Introduction to Standard C++ Console I/O

C++ Object Oriented Programming

Pei-yih Ting

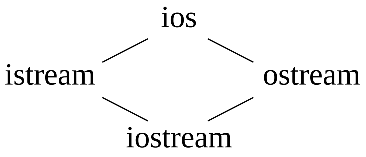
NTOU CS

Contents

- ✧ I/O class hierarchy, cin, cout
- ✧ << and >> operators
- ✧ Buffered I/O
- ✧ cin.get() and cin.getline()
- ✧ status of the stream
- ✧ Precise format control: width, precision, fill, grouped formatting flags, manipulators
- ✧ Odds and ends
- ✧ Types of I/O
- ✧ User-defined Types

Basic C++ I/O Class Hierarchy

- ✧ C++ performs all I/O through global objects in a class hierarchy



- ✧ Defined in <iostream>

```
namespace std
{
    …
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    …
}
```

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

Insertion operator <<

- ✧ The class *ostream* defines << operator for all the built-in types, ex:

```
ostream& ostream::operator<<(double x);    or
ostream& operator<<(ostream& out, double x);
```

- ✧ Usage:

sending "<< message" to cout object

```
double x;
cout << 2.54;
cout << x;
cout << 2.54 << x;
```

- ✧ Can be extended to handle user-defined types

```
CComplex x;
cout << x;
```

will be discussed after we introduce operator overloading

Extraction operator >>

- The class *istream* defines >> operator for all the built-in types, ex:

```
istream& istream::operator>>(double& x);        or
istream& operator>>(istream& in, double& x);
```

- Usage:

```
int x;
double y;
cin >> x;
cin >> y;
cin >> x >> y;
```

- Can be extended to handle user-defined types

```
CComplex x;
cin >> x;
```

will be discussed after we introduce operator overloading

Buffered I/O

- Buffer is implemented by an array of chars, meant to enhance the performance of input/output devices
- **cout** buffers the data and does not display immediately

```
int x;
cout << "hi" << "\n"; // may not be displayed immediately
while (true) x = 10;
```

```
FILE *fp;
…
fflush(fp);
```

- A simple trick to force a flush

```
cout << "hi" << endl;
```

- How to flush the buffer if you can’t wait until the end of line

```
cout << "hi" << flush << "bye";
```

- **cin** is buffered until you hit return

cin.get()

I. istream &istream::get(char &destination);

```
char cBuf;
cin.get(cBuf); // close to cin >> cBuf;
```

space, tab, newline

reference variable

skip white spaces

Not skipping white spaces

II. istream &istream::get(char *buffer, int length, char delimiter='\n');

- ➢ read up to length-1 characters or the delimiter character, whichever comes first and store them in the buffer
- ➢ the buffer is automatically terminated with a null char

```
const int kMaxChars = 100;
void main() {
    char buffer[kMaxChars];
    cin.get(buffer, kMaxChars);
}
```

default delimiter

cin.get()

- This get() does not remove the delimiter character from the stream

```
char buffer1[kMaxChars], buffer2[kMaxChars];
cin.get(buffer1, kMaxChars); // will read string input till ‘\n’
cin.get(buffer2, kMaxChars); // will read empty string
```

- ➢ Solution is to the last get() to “eat” the delimiter

```
cin.get(buffer1, kMaxChars);
char dummy; cin.get(dummy);   // or cin.ignore(1);
cin.get(buffer2, kMaxChars);
```

III. int istream::get();

the purpose of this function is to return EOF, will be useful when the input stream is a file

cin.getline() and others

- istream &istream::getline(char *buffer, int length,
  char delimiter='\n');

  this function is just like the second prototype of get() except that it eats the delimiter
- istream &istream::ignore(int length=1, int delimiter=EOF);
  - skips over length characters or until the delimiter is reached in the istream, whichever comes first
  - the delimiter is also removed from the stream
- int istream::peek();

  Return the next character in the stream without removing it, you can peek for EOF
- istream &istream::putback(char c);

  put the char back into the stream

Testing the State of the Stream

```
int GetSum() {
   char  badData;    int     number, sum;
   cout << "This program will compute the sum of numbers\nType zero to quit.\n ";
   sum = 0;
   while (true) {
      cout << "Type a number: ";
      cin >> number;
      if (cin.good()) {                // input was correct for this type
         if (number == 0) return sum;
         sum += number;
      }
      else if (cin.fail()) {           // error in input type, nothing serious
         cin.clear();                  // reset state bits in the base class
         cin.get(badData);             // read the bad input as a char
         cout << badData << " is not a number.";
      }
      else if (cin.bad())              // stream corrupted
         return sum;
   }
}
```

The base class **ios** contains a number of state bits which record the correctness of input and the output streams

Controlling the Output Format

- cout.precision() control the number of digits to display

```
for (i=0; i<8; i++) {
   cout.precision(i);
   cout << i << ' ' << pi << endl;
}
```

```
Output:
0  3.14159
1  3
2  3.1
3  3.14
4  3.142
5  3.1416
6  3.14159
7  3.141593
```

- cout.width() control the field width

  width must be set before every output

```
double x=5.6;
cout.width(4); cout << x << "first number\n";
cout.width(10); cout << x << "second number\n";
```

```
Output:
5.6 first number
5.6       second number
```

- cout.fill() specify the char to be used as spacing

```
cout.fill('.'); cout.width(10); cout << x << "first";
```

```
Output:
5.6…….first
```

Grouped Formatting Flags

- Certain formatting flags are members of bit groups, ex.
  - Setting scientific or fixed notation

```
double x;
x = 6.0225e23;
cout.setf(ios::scientific, ios::floatfield);
cout << x << '\n';
cout.setf(ios::fixed, ios::floatfield);
cout << x << '\n';
```

```
Output:
6.022500e+23
602250000000000000000000.000000
```

  - Setting justification

```
long x=-2345;
cout.width(10); cout.setf(ios::left, ios::adjustfield);
cout << x << '\n';
cout.width(10); cout.setf(ios::right, ios::adjustfield);
cout << x << '\n';
cout.width(10); cout.setf(ios::internal, ios::adjustfield);
cout << x << '\n';
```

```
Output:
-2345
     -2345
-     2345
```

Manipulators

- Special words that perform formatting tasks are called *manipulators*, ex.
  - cout << pi << endl;
  - cout << "hi" << flush << "bye";
- Some I/O member functions have manipulator equivalents
  - cout << setw(4) << x << setw(10) << y;

setw() is the parameterized manipulator equivalent of cout.width()

manipulator can be embedded within I/O statements

#include <iomanip>

- Other examples:

setprecision(4)	cout.precision(4)
setfill('x')	cout.fill('x')

Odds and Ends

- White spaces are skipped during stream extraction
  - You can turn this feature on or off

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

- User-defined stream manipulators
  - define tab manipulator

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

  - Usage: cout << tab << 'Z';

Odds and Ends

- Change the display to another base

```
cout.setf(ios::hex, ios::basefield);   // ios::dec, ios::oct
```

or using manipulators

```
cout << setbase(16) << x;  // 8, 10 or 16
```

- Current format settings

```
cout << cout.precision() << '\n';
cout << cout.width() << '\n';
cout << cout.fill() << '\n';
```

```
Output:
6
0
<space>
```

- Forcing floating-point displays

```
double x=7;
cout << x << '\n';
cout.setf(ios::showpoint); // no group
cout << x << '\n';
```

or using manipulators

```
cout << showpoint << x << '\n';
```

```
Output:
7
7.00000
```

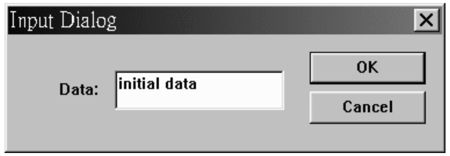
Types of I/O

- Plain vanilla applications

Input: user types in commands / Output: text written to a console window

- Dialog window approach (MFC)

```
CMyInputDialog dlg;
dlg.data = "initial data"; // output
dlg.DoModal();
strcpy(targetStr, dlg.data); // input
```



Data: initial data

- Explicit CFile class approach (MFC)

```
CFile infile;  CFileException e;
if (!infile.Open("test.dat", CFile::modeCreate | CFile::modeWrite, &e ) ) …
```

- Archive serialization approach (MFC)

```
void CAge::Serialize( CArchive& ar ) {
   CObject::Serialize( ar );
   if ( ar.IsStoring()) ar << m_years;
   else  ar >> m_years;
}
```

User-defined Types

- Old way, not suitably encapsulated:

```
CComplex number1(4, 2), number2(3, 1);
CComplex sum;
Sum = number1 + number2;
cout << sum.getReal() << " + " << sum.getImaginary() << 'i';
```

- Encapsulated:

```
cout << sum << endl;

ostream &operator<<(ostream &os, CComplex number)
{
   os << number.m_real << " + " << number.m_imaginary << 'i';
   return os;
}
```