

Input/output

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: December 3, 2024

1. I/O, what's it about?

Input: getting data from keyboard or file into your program.

Output: getting data from your program to screen or file.

The format header

2. Fmt header / library

From standard lib:

```
#include <format>  
using std::format;
```

If your compiler does not have this, from fmtlib:

```
#include <fmt/format.h>  
using fmt::format;
```

3. Simple example

The basic usage is:

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

Format string, and arguments.

4. Displaying the format result

Use `cout` or (C++23) `print`:

Code:

```
1 // iofmt/fmtbasic.cpp
2 cout << format("{}\n",2);
3 string hello_string = format
4   ("{} {}!", "Hello", "world");
5 cout << hello_string << '\n';
6 cout << format
7   ("{} {}, {} {} {}!\n",
8     "Hello", "world");
9 // c++23 only:
10 // print("{} {}, {} {} {}!\n",
11 //   "Hello", "world");
```

Output:

```
2
Hello world!
Hello, Hello world!
```

5. Right align

Right-align with > character and width:

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
```

6. 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
```


7. Number bases

Code:

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

Output:

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

8. Hex numbers

Display the numbers 0...255 in a square

```
for (int i=0; i<16; i++)  
for (int j=0; j<16; j++)  
    // output 16*i+j on base 16
```

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
e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	ea	eb	ec	ed	ee	ef
f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	fa	fb	fc	fd	fe	ff

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
```

9. Float and fixed

Floating point or normalized exponential with *e* specifier
fixed: use decimal point if it fits, *m.n* specification

Code:

```
1 // iofmt/fmtfloat.cpp
2 x = 1.234567;
3 for (int i=0; i<6; ++i) {
4     cout <<
5         format("{0:.3e}/{0:7.4}\n",
6             x);
7     x *= 10;
8 }
```

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
```

10. Treatment of leading sign

Positive sign always, nothing, blank:

Code:

```
1 // iofmt/fmtsci.cpp
2 float pi=3.14159f;
3 cout <<
4   format("|{:.2e}|{:.2e}|\n",
5         pi,-pi);
6 cout <<
7   format("|{:-.2e}|{:-.2e}|\n",
8         pi,-pi);
9 cout <<
10  format("|{: .2e}|{: .2e}|\n",
11         pi,-pi);
```

Output:

```
|+3.14e+00|-3.14e+00|
|3.14e+00|-3.14e+00|
| 3.14e+00|-3.14e+00|
```

11. fmtlib: usage

from fmtlib:

```
#include <fmt/format.h>  
using fmt::format;
```

12. fmtlib: installing

- Download: <https://github.com/fmtlib/fmt>
- Cmake installation
- add *lib/pkgconfig* to *PKG_CONFIG_PATH*

13. fmtlib: compilation

Compilation on the commandline:

```
g++ -o myprog myprog.cpp \  
    $( pkg-config --cflags fmt ) \  
    $( pkg-config --libs fmt )
```


14. fmtlib: compilation'

Using CMake:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
target_include_directories(
    ${PROGRAM_NAME} PUBLIC
    ${FMTLIB_INCLUDE_DIRS} )
target_link_directories(
    ${PROGRAM_NAME} PUBLIC
    ${FMTLIB_LIBRARY_DIRS} )
target_link_libraries(
    ${PROGRAM_NAME} PUBLIC ${FMTLIB_LIBRARIES} )
set_target_properties(
    ${PROGRAM_NAME} PROPERTIES
    BUILD_RPATH "${FMTLIB_LIBRARY_DIRS}"
    INSTALL_RPATH "${FMTLIB_LIBRARY_DIRS}"
)
```

15. fmtlib: use through pkg-config

When you install fmtlib, note the location of the .pc file, then

export

PKG_CONFIG_PATH=/the/location/from/fmtlib:\${PKG_CONFIG_PATH}

in your .bashrc (Mac users: .zshrc)

Formatted stream output

16. Formatted output

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

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

17. Default unformatted output

Code:

```
1 // io/io.cpp
2 for (int i=1; i<2000000000; 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
```

18. 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<2000000000;
    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
15     << '\n';
16 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
```

19. 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<2000000000;
      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.

20. 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<2000000000;
    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
```


21. 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;
7         i<16; ++i) {
9         for (int j=0;
7             j<16; ++j)
10             cout << i*16+j
11                 << " ";
12     cout << '\n';
13 }
```

Output:

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,
5                     123.456, 1234.5678 } )
6         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.

22. 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

23. 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
11         << '\n';
12     x *= 10;
```

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](#)

24. 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)

25. 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) <<
6         setprecision(4) << x
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
```

26. 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

27. 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'.

28. Binary I/O

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 cout << "Writing: " << x << '\n';
3 ofstream file_out;
4 file_out.open
5   ("fio_binary.out", ios::binary);
6 file_out.write
7   (reinterpret_cast<char*>(&x),
8    sizeof(double));
9 file_out.close();
```

Output:

Writing:
↪0.841471

`write` takes an address and the number of bytes.

29. Binary I/O'

Input is mirror of the output:

Code:

```
1 // io/fiobin.cpp
2 ifstream file_in;
3 file_in.open
4   ("fio_binary.out", ios::binary);
5 file_in.read
6   (reinterpret_cast<char*>(&x),
7    sizeof(double));
8 file_in.close();
9 cout << "Read    : " << x << '\n';
```

Output:

Read :
↪0.841471

Cout on classes (for future reference)

30. 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.

```
1 // stl/ostream.cpp
2 class container {
3     /* ... */
4     int value() const {
5         /* ... */
6     };
7     /* ... */
8     ostream &operator<<(ostream &os, const container &i) {
9         os << "Container: " << i.value();
10        return os;
11    };
12    /* ... */
13    container eye(5);
14    cout << eye << '\n';
```