 Type-Safe I/O
- 25.5 Unformatted I/O using `read`, `write` and `gcount`



## 25.6 Introduction to Stream Manipulators

25.6.1 Integral Stream Base: `dec`, `oct`, `hex` and `setbase`

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

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

25.6.4 User-Defined Output Stream Manipulators

## 25.7 Stream Format States and Stream Manipulators

25.7.1 Trailing Zeros and Decimal Points (`showpoint`)

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

25.7.3 Padding (`fill`, `setfill`)

25.7.4 Integral Stream Base (`dec`, `oct`, `hex`, `showbase`)

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

25.7.6 Uppercase/Lowercase Control (`uppercase`)

25.7.7 Specifying Boolean Format (`boolalpha`)

25.7.8 Setting and Resetting the Format State via Member-Function `flags`



**25.8 Stream Error States**

**25.9 Tying an Output Stream to an Input Stream**

**25.10 Wrap-Up**



# 25.1 Introduction

- **C++ standard library input/output capabilities**
  - Many I/O features are object oriented
  - Type-safe I/O
    - I/O operations are sensitive data types
    - Improper data cannot “sneak” through
  - Extensibility allows users to specify I/O for user-defined types
    - Overloading the stream insertion and extraction operators



# Software Engineering Observation 25.1

---

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



# Error-Prevention Tip 25.1

---

**C++ I/O is type safe.**



## Software Engineering Observation 25.2

---

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



## 25.2 Streams

- **C++ I/O occurs in streams – sequences of bytes**
  - **Input**
    - Bytes flow from a device to main memory
  - **Output**
    - Bytes flow from main memory to a device
  - **I/O transfers typically take longer than processing the data**



## 25.2 Streams (Cont.)

- **“Low-level”, unformatted I/O**
  - Individual bytes are the items of interest
  - High-speed, high-volume
  - Not particularly convenient for programmers
- **“High-level”, formatted I/O**
  - Bytes are grouped into meaningful units
    - Integers, floating-point numbers, characters, etc.
  - Satisfactory for most I/O other than high-volume file processing



# Performance Tip 25.1

---

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



## Portability Tip 25.1

---

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



## 25.2.1 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](http://www.unicode.org)



## 25.2.1 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



## 25.2.2 `iostream` Library Header Files

- **`<iostream>` header file**
  - Declares basic services required for all stream-I/O operations
  - Defines `cin`, `cout`, `cerr` and `cllog`
  - Provides both unformatted- and formatted-I/O services
- **`<iomanip>` header file**
  - Declares services for performing formatted I/O with parameterized stream manipulators
- **`<fstream>` header file**
  - Declares services for user-controlled file processing



## 25.2.3 Stream Input/Output Classes and Objects

- **Class templates in the `iostream` library**
  - **`basic_istream`**
    - Supports stream-input operations
  - **`basic_ostream`**
    - Supports stream-output operations
  - **`basic_iostream`**
    - Supports both stream-input and stream-output operations



## 25.2.3 Stream Input/Output Classes and Objects (Cont.)

- **typedefs**

- Declare synonyms for previously defined data types

- Example

- `typedef Card *CardPtr;`

- Makes `CardPtr` a synonym for type `Card *`

- Used to create shorter or more readable type names



## 25.2.3 Stream Input/Output Classes and Objects (Cont.)

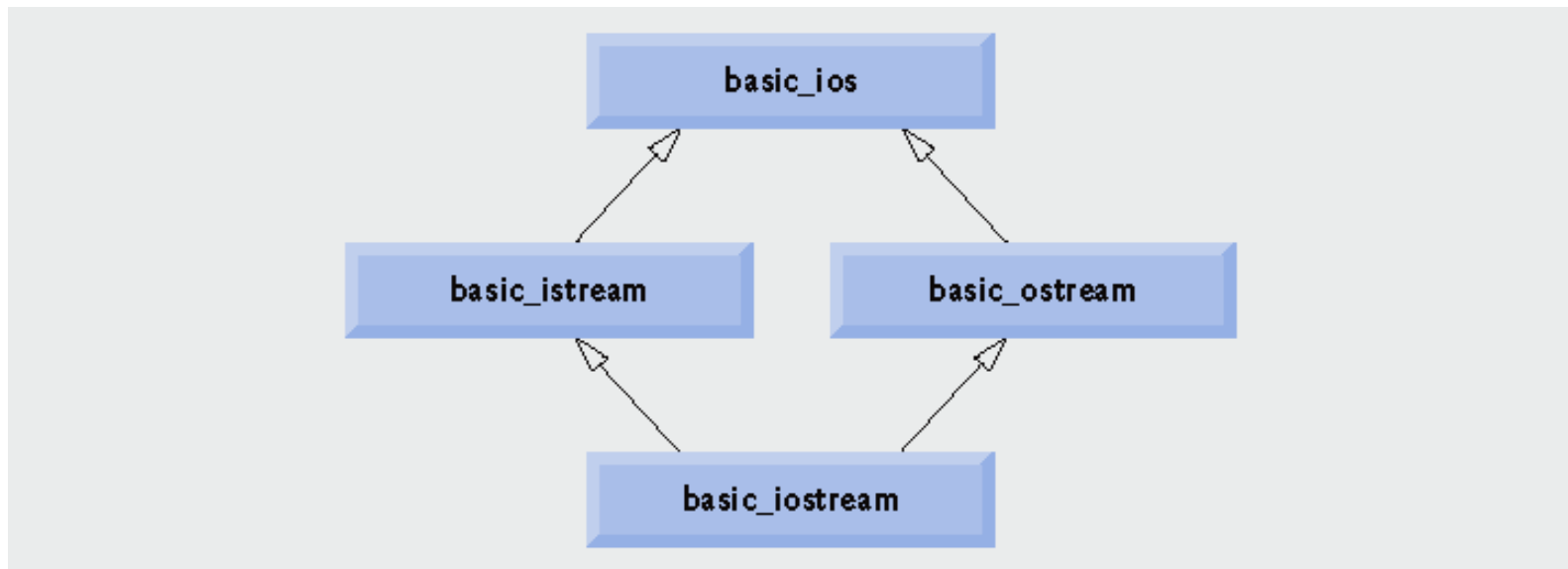
- **typedefs in `<i ostream>` library**
  - **`istream`**
    - Represents a specialization of **basic\_istream**
    - Enables **char** input
  - **`ostream`**
    - Represents a specialization of **basic\_ostream**
    - Enables **char** output
  - **`iostream`**
    - Represents a specialization of **basic\_iostream**
    - Enables **char** input and output



## 25.2.3 Stream Input/Output Classes and Objects (Cont.)

- **Stream-I/O template hierarchy**
  - **basic\_istream** and **basic\_ostream** derive from **basic\_ios**
  - **basic\_istream** 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. 25.1 | Stream-I/O template hierarchy portion.**



## 25.2.3 Stream Input/Output Classes and Objects (Cont.)

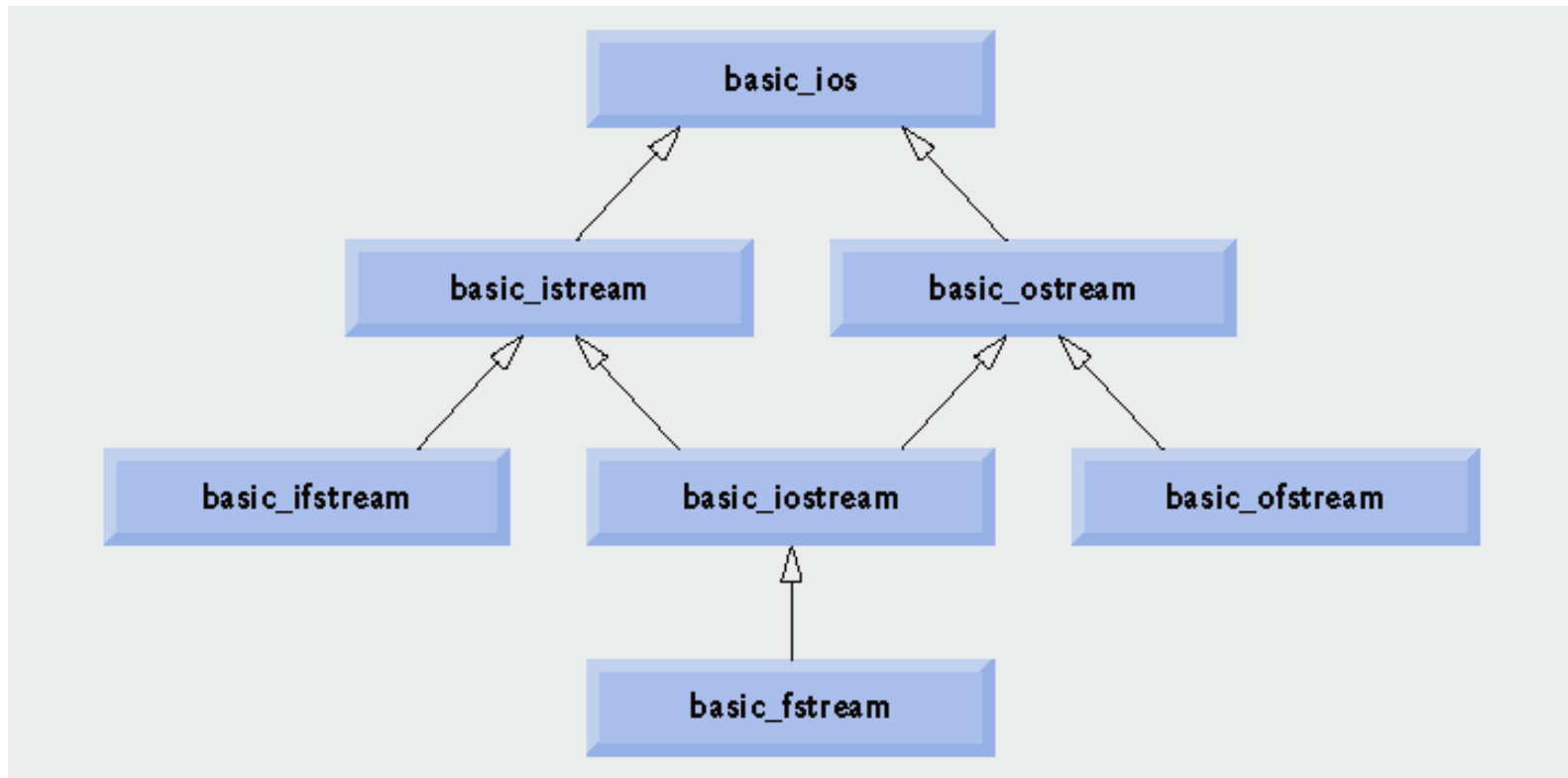
- **Standard stream objects**
  - **istream** instance
    - **cin**
      - Connected to the standard input device, usually the keyboard
  - **ostream** instances
    - **cout**
      - Connected to the standard output device, usually the display screen
    - **cerr**
      - Connected to the standard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



## 25.2.3 Stream Input/Output Classes and Objects (Cont.)

- **File-Processing Templates**
  - **basic\_istream**
    - For file input
    - Inherits from **basic\_istream**
  - **basic\_ofstream**
    - For file output
    - Inherits from **basic\_ostream**
  - **basicfstream**
    - For file input and output
    - Inherits from **basic\_istream**
- **typedef specializations**
  - **istream, ofstream and fstream**





**Fig. 25.2 | Stream-I/O template hierarchy portion showing the main file-processing templates.**



## 25.3 Stream Output

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



## 25.3.1 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

### Fig25\_03. cpp

(1 of 1)

```
1 // Fig. 25. 3: Fig25_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
```

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

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

Address prints as a hexadecimal  
(base-16) number



## 25.3.2 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 );`



## 25.4 Stream Input

- **i stream input capabilities**

- **Stream extraction operator (overloaded >> operator)**
  - **Skips over white-space characters**
  - **Returns a reference to the i stream 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**



## 25.4 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



## 25.4.1 `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**



## 25.4.1 `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

### Fig25\_04. cpp

(1 of 2)

```
1 // Fig. 25. 4: Fig25_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

```

Fig25 04. cpp

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.

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



## Outline

### Fig25\_05. cpp

(1 of 2)

```
1 // Fig. 25. 5: Fig25_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**')



## Outline

Fi g25\_05. cpp

(1 of 2)

Stream extraction operation reads up to first white-space character

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



## 25.4.1 `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

Fig25\_06. cpp

(1 of 1)

```
1 // Fig. 25. 6: Fig25_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
```



## 25.4.2 `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



## 25.4.3 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”**



## 25.5 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

Fig25\_07. cpp

(1 of 1)

```

1 // Fig. 25.7: Fig25_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

```



## 25.6 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**



## 25.6.1 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

Fig25\_08. cpp

(1 of 2)

```
1 // Fig. 25. 8: Fig25_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

Fig25\_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
```

Set base to hexadecimal

Set base to octal

Reset base to decimal

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



## 25.6.2 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

### Fig25\_09. cpp

(1 of 2)

```
1 // Fig. 25. 9: Fig25_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

Fi g25\_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

```



## 25.6.3 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)



## 25.6.3 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 25.1

---

**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 25.2

---

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



## Outline

### Fig25\_10. cpp

(1 of 2)

```
1 // Fig. 25.10: Fig25_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
```



## Outline

**Fi g25\_10. cpp**

(2 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
```



## 25.6.4 Use-Defined Output Stream Manipulators

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



## Outline

### Fig25\_11. cpp

(1 of 2)

```
1 // Fig. 25. 11: Fig25_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

Fi g25\_11. cpp

(2 of 2)

```

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

```

```

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

```



## 25.7 Stream Format States and Stream Manipulators

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



Stream Manipulator	Description
<b>ski pws</b>	Skip white-space characters on an input stream. This setting is reset with stream manipulator <b>noski pws</b> .
<b>l eft</b>	Left justify output in a field. Padding characters appear to the right if necessary.
<b>ri ght</b>	Right justify output in a field. Padding characters appear to the left if necessary.
<b>i nternal</b>	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).
<b>dec</b>	Specify that integers should be treated as decimal (base 10) values.
<b>oct</b>	Specify that integers should be treated as octal (base 8) values.
<b>hex</b>	Specify that integers should be treated as hexadecimal (base 16) values.

**Fig. 25.12 | Format state stream manipulators from `<i ostream>`. (Part 1 of 2)**



Stream Manipulator	Description
<b>showbase</b>	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 <b>noshowbase</b> .
<b>showpoint</b>	Specify that floating-point numbers should be output with a decimal point. This is used normally with <b>fixed</b> 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 <b>noshowpoint</b> .
<b>uppercase</b>	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 <b>nouppercase</b> .
<b>showpos</b>	Specify that positive numbers should be preceded by a plus sign (+). This setting is reset with stream manipulator <b>noshowpos</b> .
<b>scientific</b>	Specify output of a floating-point value in scientific notation.
<b>fixed</b>	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. 25.12 | Format state stream manipulators from `<iostream>`. (Part 2 of 2)**



## 25.7.1 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

### Fig25\_13. cpp

(1 of 2)

```
1 // Fig. 25.13: 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;
```



## Outline

Fig25\_13. cpp

(2 of 2)

```
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



## 25.7.2 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

Fig25\_14. cpp

(1 of 2)

```
1 // Fig. 25.14: 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
```



## Outline

**Fig25\_14. cpp**

(2 of 2)

**Default is right justified:**

12345

**Use `std::left` to left justify x:**

12345

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

12345



## Outline

### Fig25\_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
```



## 25.7.3 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**



## Outline

### Fig25\_16. cpp

(1 of 2)

```
1 // Fig. 25.16: Fig25_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 using std::cout;
6 using std::dec;
7 using std::endl;
8 using std::hex;
9 using std::internal;
10 using std::left;
11 using std::right;
12 using std::showbase;
13
14 #include <iomanip>
15 using std::setfill;
16 using std::setw;
17
18 int main()
19 {
20     int x = 10000;
21
22     // display x
23     cout << x << " printed as int right and left justified\n"
24         << "and as hex with internal justification.\n"
25         << "Using the default pad character (space):" << endl;
26
27     // display x with base
28     cout << showbase << setw( 10 ) << x << endl;
29
```



Outline

## Fi g25\_16. cpp

(2 of 2)

```

30 // display x with left justification
31 cout << left << setw( 10 ) << x << endl;
32
33 // display x as hex with internal justification
34 cout << internal << setw( 10 ) << hex << x << endl << endl;
35
36 cout << "Using various padding characters:" << endl;
37
38 // display x using padded characters (right justification)
39 cout << right;
40 cout.fill( '*' );
41 cout << setw( 10 ) << dec << x << endl;
42
43 // display x using padded characters (left justification)
44 cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
45
46 // display x using padded characters (internal justification)
47 cout << internal << setw( 10 ) << setfill( '^' ) << hex
48     << x << endl;
49 return 0;
50 } // end main

```

10000 printed as int right and left justified  
and as hex with internal justification.

Using the default pad character (space):

10000

10000

0x 2710

Using various padding characters:

\*\*\*\*\*10000

10000%%%%

0x^^^^2710



## 25.7.4 Integral Stream Base (dec, oct, hex, showbase)

- **Integral base with stream insertion**
  - Manipulators **dec**, **hex** and **oct**
- **Integral base with stream extraction**
  - Integers prefixed with **0** (zero)
    - Treated as octal values
  - Integers prefixed with **0x** or **0X**
    - Treated as hexadecimal values
  - All other integers
    - Treated as decimal values



## 25.7.4 Integral Stream Base (dec, oct, hex, showbase) (Cont.)

- **Stream manipulator `showbase`**
  - **Forces integral values to be outputted with their bases**
    - **Decimal numbers are output by default**
    - **Leading 0 for octal numbers**
    - **Leading 0x or 0X for hexadecimal numbers**
  - **Reset the `showbase` setting with `noshowbase`**



## Outline

### Fig25\_17.cpp

(1 of 1)

```
1 // Fig. 25.17: Fig25_17.cpp
2 // Using stream manipulator showbase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::oct;
8 using std::showbase;
9
10 int main()
11 {
12     int x = 100;
13
14     // use showbase to show number base
15     cout << "Printing integers preceded by their base:" << endl
16          << showbase;
17
18     cout << x << endl; // print decimal value
19     cout << oct << x << endl; // print octal value
20     cout << hex << x << endl; // print hexadecimal value
21     return 0;
22 } // end main
```

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



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

- **Stream manipulator `scientific`**
  - Makes floating-point numbers display in scientific format
- **Stream manipulator `fixed`**
  - Makes floating-point numbers display with a specific number of digits
    - Specified by `precision` or `setprecision`
- **Without either `scientific` or `fixed`**
  - Floating-point number's value determines the output format



## Outline

### Fig25\_18. cpp

(1 of 2)

```
1 // Fig. 25. 18: Fig25_18. cpp
2 // Displaying floating-point values in system default,
3 // scientific and fixed formats.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::scientific;
9
```



## Outline

### Fig25\_18. cpp

(2 of 2)

```
10 int main()
11 {
12     double x = 0.001234567;
13     double y = 1.946e9;
14
15     // display x and y in default format
16     cout << "Displayed in default format:" << endl
17         << x << '\t' << y << endl;
18
19     // display x and y in scientific format
20     cout << "\nDisplayed in scientific format:" << endl
21         << scientific << x << '\t' << y << endl;
22
23     // display x and y in fixed format
24     cout << "\nDisplayed in fixed format:" << endl
25         << fixed << x << '\t' << y << endl;
26     return 0;
27 } // end main
```

```
Displayed in default format:
0.00123457      1.946e+009
```

```
Displayed in scientific format:
1.234567e-003   1.946000e+009
```

```
Displayed in fixed format:
0.001235      1946000000.000000
```



## 25.7.6 Uppercase/Lowercase Control (uppercase)

- **Stream manipulator uppercase**
  - Causes hexadecimal-integer values to be output with uppercase **X** and **A-F**
  - Causes scientific-notation floating-point values to be output with uppercase **E**
  - These letters output as lowercase by default
  - Reset uppercase setting with **nouppercase**



## Outline

### Fig25\_19. cpp

(1 of 1)

```
1 // Fig. 25.19: Fig25_19. cpp
2 // Stream manipulator uppercase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::showbase;
8 using std::uppercase;
9
10 int main()
11 {
12     cout << "Printing uppercase letters in scientific" << endl
13         << "notation exponents and hexadecimal values:" << endl;
14
15     // use std::uppercase to display uppercase letters; use std::hex and
16     // std::showbase to display hexadecimal value and its base
17     cout << uppercase << 4.345e10 << endl
18         << hex << showbase << 123456789 << endl;
19     return 0;
20 } // end main
```

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



## 25.7.7 Specifying Boolean Format (`bool al pha`)

- **Stream manipulator `bool al pha`**
  - Causes `bool` variables to output as "`true`" or "`false`"
    - By default, `bool` variables output as `0` or `1`
  - `nobool al pha` sets `bool` s back to displaying as `0` or `1`
  - `bool` output formats are sticky settings



# Good Programming Practice 25.1

---

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



## Outline

### Fig25\_20.cpp

(1 of 2)

```
1 // Fig. 25.20: Fig25_20.cpp
2 // Demonstrating stream manipulators bool alpha and nobool alpha.
3 #include <iostream>
4 using std::bool_alpha;
5 using std::cout;
6 using std::endl;
7 using std::nobool_alpha;
8
9 int main()
10 {
11     bool booleanValue = true;
12
13     // display default true booleanValue
14     cout << "booleanValue is " << booleanValue << endl;
15
16     // display booleanValue after using bool alpha
17     cout << "booleanValue (after using bool alpha) is "
18         << bool_alpha << booleanValue << endl << endl;
```



## Outline

### Fi g25\_20. cpp

(2 of 2)

```
19 cout << "switch booleanValue and use nobool alpha" << endl;
20 booleanValue = false; // change booleanValue
21 cout << nobool alpha << endl; // use nobool alpha
22
23
24 // display default false booleanValue after using nobool alpha
25 cout << "booleanValue is " << booleanValue << endl;
26
27 // display booleanValue after using bool alpha again
28 cout << "booleanValue (after using bool alpha) is "
29     << bool alpha << booleanValue << endl;
30 return 0;
31 } // end main
```

```
booleanValue is 1
booleanValue (after using bool alpha) is true

switch booleanValue and use nobool alpha

booleanValue is 0
booleanValue (after using bool alpha) is false
```



## 25.7.8 Setting and Resetting the Format State via Member Function `flags`

- **Member function `flags`**
  - **With no argument**
    - **Returns a value of type `fmtflags`**
      - **Represents the current format settings**
  - **With a `fmtflags` as an argument**
    - **Sets the format settings as specified**
    - **Returns the prior state settings as a `fmtflags`**
  - **Initial return value may differ across platforms**
  - **Type `fmtflags` is of class `ios_base`**



## Outline

### Fig25\_21. cpp

(1 of 2)

```
1 // Fig. 25.21: Fig25_21.cpp
2 // Demonstrating the flags member function.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios_base;
7 using std::oct;
8 using std::scientific;
9 using std::showbase;
10
11 int main()
12 {
13     int integerValue = 1000;
14     double doubleValue = 0.0947628;
15
16     // display flags value, int and double values (original format)
17     cout << "The value of the flags variable is: " << cout.flags()
18         << "\nPrint int and double in original format:\n"
19         << integerValue << '\t' << doubleValue << endl << endl;
20
21     // use cout flags function to save original format
22     ios_base::fmtflags originalFormat = cout.flags();
23     cout << showbase << oct << scientific; // change format
24
25     // display flags value, int and double values (new format)
26     cout << "The value of the flags variable is: " << cout.flags()
27         << "\nPrint int and double in a new format:\n"
28         << integerValue << '\t' << doubleValue << endl << endl;
29
30     cout.flags( originalFormat ); // restore format
```

Save the stream's  
original format state

Restore the original  
format settings



## Outline

Fig25\_21. cpp

(2 of 2)

```
31 // display flags value, int and double values (original format)
32 cout << "The restored value of the flags variable is: "
33     << cout.flags()
34     << "\nPrint values in original format again:\n"
35     << integerValue << '\t' << doubleValue << endl;
36 return 0;
37 } // 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
```



## 25.8 Stream Error States

- **eofbit**

- Set after end-of-file is encountered
- Use member function **eof** to check
  - Returns **true** if end-of-file has been encountered
  - Returns **false** otherwise

- **failbit**

- Set when a format error occurs
  - Such as a nondigit character while inputting integers
  - The characters are still left in the stream
- Use member function **fail** to check
- Usually possible to recover from such an error



## 25.8 Stream Error States (Cont.)

- **badbit**

- Set when an error that loses data occurs
- Use member function **bad** to check
- Such a serious failure is generally nonrecoverable

- **goodbit**

- Set when none of **eofbit**, **failbit** or **badbit** is set
- Use member function **good** to check

- **Member function rdstate**

- Returns the error state of the stream
  - Incorporates **eofbit**, **badbit**, **failbit** and **goodbit**
- Using the individual member functions is preferred



## 25.8 Stream Error States (Cont.)

- **Member function `clear`**
  - Sets the specified bit for the stream
  - Default argument is `goodbit`
  - **Examples**
    - `cin.clear();`
      - Clears `cin` and sets `goodbit`
    - `cin.clear( ios::failbit );`
      - Sets `failbit`



## Outline

### Fig25\_22. cpp

(1 of 2)

```
1 // Fig. 25. 22: Fig25_22. cpp
2 // Testing error states.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int integerValue;
11
12     // display results of cin functions
13     cout << "Before a bad input operation: "
14         << "\ncin.rdstate(): " << cin.rdstate()
15         << "\n    cin.eof(): " << cin.eof()
16         << "\n    cin.fail(): " << cin.fail()
17         << "\n    cin.bad(): " << cin.bad()
18         << "\n    cin.good(): " << cin.good()
19         << "\n\nExpects an integer, but enter a character: ";
20
21     cin >> integerValue; // enter character value
22     cout << endl;
23
24     // display results of cin functions after bad input
25     cout << "After a bad input operation: "
26         << "\ncin.rdstate(): " << cin.rdstate()
27         << "\n    cin.eof(): " << cin.eof()
28         << "\n    cin.fail(): " << cin.fail()
29         << "\n    cin.bad(): " << cin.bad()
30         << "\n    cin.good(): " << cin.good() << endl << endl;
```



```
31  cin.clear(); // clear stream
32
33
34  // display results of cin functions after clearing cin
35  cout << "After cin.clear() " << "\ncin.fail(): " << cin.fail()
36      << "\ncin.good(): " << cin.good() << endl;
37  return 0;
38 } // end main
```

Fi g25\_22. cpp

(2 of 2)

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
```



## 25.8 Stream Error States (Cont.)

- **basic\_ios member function operator!**
  - Returns **true** if either **badbit** or **failbit** is set
- **basic\_ios member function void \***
  - Returns **false** if either **badbit** or **failbit** is set



## 25.9 Tying an Output Stream to an Input Stream

- **`istream` member function `tie`**

- Synchronizes an **`istream`** and an **`ostream`**

- Ensures outputs appear before their subsequent inputs

- Examples

- `cin.tie( &cout );`

- Ties standard input to standard output

- C++ performs this operation automatically

- `istream.tie( 0 );`

- Unties `istream` from the `ostream` it is tied to

