Input/output

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: August 28, 2022

1: Formatted output

1. Formatted output

- `cout` uses default formatting
- Possible: pad a number, use limited precision, format as hex, etc
- Many of these output modifiers need

```
#include <iomanip>
```

2. Default unformatted output

Code:

```
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << i << "\n";
cout << "\n";
```

Output
[io] cunformat:

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

3. Reserve space

You can specify the number of positions, and the output is right aligned in that space by default:

Code:

```
#include <iomanip>
using std::setw;
  /* ... */
  cout << "Width is 6:" << "\n";
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setw(6) << i << "\n";
  cout << "\n";

  // 'setw' applies only once:
  cout << "Width is 6:" << "\n";
  cout << ">"
       << setw(6) << 1 << 2 << 3 <<
     "\n";
  cout << "\n";
```

Output

```
[io] width:

Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

Width is 6:
>     123
```

4. Padding character

Normally, padding is done with spaces, but you can specify other characters:

Code:

```
#include <iomanip>
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.')
         << setw(6) << i
         << "\n";
```

Output
[io] formatpad:

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Note: single quotes denote characters, double quotes denote strings.

5. Left alignment

Instead of right alignment you can do left:

Code:

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.')
         << setw(6) << i << "\n";
```

Output
[io] formatleft:

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

6. Number base

Finally, you can print in different number bases than 10:

Code:

```
#include <iomanip>
using std::setbase;
using std::setfill;
  /* ... */
  cout << setbase(16)
       << setfill(' ');
  for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
      cout << i*16+j << " " ;
    cout << "\n";
  }
```

Output
[io] format16:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a
    1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a
    2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a
    3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a
    4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a
    5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a
    6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a
    7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a
    8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a
```

Exercise 1

Make the first line in the above output align better with the other lines:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

Exercise 2

Use integer output to print real numbers aligned on the decimal:

Code:

```
string quasifix(double);
int main() {
  for ( auto x : { 1.5, 12.32,
     123.456, 1234.5678 } )
    cout << quasifix(x) << "\n";
```

Output

```
[io] quasifix:

   1.5
  12.32
 123.456
1234.5678
```

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

7. Hexadecimal

Hex output is useful for addresses (chapter **??**):

Code:

```
int i;
cout << "address of i, decimal: "
     << (long)&i << "\n";
cout << "address of i, hex    : "
     << std::hex << &i << "\n";
```

Output

```
[pointer] coutpoint:

address of i,
     decimal:
     140732703427524
address of i, hex
     : 0x7ffee2cbcbc4
```

Back to decimal:

```
cout << hex << i << dec << j;
```

2: Floating point formatting

8. Floating point precision

Use `setprecision` to set the number of digits before and after decimal point:

Code:

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
  /* ... */
  x = 1.234567;
  for (int i=0; i<10; i++) {
    cout << setprecision(4) << x <<
     "\n";
    x *= 10;
  }
```

Output
[io] formatfloat:

```
1.235
12.35
123.5
1235
1.235e+04
1.235e+05
1.235e+06
1.235e+07
1.235e+08
1.235e+09
```

This mode is a mix of fixed and floating point. See the `scientific` option below for consistent use of floating point format.

9. Fixed point precision

Fixed precision applies to fractional part:

Code:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setprecision(4) << x <<
     "\n";
  x *= 10;
}
```

Output

```
[io] fix:

1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

(Notice the rounding)

10. Aligned fixed point output

Combine width and precision:

Code:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setw(10) <<
   setprecision(4) << x
       << "\n";
  x *= 10;
}
```

Output
[io] align:

```
    1.2346
   12.3457
  123.4567
 1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

11. Scientific notation

Combining width and precision:

Code:

```
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
  cout << setw(10) << setprecision(4)
       << x << "\n";
  x *= 10;
}
cout << "\n";
```

Output

```
[io] iofsci:

1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

3: File output

12. Text output to file

The *iostream* is just one example of a stream:
general mechanism for converting entities to exportable form.
In particular: file output works the same as screen output.

Use:

Code:

```
#include <fstream>
using std::ofstream;
  /* ... */
  ofstream file_out;
  file_out.open
    ("fio_example.out");
  /* ... */
  file_out << number << "\n";
  file_out.close();
```

Output
[io] fio:

```
echo 24 | ./fio ; \
          cat
    fio_example.out
A number please:
Written.
24
```

Compare: `cout` is a stream that has already been opened to your terminal 'file'.

13. Binary output

Binary output: write your data byte-by-byte from memory to file.
(Why is that better than a printable representation?)

Code:

```
ofstream file_out;
file_out.open
  ("fio_binary.out",ios::binary);
/* ... */
file_out.write( (char*)(&number),4);
```

Output
[io] fiobin:

```
echo 25 | ./fiobin ;
    \
          od
    fio_binary.out
A number please:
    Written.
0000000    000031
    000000
0000004
```

4: Cout on classes (for future reference)

14. Redefine less-less

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

```
class container {
  /* ... */
  int value() const {
  /* ... */
  };
  /* ... */
ostream &operator<<(ostream &os,const container &i) {
  os << "Container: " << i.value();
  return os;
};
  /* ... */
  container eye(5);
  cout << eye << "\n";
```