Object-Oriented Programming
(in C++)

Stream Input/Output

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

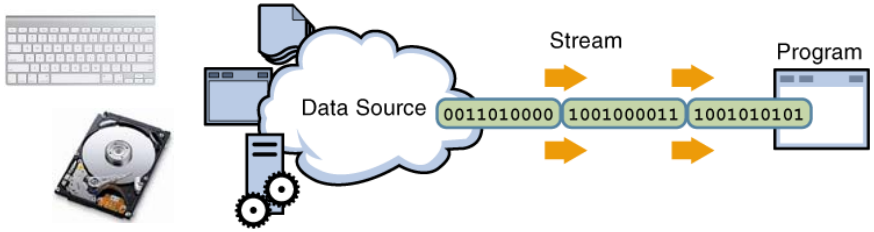
Stream I/O

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

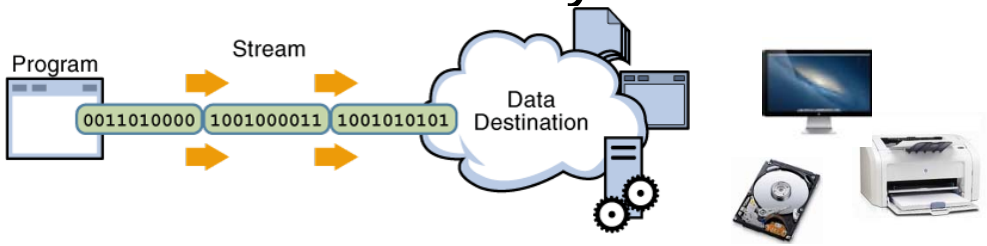
Use the C++-style, type-safe I/O exclusively in C++ programs, even though C-style I/O is available to C++ programmers

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

Streams

- "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
- "Low-level", unformatted I/O
  - Individual bytes are the items of interest
  - High-speed, high-volume
  - Not particularly convenient for programmers

Classic Streams vs. Standard Streams

- In the past, the C++ classic stream libraries enabled input and output of `chars`
  - A `char` normally occupies one byte
    - Represents only a limited set of characters (such as those in the ASCII character set)
- C++ includes the standard stream libraries, which enable I/O operations with Unicode characters (www.unicode.org)
  - C++ includes an additional character type called **`wchar_t`**, which can store 2-byte Unicode characters

iostream Library Header Files

- <iostream> header file
  - Declares basic services required for all stream-I/O operations
  - Defines cin, cout, cerr, and clog
  - Provides both unformatted- and formatted-I/O services
- <iomanip> header file
  - Declares services for performing formatted I/O with parameterized stream manipulators, such as setw and setprecision
- <fstream> header file
  - Declares services for user-controlled file processing
  - Will be discussed in Chapter 17 (file processing)

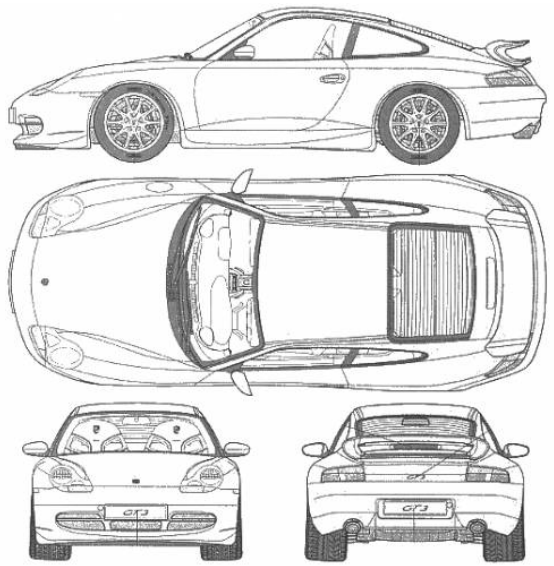
Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

Classes vs Objects: A Preview

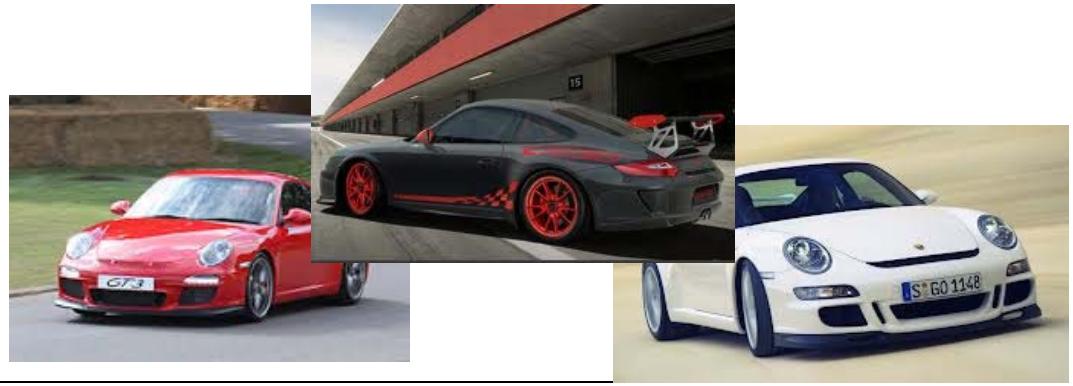
- A class is like a car blueprint

```
class car {
   string engine;
   string wheels;
   string color;
   ...
   void accelerate() { ... }
   void brake() { ... }
   ...
};
```



- An object is an instantiated class (like a car)

```
int main {
   car car1;
   car car2;
   car car3;
   ...
}
```



Stream I/O Classes

- Includes many class templates
  - `basic_istream`
    - Supports stream-input operations
  - `basic_ostream`
    - Supports stream-output operations
  - `basic_iostream`
    - Supports stream-input/-output operations
- Provides a set of `typedefs` that provide aliases for these template specializations

```
namespace std {
  template<class E, class T = char_traits<E> >
  class basic_istream { ... };
  typedef basic_istream<char, char_traits<char> > istream;
}
```

Stream I/O Objects

- There are some predefined instances/objects
  - `istream` object
    - `cin:` connected to the standard input device, usually the keyboard
  - `ostream` objects
    - `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

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

Stream Output

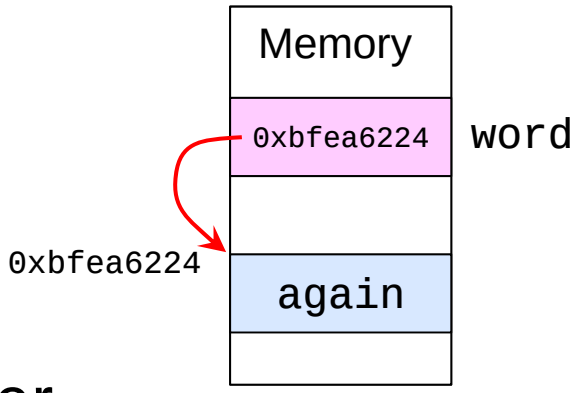
- `ostream` output capabilities
  - Can output
    - Standard data types (with << operator)
    - Characters
    - Unformatted data
    - Integers
    - Floating-point values
    - Values in fields

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

```
char *word = “again”;
cout << word << endl;
```

```
again
```



  - Solution
    - Cast the **char *** to a **void ***
    - **void *** : a special type of pointer
      - void pointers are pointers that point to a value that has no type

```
char *word = “again”;
cout << (void *)word << endl;
cout << static_cast<void *>(word) << endl;
```

```
0xbfea6224
0xbfea6224
```

C-style casting

C++-style casting

ostream::put(char)

- `ostream` member function **`put`**

```
ostream& put(char c);
```

  - Returns a reference to the same `ostream` object
    - So the `ostream` object outputs a **character**
    - 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 );
```

```
AA
A
```

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

Stream Input

- `istream` input capabilities
  - Stream extraction operator (overloaded >> operator)
    - Skips over **white-space characters**
    - Returns a reference to the `istream` object
    - Example:

```
string x1, x2, x3;
cout << "Enter your text: ";
cin >> x1 >> x2 >> x3;
```

```
Enter your text: Object-Oriented Programming Language
```

x1 x2 x3

Stream Input (Cont'd)

- When the reference returned by **>>** is 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
- Example:

```
int
char character;
cout >> "Enter your text: ";
while ( cin >> character) {
  cout << "The character is" << character << endl;
}
```

```
Enter your text: abc d e
The character is a
The character is b
The character is c
The character is d
The character is e
(waiting for input)
```

```
Enter your text: 12 3 a
The character is 12
The character is 3
(program terminated)
```

istream::get( )

- With no arguments `int get();`
  - Returns one character input from the stream
    - Any character, including **white-space and non-graphic characters, EOF**
  - Returns **EOF** when end-of-file is encountered / input fails
    - EOF is a constant (type **`int`**) representing End-of-File has been reached in a reading operation
  - **`cin.eof()`** checks whether **EOF** has occurred on **`cin`**
- Example:

```
char a = (char)cin.get();
int ascii = cin.get();
```

eof, get, and put: An Example

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

```
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
```

<Ctrl>-z on Windows, <Ctrl>-d on UNIX and Mac

istream::get(char&)

- With a character-reference argument

```
istream& get(char& c);
```

  - Stores input character (*not including EOF*) in the character-reference argument
  - Returns a reference to the `istream` object, which could be used as a condition
  - Leaves **c** unchanged if input fails

- Example

```
char ch1;
cin.get(ch1);
```

istream::get(char*,int,char)

- With three arguments:

```
istream& get(char* s, int n, char delim ='\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
- Example:

```
char buffer[80];
cin.get(buffer, 80);
cin.get(buffer, 80, ',');
```

istream::getline(char*,int,char)

- With three arguments:

```
istream& getline(char* s, int n, char delim ='\n');
```

  - Operates similarly to istream::get(char*,int,char)
  - *Delimiter is removed from the stream, but not placed in array*
- Example

```
char buffer[80];
cin.getline(buffer, 80);
cin.getline(buffer, 80, ',');
```

getline vs get

- getline

```
char buffer1[80];
char buffer2[80];
cin.getline(buffer1, 80, ',');
cin.getline(buffer2, 80, ',');
cout << “buffer1=“ << buffer1 << endl;
cout << “buffer2=“ << buffer2 << endl;
```

cin baz

```
foo,bar,baz
buffer1=foo
buffer2=bar
```

- get

```
char buffer1[80];
char buffer2[80];
cin.get(buffer1, 80, ',');
cin.get(buffer2, 80, ',');
cout << “buffer1=“ << buffer1 << endl;
cout << “buffer2=“ << buffer2 << endl;
```

cin ,bar,baz

```
foo,bar,baz
buffer1=foo
buffer2=
```

istream::ignore

```
istream& ignore( int n = 1, int delim = EOF );
```

- Reads and discards a designated number of characters or terminates upon encountering a designated delimiter
  - n: maximum number of characters to extract (and ignore)
  - delim: the function stops extracting characters as soon as an extracted character compares equal to this delimiting character

istream::ignore: An Example

```
#include <iostream>
using namespace std;

int main() {
  char first, last;

  cout << "Please enter your name: ";

  first = cin.get();      // get one character
  cin.ignore(256,' ');    // ignore until space

  last = cin.get();       // get one character

  cout << "Your initials are " << first << last << endl;

  return 0;
}
```

```
Please enter your name: John Smith
Your initials are JS
```

istream::putback & istream::peek

- istream member function putback

```
istream& putback( char c );
```

  - Places previous character obtained by a get from the input stream back into the stream

- istream member function peek

```
int peek( );
```

  - Returns the next character in the input stream, but does not remove it from the stream

putback vs peek: An Example

```
#include <iostream>
using namespace std;

int main() {
  cout << "Please enter a number or a word: ";
  char c = cin.get();  // cin.peek();
  if ( (c >= '0') && (c <= '9') ) {
    int n;
    cin.putback(c);  // skipped if cin.peek() is called
    cin >> n;
    cout << "You entered a number: " << n << endl;
  } else {
    char str[80];
    cin.putback(c);  // skipped if cin.peek() is called
    cin >> str;
    cout << "You entered a word: " << str << endl;
  }
  return 0;
}
```

```
Please, enter a number or a word: foobar
You entered the word: foobar
```

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
  - The program is able to “stay in control”

```
#include <iostream>

class myClass {...// ‘<<‘ must be overloaded }

int main() {
  myClass var;
  cout << var;
  return 0;
}
```

“Operator overloading” will be discussed later in Chapter 11

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

Unformatted Input: istream::read

- `istream` member function `read`

```
istream& read( char* s, int n );
```

  - Reads a block of data of *n* bytes and stores it in the array pointed by *s*.
  - If the End-of-File is reached before *n* bytes have been read, the array will contain all the elements read until it

- Example:

```
char buffer[80];
cin.read(buffer,20);
```

read 20 bytes from the input stream to buffer

istream::gcount

- `istream` member function `gcount`

```
int gcount ( ) const;
```

  - Reports number of bytes read by **last** input operation

- Examples

```
char buffer[80];
cin.read(buffer,20);
cout << cin.gcount() << endl;
```

```
Entering a sentences.....
20
```

Unformatted Output: ostream::write

- `ostream` member function `write`

```
ostream& write ( const char* s, int n );
```

  - Outputs *n* bytes from a character array

```
const int SIZE = 80;
char buffer[ SIZE ]; // create array of 80 characters

// use function read to input characters into buffer
cout << "Enter a sentence:" << endl;
cin.read( buffer, 20 );

// use functions write and gcount to display buffer characters
cout << endl << "The sentence entered was:" << endl;
cout.write( buffer, cin.gcount() );
cout << endl;
```

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

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

Stream Manipulators

- C++ provides various stream manipulators that perform formatting tasks
  - Setting base (`dec`, `oct`, `hex`, `setbase`)
  - Setting precision (`setprecision`)
  - Setting field widths (`setw`)
  - Setting justifications (`left`, `right`, `internal`)
  - Setting and unsetting format state (`setioflags`, `resetioflags`)
  - Setting the fill character in fields (`setfill`)
  - Flushing streams (`unitbuf`)
  - Skipping white space in the input stream (`skipws`)
  - ...

Integral Stream Base: dec, oct, and hex

- 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 (base 10)
  - setbase manipulator: Takes one integer of 8, 10, or 16 to set the base to octal, decimal, or hexadecimal
    - Stream base values are "sticky"
    - Remain until explicitly changed to another base value
  - showbase manipulator: 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

Stream Manipulators for Integral Base: An Example

```
#include <iostream>
#include <iomanip> // for setbase manipulator
using namespace std;

int main() {
  int number;

  cout << "Please enter a decimal number: ";
  cin >> number;

  cout << number << " in hexadecimal is: " << hex
    << number << endl;
  cout << number << " in hexadecimal is: ”
    << showbase << number << endl;
  cout << number << " in hexadecimal is: " << setbase(8)
    << number << endl;

  return 0;
}
```

A "sticky" manipulator remains in effect for all subsequent fields, even in subsequent `cout` lines. A non-sticky manipulator affects only the next field.

```
Please enter a decimal number: 10
10 in hexadecimal is: a
a in hexadecimal is: 0xa
0xa in hexadecimal is: 012
```

Floating-Point Precision: `setprecision`

- Sets the precision of floating-point numbers
  - Number of digits displayed to the right of the decimal point
  - **`setprecision`** parameterized stream **manipulator (<iomanip>)**

```
float x = 3.14159;

cout << x << endl;

cout << setprecision(3) << scientific << x << endl;
cout << setprecision(3) << fixed << x << endl;

cout.precision(4); // use member function to set the precision
cout << x << endl;
```

Fixed notation

Scientific notation

```
3.14159
3.142e+00
3.142
3.1416
```

Field Width: `setw`

- Sets the field width
- Used for ostream
  - Sets the number of character positions in which value is output
    - Fill characters are inserted as padding
    - Values wider than the field are not truncated
- Used for istream
  - Sets the maximum number of characters that should be input
    - For **`char`** array, maximum of **one fewer** characters than the width will be read (to accommodate null character)
- Field width settings are **not sticky**

setw: An Example

```
#include <iostream>
#include <iomanip> // for setw manipulator
using namespace std;

int main() {
  char buffer[80];

  cin >> setw(5) >> buffer;
  cout << buffer << endl;
  cout << setw(3) << buffer << endl;
  cout << setw(10) << buffer << endl;

  cin.width(5);   // use member function to set the width
  cout.width(10); // use member function to set the width
  cin >> buffer;
  cout << buffer << endl;

  return 0;
}
```

```
abcdefghijklmn
abcd
abcd
      abcd
      efgh
```

User-Defined Output Stream Manipulators

- C++ allows users to define their own stream manipulators

```
#include <iostream>
using namespace std;

ostream& tab( ostream& output) {
  return output << '\t';
}

int main() {
  cout << "User-defined" << tab << "manipulator" << endl;
  return 0;
}
```

```
User-defined        manipulator
```

Trailing Zeros and Decimal Points: showpoint

- Forces a floating-point number to be output with its decimal point and trailing zeros

```
#include <iostream>
using namespace std;

int main()
{
   // display double values with default stream format
   cout << "Before using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl << endl;

   // display double value after showpoint
   cout << showpoint
      << "After using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl;
} // 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
```

Justification: `left` and `right`

- Justification in a field (**`<iostream>`**)

  `sticky setting`

  - Manipulator `left`
    - Fields are left-justified
    - Padding characters to the right
  - Manipulator `right` (by default)
    - Fields are right-justified
    - Padding characters to the left

```
int x = 12345;
cout << setw(10) << x << endl;
cout << left << setw(10) << x << endl;
cout << right << setw(10) << x << endl;
```

```
     12345
12345
     12345
```

Justification: `internal`

- Indicates that
  - A number's sign (or base) should be left justified within a field
  - The number's magnitude should be right justified
  - Intervening spaces should be padded with the fill character

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

```
+       123
```

Padding: `setfill`

- Padding in a field (`<iomanip>`)
  - Specifies the fill character
  - Fill characters are used to pad a field
  - `sticky` setting

```
int x = 10000;
// display x as hex with internal justification
cout << internal << setw( 10 ) << hex << x << endl << endl;

cout << "Using various padding characters:" << endl;

// display x using padded characters (right justification)
cout << right;
cout.fill( '*' ); // use member function to set the fill character
cout << setw( 10 ) << dec << x << endl;

// display x using padded characters (left justification)
cout << left << setw( 10 ) << setfill( '%' ) << x << endl;

// display x using padded characters (internal justification)
cout << internal << setw( 10 ) << setfill( '^' ) << hex
   << x << endl;
```

```
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

ostream::flags Member Function (1/2)

- flags() returns the current format settings as a fmtflags data type (representing the format state/flag)
- flags(fmtflags) sets the format flags and returns the prior state settings

```
#include <iostream>
using namespace std;

int main() {
  int i = 10;

  ios_base::fmtflags originalFormat = cout.flags();
  cout << showbase << oct << "i=" << i << endl;
  cout << "i=" << i << endl;

  cout.flags(originalFormat);
  cout << "i=" << i << endl;

  return 0;
}
```

```
i=012
i=012
i=10
```

`ostream::flags` Member Function (2/2)

- Sets the given flags and clears any other flags already set

```
#include <iostream>
using namespace std;

int main() {
  cout.flags(ios::showpos | ios::showpoint);
  cout << 4.0 << endl;
  cout << fixed << 4.0 << endl;
  return 0;
}
```

```
+4.00000
+4.000000
```

- The default precision of a floating-point number is 6
  - When neither `fixed` nor `scientific` is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display)

ostream::setf Member Function

- Sets the given flags and does not clear any other flags already set

```
#include <iostream>
using namespace std;

int main() {
  cout.setf(ios::showpos | ios::showpoint);
  cout << 4.0 << endl;
  cout.unsetf(ios::showpos);
  cout << fixed << 4.0 << endl;
  return 0;
}
```

```
+4.00000
4.000000
```

Stream Manipulator	Sticky	Header File	Flag Name (fmtflags type)	Default	Member Function	Description
dec	○	iostream	ios::dec	Set		Use decimal base
hex	○	iostream	ios::hex	Not set		Use hexadecimal base
oct	○	iostream	ios::oct	Not set		Use octal base
showbase	○	iostream	ios::showbase	Not set		Show numerical base prefixes
fixed	○	iostream	ios::fixed	Not set		Use fixed floating-point notation
scientific	○	iostream	ios::scientific	Not set		Use scientific floating-point notation
showpoint	○	iostream	ios::showpoint	Not set		Show decimal point
left	○	iostream	ios::left	Not set		Adjust output to the left
right	○	iostream	ios::right	Set		Adjust output to the right
internal	○	iostream	ios::internal	Not set		Pad after sign or base character
showpos	○	iostream	ios::showpos	Not set		Show positive sign
uppercase	○	iostream	ios::uppercase	Not set		Uppercase A-F for hex
setiosflags(f)	○	iomanip			setf(f)	Set format flags
resetiosflags(f)	○	iomanip			unsetf(f)	Reset format flags
setbase(n)	○	iomanip				Set basefield flag
setw(n)	×	iomanip			width(n)	Set field width
setprecision(n)	○	iomanip			precision(n)	Set decimal precision
setfill(c)	○	iomanip			fill(c)	Set fill character

Outline

- Introduction to Stream I/O
  - Streams
  - Stream I/O Classes and Objects
- Formatted I/O
  - Stream Output
  - Stream Input
- Unformatted I/O
  - Stream Input
  - Stream Output
- More for Formatted I/O
  - Stream Manipulators
  - Stream Format States
  - Stream Error States

Stream Error State (`iostate` Type)

- The state of an I/O stream is stored in the following bits:

`iostate` flag	Indication	Member functions to check state (return `true` if ...)			
		`good()`	`eof()`	`fail()`	`bad()`
`goodbit`	Set if no errors	`true`			
`eofbit`	Set after End-of-File reached on input operation		`true`		
`failbit`	Set when a logical error on I/O operation			`true`	
`badbit`	Set when a read/write error on I/O operation			`true`	`true`

- `rdstate()` member function
  - Returns the error state
- `clear()` member function
  - Restores the state to “good”

- Format error
- No characters are input

Stream Error State: An Example

```
#include <iostream>
using namespace std;

void print_state() {
  cout << "cin state:” << " good()=" << cin.good()
    << " eof()=" << cin.eof() << " fail()=" << cin.fail()
    << " bad()=" << cin.bad() << endl;
}

int main() {
  cout.flags(ios::boolalpha); // Alphanumerical bool values
  print_state();
  cout << "Please enter a number: ";
  int number;
  cin >> number;
  print_state();

  return 0;
}
```

```
cin state: good()=true eof()=false fail()=false bad()=false
Please enter a number: A
cin state: good()=false eof()=false fail()=true bad()=false
```