Lorem ipsum dolor sit amet, consectetur adipisicing elit. Autem nihil mollitia labore harum dolore quisquam expedita. Earum enim doloremque repellat in laudantium ut cum. Maxime, consectetur! Vitae iusto molestias rem ipsum sint, vel natus perspiciatis architecto inventore, in voluptatibus repellendus vero facilis? Sed iure illum ducimus sapiente cupiditate ea deleniti asperiores quod distinctio quo iusto repellat officia atque, est odit ullam aut provident unde fugit eos? Numquam dolorem rerum a eveniet aperiam molestias debitis labore perferendis? Ipsum dicta rem, labore debitis quaerat iure perspiciatis modi voluptatem quo! Quo culpa libero harum, quia quod odio, optio amet ipsam non commodi nam accusantium ex sint ut debitis accusamus autem. Deleniti cupiditate nulla odit hic reprehenderit illum repellat placeat aperiam amet ullam, accusamus minima dolorem est tenetur ut quam tempora? Eius veniam nostrum accusamus, placeat quaerat quasi cum aperiam error quas repellendus atque. Ab aspernatur sapiente harum commodi quaerat distinctio voluptatum, natus autem explicabo perferendis tempore aut ipsa quod praesentium alias optio nam quasi reprehenderit voluptate ad quo sint tenetur soluta. Repellat at ad magni officia labore porro quas, officiis voluptatum corrupti ab numquam? Impedit dolor distinctio, in aut odit nisi consequatur facere autem, iure, quas non possimus tempora magni veritatis nulla exercitationem.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Cum eius numquam fugiat. Numquam, veritatis illum! Nulla vitae, optio reprehenderit tenetur similique iste modi consectetur libero debitis incidunt quisquam voluptates sed earum ipsum dolorem facere. Repudiandae recusandae itaque suscipit, illo aspernatur laudantium dolor libero similique officiis error corrupti, et, aliquam quia nesciunt eveniet. Culpa possimus, doloremque debitis molestiae assumenda optio quaerat? Fuga, illo! Reprehenderit rem repellendus harum natus soluta sed eveniet aliquid modi libero quibusdam nulla quidem blanditiis, suscipit omnis, qui nihil enim! Possimus qui accusamus molestiae accusantium veritatis eveniet doloribus voluptate perspiciatis. Doloribus inventore accusamus minima nam perferendis perspiciatis quia?

Capitolo 11- C++ Stream Input/Output

Outline

11.1 Introduction

- Many C++ I/O features are object-oriented
  - use references, function overloading and operator overloading
- C++ uses type safe I/O
  - Each I/O operation is automatically performed in a manner sensitive to the data type
- Extensibility
  - Users may specify I/O of user-defined types as well as standard types

11.2 Streams

- Stream
  - A transfer of information in the form of a sequence of bytes

- I/O Operations:
  - Input: A stream that flows from an input device (i.e.: keyboard, disk drive, network connection) to main memory
  - Output: A stream that flows from main memory to an output device (i.e.: screen, printer, disk drive, network connection)

11.2 Streams (II)

- I/O operations are a bottleneck
  - The time for a stream to flow is many times larger than the time it takes the CPU to process the data in the stream
- Low-level I/O
  - unformatted
  - individual byte unit of interest
  - high speed, high volume, but inconvenient for people
- High-level I/O
  - formatted
  - bytes grouped into meaningful units: integers, characters, etc.
  - good for all I/O except high-volume file processing

11.2.1 Iostream Library Header Files

- **`iostream`** library:
  - **`<iostream.h>`**: Contains **`cin`**, **`cout`**, **`cerr`**, and **`clog`** objects
  - **`<iomanip.h>`**: Contains *parameterized stream manipulators*
  - **`<fstream.h>`**: Contains information important to user-controlled file processing operations

11.2.2 Stream Input/Output Classes and Objects

- **`ios`:**
  - **`istream`** and **`ostream`** inherit from **`ios`**
    - **`iostream`** inherits from **`istream`** and **`ostream`**.
- **`<<`** (left-shift operator): overloaded as *stream insertion operator*
- **`>>`** (right-shift operator): overloaded as *stream extraction operator*
- Used with **`cin`**, **`cout`**, **`cerr`**, **`clog`**, and with user-defined stream objects

11.2.2 Stream Input/Output Classes and Objects (II)

- **`istream:`** input streams

  `cin >> someVariable`;

  - **`cin`** knows what type of data is to be assigned to **`someVariable`** (based on the type of **`someVariable`**).

- **`ostream:`** output streams
  - **`cout << someVariable;`**
    - **`cout`** knows the type of data to output
  - **`cerr << someString;`**
    - Unbuffered. Prints **`someString`** immediately.
  - **`clog << someString;`**
    - Buffered. Prints **`someString`** as soon as output buffer is full or flushed.

11.3 Stream Output

- **`ostream`**: performs formatted and unformatted output
  - Uses **`put`** for characters and **`write`** for unformatted characters
  - Output of numbers in decimal, octal and hexadecimal
  - Varying precision for floating points
  - Formatted text outputs

11.3.1 Stream-Insertion Operator

- **`<<`** is overloaded to output built-in types
  - can also be used to output user-defined types.
  - **`cout << '\n';`**
    - prints newline character
  - **`cout << endl;`**
    - **`endl`** is a stream manipulator that issues a newline character and flushes the output buffer
  - **`cout << flush;`**
    - **`flush`** flushes the output buffer.

11.3.2 Cascading Stream-Insertion/Extraction Operators

- **`<<`** : Associates from left to right, and returns a reference to its left-operand object (i.e. **`cout`**).
  - This enables cascading

```
cout << "How" << " are" << " you?";
```

Make sure to use parenthesis:

```
cout << "1 + 2 = " << (1 + 2);
  NOT
cout << "1 + 2 = " << 1 + 2;
```

11.3.3 Output of `char *` Variables

- **`<<`** will output a variable of type **`char *`** as a string

- To output the address of the first character of that string, cast the variable as type **`void *`**

```cpp
// Fig. 11.8: fig11_08.cpp
// Printing the address stored in a char* variable
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   char *string = "test";

   cout << "Value of string is: " << string
        << "\nValue of static_cast< void * >( string ) is: "
        << static_cast< void * >( string ) << endl;
   return 0;
}
```

Outline

1. Initialize string

2. Print string

2.1 cast into `void *`

2.2 Print value of pointer (address of string)

Program Output

```
Value of string is: test
Value of static_cast< void *>( string ) is: 0046C070
```

11.3.4 Character Output with Member Function put; Cascading puts

- **`put`** member function
  - outputs one character to specified stream

    `cout.put( 'A');`
  - returns a reference to the object that called it, so may be cascaded

    `cout.put( 'A' ).put( '\n' );`
  - may be called with an ASCII-valued expression

    `cout.put(65);`

    outputs **`A`**

11.4 Stream Input

- **`>>`** (stream-extraction)
  - used to perform stream input
  - Normally ignores whitespaces (spaces, tabs, newlines)
  - Returns zero (**`false`**) when **`EOF`** is encountered, otherwise returns reference to the object from which it was invoked (i.e. **`cin`**)
    - This enables cascaded input.

    ```
    cin >> x >> y;
    ```

- **`>>`** controls the state bits of the stream
  - **`failbit`** set if wrong type of data input
  - **`badbit`** set if the operation fails

11.4.1 Stream-Extraction Operator

- `>>` and `<<` have relatively high precedence
  - conditional and arithmetic expressions must be contained in parentheses

- Popular way to perform loops

```
while (cin >> grade)
```

  - extraction returns `0` (`false`) when `EOF` encountered, and loop ends

```cpp
// Fig. 11.11: fig11_11.cpp
// Stream-extraction operator returning false on end-of-file.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int grade, highestGrade = -1;

   cout << "Enter grade (enter end-of-file to end): ";
   while ( cin >> grade ) {
      if ( grade > highestGrade )
         highestGrade = grade;

      cout << "Enter grade (enter end-of-file to end): ";
   }

   cout << "\n\nHighest grade is: " << highestGrade << endl;
   return 0;
}
```

Outline

1. Initialize variables

2. Perform loop

3. Output

Program Output

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z
Highest grade is: 99
```

11.4.2 `get` and `getline` Member Functions

- **`cin.get()`**: inputs a character from stream (even white spaces) and returns it

- **`cin.get( c )`**: inputs a character from stream and stores it in **`c`**

11.4.2 `get` and `getline` Member Functions (II)

- **`cin.get(array, size):`**
  - accepts 3 arguments: array of characters, the size limit, and a delimiter (default of **`'\n'`**).
  - Uses the array as a buffer
  - When the delimiter is encountered, it remains in the input stream
  - Null character is inserted in the array
  - unless delimiter flushed from stream, it will stay there

- **`cin.getline(array, size)`**
  - operates like **`cin.get(buffer, size)`** but it discards the delimiter from the stream and does not store it in array
  - Null character inserted into array

```
// Fig. 11.12: fig11_12.cpp
// Using member functions get, put and eof.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   char c;

   cout << "Before input, cin.eof() is " << cin.eof()
        << "\nEnter a sentence followed by end-of-file:\n";

   while ( ( c = cin.get() ) != EOF )
      cout.put( c );

   cout << "\nEOF in this system is: " << c
   cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
   return 0;
}
```

`cin.eof()` returns **`false (0)`** or **`true (1)`**

`cin.get()` returns the next character from input stream, including whitespace.

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
EOF in this system is: -1
After input cin.eof() is 1
```

Outline

1. Initialize variables

2. Input data

2.1 Function call

3. Output

Program Output

Outline

```
// Fig. 11.14: fig11_14.cpp
// Character input with member function getline.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   const SIZE = 80;
   char buffer[ SIZE ];

   cout << "Enter a sentence:\n";
   cin.getline( buffer, SIZE );

   cout << "\nThe sentence entered is:\n" << buffer << endl;
   return 0;
}
```

1. Initialize variables

2. Input

2.1 Function call

3. Output

Program Output

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

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

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

- **`ignore`** member function
  - skips over a designated number of characters (default of one)
  - terminates upon encountering a designated delimiter (default is **`EOF`**, skips to the end of the file)
- **`putback`** member function
  - places the previous character obtained by **`get`** back in to the stream.
- **`peek`**
  - returns the next character from the stream without removing it

11.4.4 Type-Safe I/O

- **<<** and **>>** operators
  - Overloaded to accept data of different types
  - When unexpected data encountered, error flags set
  - Program stays in control

11.5 Unformatted I/O with `read`, `gcount` and `write`

- **`read`** and **`write`** member functions
  - unformatted I/O
  - input/output raw bytes to or from a character array in memory
  - Since the data is unformatted, the functions will not terminate at a **`newline`** character for example.
    - Instead, like **`getline`**, they continue to process a designated number of characters.
  - If fewer than the designated number of characters are read, then the failbit is set.
- **`gcount`**:
  - returns the total number of characters read in the last input operation.

Outline

1. Initialize objects

2. Input

3. Output

```cpp
// Fig. 11.15: fig11 15.cpp
// Unformatted I/O with read, gcount and write.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   const int SIZE = 80;
   char buffer[ SIZE ];

   cout << "Enter a sentence:\n";
   cin.read( buffer, 20 );
   cout << "\nThe sentence entered was:\n";
   cout.write( buffer, cin.gcount() );
   cout << endl;
   return 0;
}
```

Only reads first **20** characters

g.count() returns **20** because that was the number of characters read by the last input operation.

Program Output

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

11.6 Stream Manipulators

- stream manipulator capabilities:
  - setting field widths
  - setting precisions
  - setting and unsetting format flags
  - setting the fill character in fields
  - flushing streams
  - inserting a newline in the output stream and flushing the stream inserting a null character in the output stream and skipping whitespace in the input stream.

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

- **`oct`, `hex`,** or **`dec`**:
  - change base of which integers are interpreted from the stream.

  Example:

```
int n = 15;
 cout << hex << n;
```

  - prints **`"F"`**

- **`setbase`**:
  - changes base of integer output
  - load **`<iomanip>`**
  - Accepts an integer argument (**`10`**, **`8`**, or **`16`**)

    ```
    cout << setbase(16) << n;
    ```

  - parameterized stream manipulator - takes an argument

```cpp
// Fig. 11.16: fig11_16.cpp
// Using hex, oct, dec and setbase stream manipulators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::hex;
using std::dec;
using std::oct;
using std::setbase;

int main()
{
   int n;

   cout << "Enter a decimal number: ";
   cin >> n;

   cout << n << " in hexadecimal is: "
        << hex << n << '\n'
        << dec << n << " in octal is: "
        << oct << n << '\n'
        << setbase( 10 ) << n << " in decimal is: "
        << n << endl;

   return 0;
}
```

```
Enter a decimal number: 20
```

```
20 in hexadecimal is: 14
```

```
20 in octal is: 24
```

```
20 in decimal is: 20
```

Outline

1. Load header

1.1 Initialize variables

2. Input number

3. Output in hex

3.1 Output in octal

3.2 Output in decimal

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

Program Output

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

- **`precision`**
  - member function
  - sets number of digits to the right of decimal point

    `cout.precision(2);`
  - **`cout.precision()`** returns current precision setting
- **`setprecision`**
  - parameterized stream manipulator
  - Like all parameterized stream manipulators, **`<iomanip>`** required
  - specify precision:

    ```
    cout << setprecision(2) << x;
    ```
- For both methods, changes last until a different value is set

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

- **`ios width`** member function
  - sets field width (number of character positions a value should be output or number of characters that should be input)
  - returns previous width
  - if values processed are smaller than width, fill characters inserted as padding
  - values are not truncated - full number printed
  - **`cin.width(5);`**
- **`setw`** stream manipulator

```
cin >> setw(5) >> string;
```

- Remember to reserve one space for the null character

```
// fig11_18.cpp
// Demonstrating the width member function
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int w = 4;
   char string[ 10 ];

   cout << "Enter a sentence:\n";
   cin.width( 5 );

   while ( cin >> string ) {
      cout.width( w++ );
      cout << string << endl;
      cin.width( 5 );
   }

   return 0;
}
```

Outline

1. Initialize variables

2. Input sentence

2.1 Set width

2.2 Loop and change width

3. Output

```
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

Program Output

11.6.4 User-Defined Manipulators

- We can create our own stream manipulators
  - **`bell`**
  - **`ret`** (carriage return)
  - **`tab`**
  - **`endLine`**

- parameterized stream manipulators
  - consult installation manuals

11.7 Stream Format States

- Format flags
  - specify formatting to be performed during stream I/O operations
- **`setf`**, **`unsetf`** and **`flags`**
  - member functions that control the flag settings

11.7.1 Format State Flags

- Format State Flags
  - defined as an enumeration in class **`ios`**
  - can be controlled by member functions
  - **`flags -`** specifies a value representing the settings of all the flags
    - returns **`long`** value containing prior options
  - **`setf`** - one argument, "ors" flags with existing flags
  - **`unsetf -`** unsets flags
  - **`setiosflags`** - parameterized stream manipulator used to set flags
  - **`resetiosflags`** - parameterized stream manipulator, has same functions as **`unsetf`**
- Flags can be combined using bitwise or "**`|`**"

11.7.2 Trailing Zeros and Decimal Points (`ios::showpoint`)

- **`ios::showpoint`**
  - forces a float with an integer value to be printed with its decimal point and trailing zeros

```
cout.setf(ios::showpoint)
cout << 79;
```

`79` will print as **`79.00000`**

- number of zeros determined by precision settings

11.7.3 Justification (`ios::left`, `ios::right`, `ios::internal`)

- **`ios::left`**
  - fields to left-justified with padding characters to the right
- **`ios::right`**
  - default setting
  - fields right-justified with padding characters to the left
- Character used for padding set by
  - **`fill`** member function
  - **`setfill`** parameterized stream manipulator
  - default character is space

11.7.3 Justification (`ios::left`, `ios::right`, `ios::internal`) (II)

- **`internal`** flag
  - number's sign left-justified
  - number's magnitude right-justified
  - intervening spaces padded with the fill character
- **`static`** data member **`ios::adjustfield`**
  - contains **`left`**, **`right`** and **`internal`** flags
  - **`ios::adjustfield`** must be the second argument to **`setf`** when setting the **`left`**, **`right`** or **`internal`** justification flags.

```
cout.setf( ios::left, ios::adjustfield);
```

```cpp
// Fig. 11.22: fig11_22.cpp
// Left-justification and right-justification.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setw;
using std::setiosflags;
using std::resetiosflags;

int main()
{
   int x = 12345;

   cout << "Default is right justified:\n"
        << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
        << "\nUse setf to set ios::left:\n" << setw(10);

   cout.setf( ios::left, ios::adjustfield );
   cout << x << "\nUse unsetf to restore default:\n";
   cout.unsetf( ios::left );
   cout << setw( 10 ) << x
        << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
        << "\nUse setiosflags to set ios::left:\n"
        << setw( 10 ) << setiosflags( ios::left ) << x
        << "\nUse resetiosflags to restore default:\n"
        << setw( 10 ) << resetiosflags( ios::left )
        << x << endl;
   return 0;
}
```

Outline

1. Initialize variable

2. Use parameterized stream manipulators

3. Output

```
Default is right justified:
     12345
```

```
USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
```

```
USING PARAMETERIZED STREAM MANIPULATORS
Use setiosflags to set ios::left:
12345
```

```
Use resetiosflags to restore default:
     12345
```

Outline

Program Output

```
Default is right justified:
     12345

USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
Use unsetf to restore default:
     12345

USING PARAMETERIZED STREAM MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
     12345
```

11.7.4 Padding(`fill, setfill`)

- **`fill`** member function
  - specifies the fill character
  - space is default
  - returns the prior padding character

```
cout.fill( '*');
```

- **`setfill`** manipulator
  - also sets fill character

```
cout << setfill ('*');
```

```cpp
// Fig. 11.24: fig11_24.cpp
// Using the fill member function and the setfill
// manipulator to change the padding character for
// fields larger than the values being printed.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setw;
using std::hex;
using std::dec;
using std::setfill;

int main()
{
   int x = 10000;
```

Outline

1. Load header

1.1 Initialize variable

Outline

2. Set fill character

3. Output

```

   cout << x << " printed as int right and left justified\n"
        << "and as hex with internal justification.\n"
        << "Using the default pad character (space):\n";
   cout.setf( ios::showbase );
   cout << setw( 10 ) << x << '\n';
   cout.setf( ios::left, ios::adjustfield );
   cout << setw( 10 ) << x << '\n';
   cout.setf( ios::internal, ios::adjustfield );
   cout << setw( 10 ) << hex << x;

   cout << "\n\nUsing various padding characters:\n";
   cout.setf( ios::right, ios::adjustfield );
   cout.fill( '*' );
   cout << setw( 10 ) << dec << x << '\n';
   cout.setf( ios::left, ios::adjustfield );
   cout << setw( 10 ) << setfill( '%' ) << x << '\n';
   cout.setf( ios::internal, ios::adjustfield );
   cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
   return 0;
}
```

Program Output

```
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
```

11.7.5- Integral Stream Base (ios::dec, ios::oct, ios::hex, ios::showbase)

- **`ios::basefield`** static member
  - used similarly to **`ios::adjustfield`** with **`setf`**
  - includes the **`ios::oct`**, **`ios::hex`** and **`ios::dec`** flag bits
  - specify that integers are to be treated as octal, hexadecimal and decimal values
  - default is decimal
  - default for stream extractions depends on form inputted
    - integers starting with **`0`** are treated as octal
    - integers starting with **`0x`** or **`0X`** are treated as hexadecimal
  - once a base specified, settings stay until changed

11.7.6 Floating-Point Numbers; Scientific Notation (`ios::scientific,ios::fixed`)

- **`ios::scientific`**
  - forces output of a floating point number in scientific notation:
    - **`1.946000e+009`**

- **`ios::fixed`**
  - forces floating point numbers to display a specific number of digits to the right of the decimal (specified with **`precision`**)

11.7.6 Floating-Point Numbers; Scientific Notation (II)

- **`static`** data member **`ios::floatfield`**
  - contains **`ios::scientific`** and **`ios::fixed`**
  - used similarly to **`ios::adjustfield`** and **`ios::basefield`** in **`setf`**
    - **`cout.setf(ios::scientific, ios::floatfield);`**
  - **`cout.setf(0, ios::floatfield)`** restores default format for outputting floating-point numbers

```cpp
// Fig. 11.26: fig11_26.cpp
// Displaying floating-point values in system default,
// scientific, and fixed formats.
#include <iostream>

using std::cout;
using std::endl;
using std::ios;

int main()
{
   double x = .001234567, y = 1.946e9;

   cout << "Displayed in default format:\n"
        << x << '\t' << y << '\n';
   cout.setf( ios::scientific, ios::floatfield );
   cout << "Displayed in scientific format:\n"
        << x << '\t' << y << '\n';
   cout.unsetf( ios::scientific );
   cout << "Displayed in default format after unsetf:\n"
        << x << '\t' << y << '\n';
   cout.setf( ios::fixed, ios::floatfield );
   cout << "Displayed in fixed format:\n"
        << x << '\t' << y << endl;
   return 0;
}
```

Outline

1. Initialize variables

2. Set flags

3. Output

```
Displayed in default format:
0.00123457     1.946e+009
Displayed in scientific format:
1.234567e-003  1.946000e+009
Displayed in default format after unsetf:
0.00123457     1.946e+009
Displayed in fixed format:
0.001235       1946000000.000000
```

Program Output

11.7.7 Uppercase/Lowercase Control (`ios::uppercase`)

- **`ios::uppercase`**
  - forces uppercase **`E`** to be output with scientific notation

    `4.32E+010`
  - forces uppercase **`X`** to be output with hexadecimal numbers, and causes all letters to be uppercase

    `75BDE`

11.7.8 Setting and Resetting the Format Flags (`flags, setiosflags, resetiosflags`)

- **`flags`** member function
  - without argument, returns the current settings of the format flags (as a **`long`** value)
  - with a **`long`** argument, sets the format flags as specified
    - returns prior settings
- **`setf`** member function
  - sets the format flags provided in its argument
  - returns the previous flag settings as a **`long`** value

```
long previousFlagSettings =
   cout.setf( ios::showpoint | ios::showpos );
```

11.7.8 Setting and Resetting the Format Flags (`flags`, `setiosflags`, `resetiosflags`) (II)

- **`setf`** with two **`long`** arguments

  ```
  cout.setf( ios::left, ios::adjustfield );
  ```

  clears the bits of **`ios::adjustfield`** then sets **`ios::left`**

  - This version of setf can be used with
  - **`ios::basefield (ios::dec, ios::oct, ios::hex)`**
  - **`ios::floatfield (ios::scientific, ios::fixed)`**

  -
    ```
    ios::adjustfield (ios::left, ios::right,
                      ios::internal )
    ```
- **`unsetf`**
  - resets specified flags
  - returns previous settings

```
// Fig. 11.28: fig11_28.cpp
// Demonstrating the flags member function.
#include <iostream>

using std::cout;
using std::endl;
using std::ios;


int main()
{
   int i = 1000;
   double d = 0.0947628;

   cout << "The value of the flags variable is: "
        << cout.flags()
        << "\nPrint int and double in original format:\n"
        << i << '\t' << d << "\n\n";
   long originalFormat =
           cout.flags( ios::oct | ios::scient
   cout << "The value of the flags variable i
        << cout.flags()
        << "\nPrint int and double in a new format\n"
        << "specified using the flags member function:\n"
        << i << '\t' << d << "\n\n";
   cout.flags( originalFormat );
   cout << "The value of the flags variable is: "
        << cout.flags()
        << "\nPrint values in original format again:\n"
        << i << '\t' << d << endl;
   return 0;
}
```

Outline

1. Initialize variables

2. Set flags

3. Output

```
The value of the flags variable is: 0
```

```
Print int and double in original format:
```

```
Print int and double in a new format
specified using the flags member function:
1750    9.476280e-002
```

Notice how **`originalFormat`** (a **`long`**) is t

```
The value of the flags variable is: 0
```

```
Print values in original format again:
1000    0.0947628
```

```
The value of the flags variable is: 0
Print int and double in original format:
1000    0.0947628

The value of the flags variable is: 4040
Print int and double in a new format
specified using the flags member function:
1750    9.476280e-002

The value of the flags variable is: 0
Print values in original format again:
1000    0.0947628
```

Program Output

11.8 Stream Error States

- **`eofbit`**
  - set for an input stream after end-of-file encountered
  - **`cin.eof()`** returns **`true`** if end-of-file has been encountered on **`cin`**

- **`failbit`**
  - set for a stream when a format error occurs
  - **`cin.fail()`** - returns **`true`** if a stream operation has failed
  - normally possible to recover from these errors

11.8 Stream Error States (II)

- **`badbit`**
  - set when an error occurs that results in data loss
  - **`cin.bad()`** returns **`true`** if stream operation failed
  - normally nonrecoverable
- **`goodbit`**
  - set for a stream if neither **`eofbit`**, **`failbit`** or **`badbit`** are set
  - **`cin.good()`** returns **`true`** if the **`bad`**, **`fail`** and **`eof`** functions would all return false.
  - I/O operations should only be performed on "good" streams
- **`rdstate`**
  - returns the state of the stream
  - stream can be tested with a **`switch`** statement that examines all of the state bits
  - easier to use **`eof`**, **`bad`**, **`fail`**, and **`good`** to determine state

11.8 Stream Error States (III)

- **`clear`**
  - used to restore a stream's state to "good"
  - **`cin.clear()`** clears **`cin`** and sets **`goodbit`** for the stream.
  - **`cin.clear( ios::failbit )`** actually sets the **`failbit`**.
    - might do this when encountering a problem with a user-defined type
- Other operators
  - **`operator!`**
    - returns **`true`** if **`badbit`** or **`failbit`** set
  - **`operator void*`**
    - returns **`false`** if **`badbit`** or **`failbit`** set
  - useful for file processing

Outline

1. Initialize variable

2. Function calls

```cpp
// Fig. 11.29: fig11_29.cpp
// Testing error states.
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main()
{
   int x;
   cout << "Before a bad input operation:"
        << "\ncin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good()
        << "\n\nExpects an integer, but enter a character: ";
   cin >> x;

   cout << "\nAfter a bad input operation:"
        << "\ncin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good() << "\n\n";

   cin.clear();

   cout << "After cin.clear()"
        << "\ncin.fail(): " << cin.fail()
        << "\ncin.good(): " << cin.good() << endl;
   return 0;
}
```

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

```
Expects an in
```

```
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
```

```
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
```

Program Output

11.9 Tying an Output Stream to an Input Stream

- **`tie`** member function
  - synchronize operation of an **`istream`** and an **`ostream`**
  - outputs appear before subsequent inputs
  - automatically done for **`cin`** and **`cout`**
- **`inputStream.tie( &outputStream );`**
  - ties **`inputStream`** to **`outputStream`**
  - **`cin.tie( &cout)`** done automatically
- **`inputStream.tie( 0 );`**
  - unties **`inputStream`** from an output stream