Input/output

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: September 25, 2023

The fmtlib library

1. Simple example

The basic usage is:

```
format("string {} brace expressions",2);
```

Format string, and arguments.

2. Displaying the format result

```
auto s = std::format( /* formatting stuff */ );
cout << s.str() << '\n';
```

3. Right align

Right-align specifier:

Code:

```
1 // io/fmtlib.cpp
2     for (int i=10; i<2000000000;
       i*=10)
3       fmt::print("{:>6}\n",i);
```

Output:

```
    10
   100
  1000
 10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

4. Padding character

Other than space for padding:

Code:

```
1 // io/fmtlib.cpp
2     for (int i=10; i<2000000000;
       i*=10)
3       fmt::print("{0:.>6}\n",i);
```

Output:

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

5. Number bases

Code:

```
1 // io/fmtlib.cpp
2     fmt::print
3       ("{0} = {0:b} bin\n",17);
4     fmt::print
5       ("    = {0:o} oct\n",17);
6     fmt::print
7       ("    = {0:x} hex\n",17);
```

Output:

```
17 = 10001 bin
    = 21 oct
    = 11 hex
```

6. Float and fixed

Floating point or normalized exponential
fixed: use decimal point if it fits

Code:

```
1 // io/fmtfloat.cpp
2   x = 1.234567;
3   for (int i=0; i<6; ++i) {
4     fmt::print
5       ("{0:.3e}/{0:7.4}\n",x);
6     x *= 10;
7   }
```

Output:

```
1.235e+00/  1.235
1.235e+01/  12.35
1.235e+02/  123.5
1.235e+03/   1235
1.235e+04/1.235e+04
1.235e+05/1.235e+05
```

Formatted stream output

7. Formatted output

From `iostream`: `cout` uses default formatting.

Possible manipulation in `iomanip` header: pad a number, use limited precision, format as hex, etc.

8. Default unformatted output

Code:

```
1 // io/io.cpp
2   for (int i=1; i<200000000; i*=10)
3     cout << "Number: " << i << '\n';
```

Output:

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

9. Reserve space

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

Code:

```
1  // io/width.cpp
2  #include <iomanip>
3  using std::setw;
4      /* ... */
5    cout << "Width is 6:" << '\n';
6    for (int i=1; i<200000000; i*=10)
7      cout << "Number: "
8           << setw(6) << i << '\n';
9    cout << '\n';
10
11   // `setw' applies only once:
12   cout << "Width is 6:" << '\n';
13   cout << ">"
14        << setw(6) << 1 << 2 << 3 <<
        '\n';
15   cout << '\n';
```

Output:

```
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
```

10. Padding character

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

Code:

```
1 // io/formatpad.cpp
2 #include <iomanip>
3 using std::setfill;
4 using std::setw;
5     /* ... */
6   for (int i=1; i<200000000; i*=10)
7     cout << "Number: "
8          << setfill('.')
9          << setw(6) << i
10         << '\n';
```

Output:

```
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.

11. Left alignment

Instead of right alignment you can do left:

Code:

```
1  // io/formatleft.cpp
2  #include <iomanip>
3  using std::left;
4  using std::setfill;
5  using std::setw;
6      /* ... */
7    for (int i=1; i<200000000; i*=10)
8      cout << "Number: "
9           << left << setfill('.')
10          << setw(6) << i << '\n';
```

Output:

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

12. Number base

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

Code:

```
 1 // io/format16.cpp
 2 #include <iomanip>
 3 using std::setbase;
 4 using std::setfill;
 5     /* ... */
 6   cout << setbase(16)
 7        << setfill(' ');
 8   for (int i=0; i<16; ++i) {
 9     for (int j=0; j<16; ++j)
10       cout << i*16+j << " " ;
11     cout << '\n';
12   }
```

Output:

```
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 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
```

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:

```
1 // io/quasifix.cpp
2 string quasifix(double);
3 int main() {
4   for ( auto x : { 1.5, 12.32,
       123.456, 1234.5678 } )
5     cout << quasifix(x) << '\n';
```

Output:

```
   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.

13. Hexadecimal

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

Code:

```
1 // pointer/coutpoint.cpp
2   int i;
3   cout << "address of i, decimal: "
4        << (long)&i << '\n';
5   cout << "address of i, hex    : "
6        << std::hex << &i << '\n';
```

Output:

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

Back to decimal:

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

Floating point formatting

14. Floating point precision

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

Code:

```
1  // io/formatfloat.cpp
2  #include <iomanip>
3  using std::left;
4  using std::setfill;
5  using std::setw;
6  using std::setprecision;
7      /* ... */
8    x = 1.234567;
9    for (int i=0; i<10; ++i) {
10     cout << setprecision(4) << x <<
         '\n';
11     x *= 10;
12   }
```

Output:

```
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.

15. Fixed point precision

Fixed precision applies to fractional part:

Code:

```
1 // io/fix.cpp
2   x = 1.234567;
3   cout << fixed;
4   for (int i=0; i<10; ++i) {
5     cout << setprecision(4) << x <<
       '\n';
6     x *= 10;
7   }
```

Output:

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

(Notice the rounding)

16. Aligned fixed point output

Combine width and precision:

Code:

```
1 // io/align.cpp
2   x = 1.234567;
3   cout << fixed;
4   for (int i=0; i<10; ++i) {
5     cout << setw(10) <<
       setprecision(4) << x
6          << '\n';
7     x *= 10;
8   }
```

Output:

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

17. Scientific notation

Combining width and precision:

Code:

```
1 // io/iof.cpp
2   x = 1.234567;
3   cout << scientific;
4   for (int i=0; i<10; ++i) {
5     cout << setw(10) <<
       setprecision(4)
6          << x << '\n';
7     x *= 10;
8   }
9   cout << '\n';
```

Output:

```
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
```

File output

18. Text output to file

Use:

Code:

```
1 // io/fio.cpp
2 #include <fstream>
3 using std::ofstream;
4     /* ... */
5   ofstream file_out;
6   file_out.open
7     ("fio_example.out");
8     /* ... */
9   file_out << number << '\n';
10  file_out.close();
```

Output:

```
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'.

19. Binary output

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

Code:

```
1 // io/fiobin.cpp
2   ofstream file_out;
3   file_out.open
4     ("fio_binary.out",ios::binary);
5     /* ... */
6   file_out.write(
       (char*)(&number),4);
```

Output:

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

Cout on classes (for future reference)

20. 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.

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