Object-Oriented Programming: C++ Stream Input/Output

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

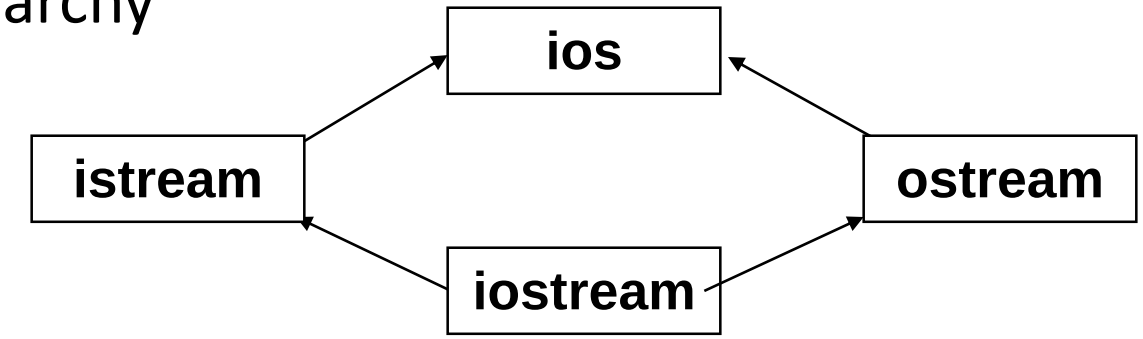
Slides credited from 李蔡彥 and 廖峻鋒

Input / Output

- I/O class hierarchy
- Overloading << and >>
- Using `cin` and `cout` as objects
- Checking I/O status
- Precise format control
- File inheritance
- Formatted file I/O
- Unformatted file I/O
- Random access file
- String stream processing

Basic C++ I/O Class Hierarchy

- I/O class hierarchy



- I/O symbols such as 'cout' are actually objects of I/O classes.

```
extern istream cin;  // stdin in C
extern ostream cout; // stdout in C
extern ostream cerr; // stderr in C
extern ostream clog;
```

- How does `cout << "hello"` work?

  The class ostream overloads << for all built-in type using inline functions. For user-defined types, you can overload >> and << as well.

Overloading << and >>

```
class PointT {
    friend ostream &operator<<(ostream &curStream, PointT p);
    friend istream &operator>>(istream &curStream, PointT p);
    ...
  private:
    int x, y;
};
ostream &operator<<(ostream &curStream, PointT p) {
    curStream <<"(" << p.x << ", " << p.y << ")";
    return curStream;
}
istream &operator>>(istream &curStream, PointT p) {
    cout << "Please input x:";
    curStream >> p.x;
    cout << "Please input y:";
    curStream >> p.y;
    return curStream;
}
```

- Why do we return the stream itself? For something like:

  `cout << p1 << p2 << endl;`

Formatted Stream Output

- `precision()`: controlling the number of digits to display

```
for(int i=0; i<5; i++) {
    cout.precision(i);
    cout << i <<' '<<pi << endl;
}
```

Output:

```
0 3
1 3
2 3.1
3 3.14
4 3.142
```

- `width()`: controlling the field width

```
double x = 1.2;
cout.width(4);
cout << x << "|\n";
cout.width(8);
cout << x << "|\n";
```

Output:

```
1.2 |
1.2      |
```

- `fill()`: specifying the char to be used as spacing.

```
double x = 1.2;
cout.fill('.');
cout.width(4);
cout << x << "|\n";
cout.width(8); // again
cout << x << "|\n";
```

Output:

```
1.2.|
1.2.....|
```

Grouped Formatting Flags [code]

- `setf(long fmtflags, long mask):` the second flag is the group flag while the first is a specific flag in the group flag.
- Setting scientific or fixed notation

```
double x=3.14159e5
cout.setf(ios::scientific, ios::floatfield);
cout << x << '\n';
cout.setf(ios::fixed, ios::floatfield);
cout << x << '\n';
```

Output:

```
3.14159e+5
314159.000000
```

- Setting justification

```
long x = -3456;
cout.width(10);
fmtflags old=cout.setf(ios::left, ios::adjustfield);
cout << x << '\n';
cout.width(10);
cout.setf(ios::internal, ios::adjustfield);
cout << x << '\n';
cout.width(10);
cout.setf(old, ios::adjustfield);
cout << x << '\n';
```

Output:

```
-3456
-      3456
      -3456
```

Default: right

Manipulators

- Manipulators: special words that perform formatting tasks.

```
#include <iomanip.h>
cout << pi << endl;
cout << "hello" << flush << "there\n";
```

endl and flush are manipulators

- Some I/O member functions have manipulator equivalents.

```
cout << setw(4) << x << setw(10) << y;
```

  - `setw` is the manipulator equivalent of `cout.width()`
  - The advantage is that they can be embedded within I/O statements.
  - Manipulators with arguments are called *parameterized stream manipulators.*

- Other examples:

`setprecision(4)` is equivalent to `cout.precision(4);`
`setfill('.')` is equivalent to `cout.fill('.');`

Other Useful I/O Functions

- Skipping white space. (White space is normally skipped.)

```
char x;
cin.unsetf(ios::skipws); // turn off skipping white space
cin >> x; // a white space can be read in
cout << x;
cin.setf(ios::skipws); // turn skipping white space on
```

- User-defined stream manipulators [code]

```
ostream &tab(ostream & currentStream) {
    return currentStream << '\t';
}
int main() {
    char x = 'A';
    cout << tab << x;
}
```

Other Useful I/O Functions [code]

- Changing the display to another base (8, 10, or 16)

```
int x = 15;
cout.setf(ios::oct, ios::basefield);
cout << x <<'\n';

cout.setf(ios::dec, ios::basefield);
cout << x <<'\n';

cout.setf(ios::hex, ios::basefield);
cout << x <<'\n';

cout << setbase(16) << x; // manipulator equivalent
```

Other Useful I/O Functions Continued

- Determining the current settings:

```
int currentPrecision;
int currentWidth;
int currentFill;
currentPrecision = cout.precision();
currentWidth = cout.width();
currentFill = cout.fill();
cout << currentPrecision << '\n';
cout << currentWidth << '\n';
cout << currentFill << '\n';
```

Output:
6
0
32 <space>

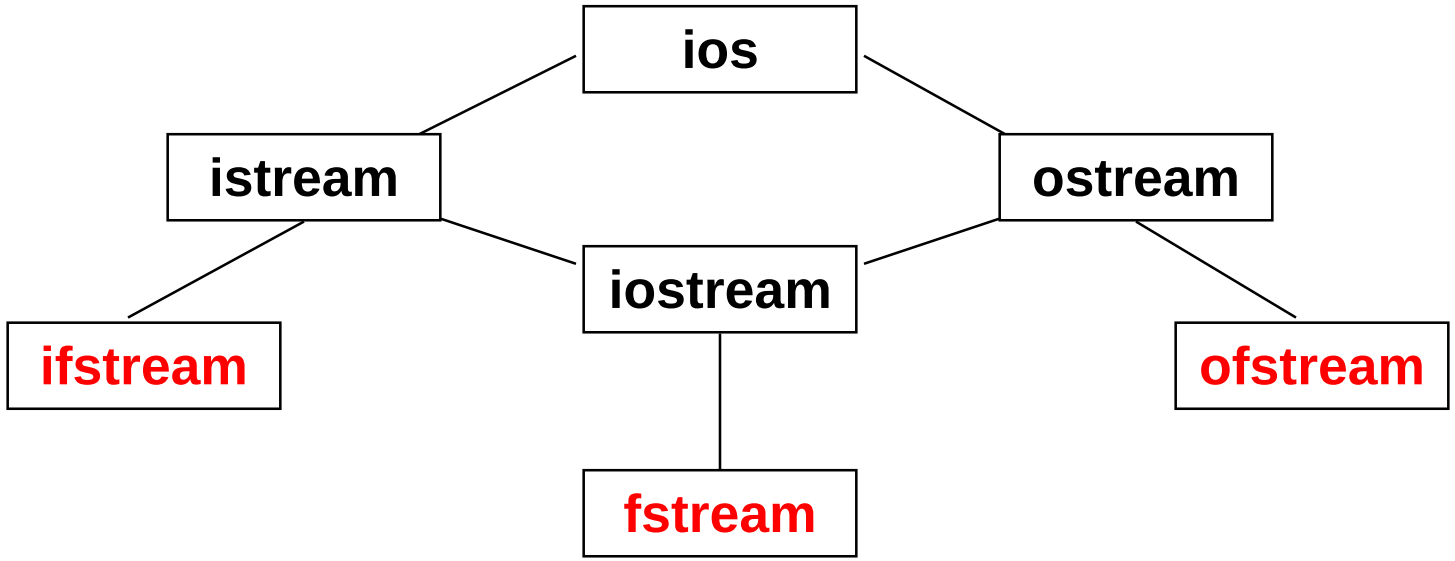
- Forcing floating-point numbers to show tailing zero.

```
double x=7;
cout << x << '\n';
cout.setf(ios::showpoint);
cout << x << '\n';
// manipulator equivalent
cout << showpoint << x << '\n';
```

Output:
7
7.00000
7.00000

File I/O Inheritance

- In C++ files classes are inherited from console classes.



- All member of the console classes are available in *exactly* the same form for file processing.
- Console I/O is always in formatted form but file I/O can be formatted or unformatted (raw bytes).

Unformatted File I/O [code]

- C++ uses member functions: *read* and *write* for binary I/O.

```
#include <iostream>
#include <fstream>
using namespace std;
const int kArraySize = 5;
int main() {
    int array[kArraySize], newArray[kArraySize];
    ofstream outFile("array.dat");
    if (!outFile) {
        cerr << "Can't open file: array.dat\n"; exit;
    }
    for(int i=0; i<kArraySize; i++)
        array[i] = i;
    outFile.write((char *)array, sizeof(int)*kArraySize);
    outFile.close();
    ifstream inFile("array.dat");
    if (!inFile) {
        cerr << "Can't open file: array.dat\n"; exit;
    }
    inFile.read((char *)newArray, sizeof(int)*kArraySize);
    for(int j=0; j<kArraySize; j++) {
        cout << newArray[j] << " ";
    }
}
```

Output:
0 1 2 3 4

Random Access Files

- Absolute file position

```
// seek get: offset from the beginning for istream
seekg(offset);

// seek put: offset from the beginning for ostream
seekp(offset);
```

- Relative file position

```
seekg(offset, ios::beg); // seek get from beginning
seekg(offset, ios::cur); // seek get from current
seekg(offset, ios::end); // seek get from end
seekp(offset, ios::beg); // seek put from beginning
etc.
```

  - The offset must be negative when ios::end is used.

- Current file position

```
tellg(); // return the current file position as a long
```

Using Random Access Files [code]

- Using a file as an array.

```
const int kMaxRooms=100;
const int kMaxName=100;
class HotelT {
  public:
   HotelT();
   void addRoom(int room, char *guest);
   void showRoom(int room);
  private:
   fstream ioFile; // file for input and output
   bool validRoom(int room);
};
HotelT::HotelT() {
   ioFile.open("hotel.dat", ios::in|ios::out);
   if (!ioFile) {
      cerr << "Can't open file: hotel.dat\n";
      exit(1);
   }
   char temp[kMaxName] = "";
   for(int i=0;i<kMaxRooms; i++)
      ioFile.write(temp, sizeof(temp));
}
```

Using Random Access Files Continued

```
void HotelT::addRoom(int room, char *guest) {
    char temp[kMaxName];
    if (validRoom(room)) {
        ioFile.seekp(room*sizeof(temp));
        ioFile.write(guest, strlen(guest)+1);
    }
}
void HotelT::showRoom(int room) {
    char guest[kMaxName];
    if (validRoom(room)) {
        ioFile.seekg(room*sizeof(guest));
        ioFile.read(guest, kMaxName);
        if (strcmp(guest, "") == 0)
            cout << "There is no one in room" << room << ".\n";
        else
            cout << guest <<  " is in room " << room << ".\n";
    }
}
int main() {
    HotelT grandHotel;
    grandHotel.addRoom(5, "Li");
    grandHotel.showRoom(5);
    grandHotel.showRoom(6);
}
```

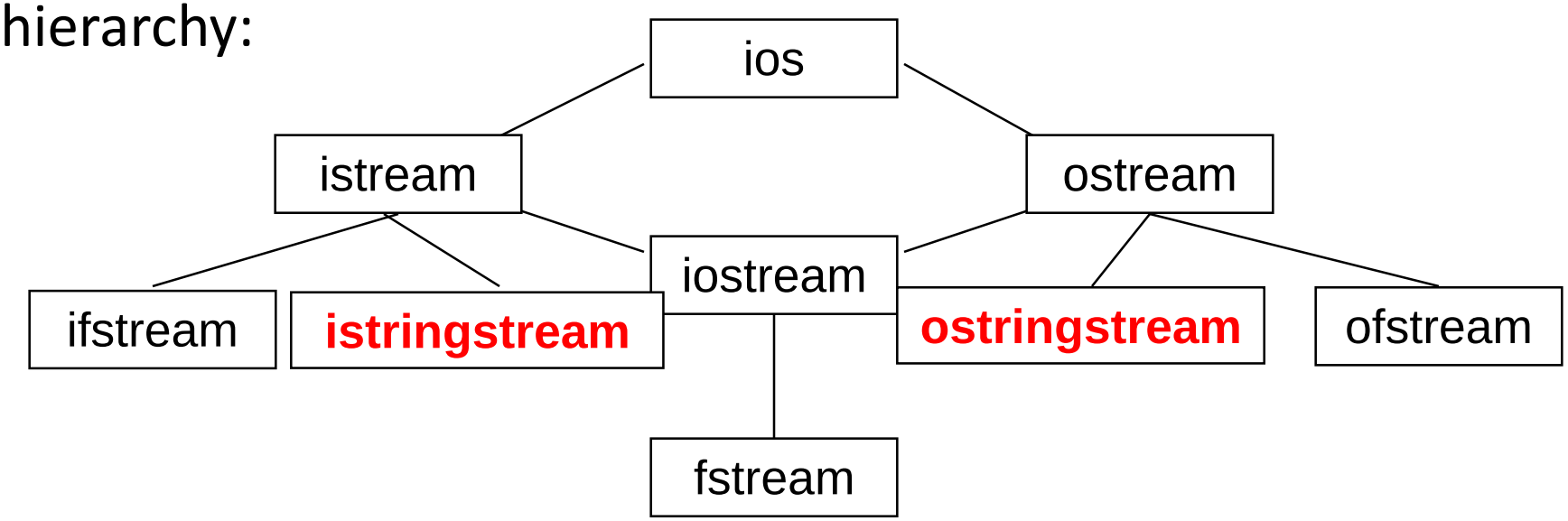
Output:

Li is in room 5.

There is no one in room 6.

String Stream Processing

- Class hierarchy:



- `istringstream` and `ostringstream` objects are effectively files that acts like a string.
- One can take advantage of console formatting to construct a string. (Like `sprintf`) this string can be saved to a file or output to the console at a later time.

Using Output String Stream

```
#include <strstream>
ostrstream outputStream;
char *result;
outputStream.precision(10);
outputStream << "The value of pi to a precision of 10 is"
             << Pi << ends; // ends put a NULL at the end
result = outputStream.str();
cout << result;
```

Output:

The value of pi to a precision of 10 is 3.141592653

The string is retrieved through the member function `str()`.

Once the client has called `str()`, not additional data can be added even if the string has not been null terminated.

```
result = outputStream.str();
outputStream << "more data";
if (outputStream.fail()) // this will be true
      cerr<<"failure";
```

Using Output String Stream Continued

- `ostrstream` has a second overloaded constructor whereby the client supplies the character array to be used.
- If the output string is longer than the buffer, the fail bit will be set.

```
const int kBufferSize = 10;
char buffer[kBufferSize];
ostrstream outputStream(buffer, kBufferSize);
char *result;
outputStream.precision(10);
outputStream << "The value of pi to a precision of 10 is"
<< pi << ends;
result = outputStream.str();
cout << result << '\n';
if (outputStream.fail()) // this will be true
   cout << "failure";
```

Output:
The value failure

Using `istrstream`

- An `istrstream` object contains a character array from which data can be extracted.

```
const int kBufferSize = 100;
const int kStringSize = 50;
int main() {
    char buffer[kBufferSize] = "pi is 3.14159";
    istrstream inputStream(buffer, kBufferSize);
    char string1[kStringSize], string2[kStringSize];
    double value;
    inputStream >> string1 >> string2 >> value;
   cout << string1 << '\n' << string2 << '\n' << value;
}
```

Output:
Pi
Is
3.14159

- The null terminator is treated as the `EOF`. Trying to extract beyond the null terminator will result in the `failbit` being set.