Input/output

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: September 16, 2025

1. Input / Output

How do you get input data into your program?

How do you send output data to screen or file?

- Formatted output: readable, to screen or file.
- Binary: not readable, only to file.

Formatted output

2. Formatted output

Multiple ways to have readable output to screen or file:

- Use `printf` from C;
- From `iostream`: `cout` uses default formatting.
  Formatting manipulation in *iomanip* header.
- In C++ 20/23 use `format` and `print` headers.

3. Modern formatted output

The basic principle of formatting:

```
1 #include <format>
2 int i=2;
3 cout << std::format("substituting {} brace expressions\n",i);
4
5 #include <print>
6 std::print("substituting {} brace expressions\n",i);
7 // C++23:
8 std::println("substituting {} brace expressions",i);
```

Specify format string, and arguments.

4. Format arguments

Argument mechanism:

- Arguments indicated by curly braces in the format string;
- Optional: Braces can contain numbers (and modifiers, see next);
- This example uses both `format` and `println`.

Code:

```
1 // iofmt/fmtbasic.cpp
2 println("{}",2);
3 string hello_string = format
4   ("{} {}!","Hello","world");
5 cout << hello_string << '\n';
6 println
7   ("{0}, {0} {1}!",
8    "Hello","world");
```

Output:

```
1 2
2 Hello world!
3 Hello, Hello world!
```

5. Right align

Right-align with (optional arg number), colon, > character and integer width:

Code:

```
1 // iofmt/fmtlib.cpp
2 for (int i=10; i<200000000; i*=10)
3   // specifying arg 0 is optional
4   print("{0:>6}\n",i);
```

Output:

```
1       10
2      100
3     1000
4    10000
5   100000
6  1000000
7 10000000
8 100000000
```

6. Padding character

You can use other characters than space for padding:

Code:

```
1 // iofmt/fmtlib.cpp
2 for (int i=10; i<200000000; i*=10)
3   print("{0:.>6}\n",i);
```

Output:

```
1 ....10
2 ...100
3 ..1000
4 .10000
5 100000
6 1000000
7 10000000
8 100000000
```

7. Number bases

Other number bases than decimal:
binary, octal, and hexadecimal

Code:

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

Output:

```
1 17 = 10001 bin
2    = 21 oct
3    = 11 hex
```

8. 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   println
5     ("{0:.3e}/{0:7.4}",
6      x);
7   x *= 10;
8 }
```

Output:

```
1 1.235e+00/  1.235
2 1.235e+01/  12.35
3 1.235e+02/  123.5
4 1.235e+03/   1235
5 1.235e+04/1.235e+04
6 1.235e+05/1.235e+05
```

- `:3e` is float with 3 digits after the radix.
- `7.4` means 7 places and 4 digits of precision.

9. Treatment of leading sign

Positive sign always, if needed, replace by blank:

Code:

```
1 // iofmt/fmtsci.cpp
2 float pi=3.14159f;
3 println("|{:+.2e}|{:+.2e}|",
4         pi,-pi);
5 println("|{:-.2e}|{:-.2e}|",
6         pi,-pi);
7 println("|{: .2e}|{: .2e}|",
8         pi,-pi);
```

Output:

```
1 |+3.14e+00|-3.14e+00|
2 |3.14e+00|-3.14e+00|
3 | 3.14e+00|-3.14e+00|
```

10. Booleans

Booleans are by default printed as `true` or `false`:

```
1 format( "{}", true ); // gives `true'
```

To get them printed as zero or one, use the `:d` modifier:

```
1 format( "{:d}", true ); // gives `1'
```

11. Number base

Print base 16. Here is the naive output:

fmt/fmt256.cpp

```
(int i=0; i<16; ++i) {
r (int j=0; j<16; ++j)
print("{} ",i*16+j);
intln();
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29
    ↪30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45
    ↪46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61
    ↪62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77
    ↪78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93
    ↪94 95
96 97 98 99 100 101 102 103 104 105 106
    ↪107 108 109 110 111
112 113 114 115 116 117 118 119 120 121
    ↪122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137
    ↪138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153
    ↪154 155 156 157 158 159
```

Exercise 1

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

```
 1 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
 2 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
 3 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
 4 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
 5 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
 6 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
 7 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
 8 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
 9 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
10 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
11 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
12 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
13 c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
14 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
15 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
16 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Exercise 2

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     1.5
2    12.32
3   123.456
4 1234.5678
```

Hint: output integral and fractional part separately as integers. Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

Floating point formatting

12. 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  1.235
2  12.35
3  123.5
4  1235
5  1.235e+04
6  1.235e+05
7  1.235e+06
8  1.235e+07
9  1.235e+08
10 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.

13. 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 1.2346
2 12.3457
3 123.4567
4 1234.5670
5 12345.6700
6 123456.7000
7 1234567.0000
8 12345670.0000
9 123456700.0000
10 1234567000.0000
```

(Notice the rounding)

14. 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      1.2346
2     12.3457
3    123.4567
4   1234.5670
5 12345.6700
6 123456.7000
7 1234567.0000
8 12345670.0000
9 123456700.0000
10 1234567000.0000
```

15. 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 1.2346e+00
2 1.2346e+01
3 1.2346e+02
4 1.2346e+03
5 1.2346e+04
6 1.2346e+05
7 1.2346e+06
8 1.2346e+07
9 1.2346e+08
10 1.2346e+09
```

File output

16. 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:

```
1 echo 24 | ./fio ; \
2           cat
      ↪fio_example.out
3 A number please:
4 Written.
5 24
```

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

17. 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:

```
1 Writing:
      ↪0.841471
```

write takes an address and the number of bytes.

18. 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:

```
1 Read    :
      ↪0.841471
```

Cout on classes (for future reference)

19. Formatter for your class

```
1  // iofmt/fmtclass.cpp
2  template<>
3  class std::formatter<XYclass>{
4  public:
5    constexpr auto parse( std::format_parse_context& ctx ) {
6      return ctx.begin(); };
7    auto format( const XYclass& z,std::format_context& ctx ) const {
8      return std::format_to
9        ( ctx.out(),"[ {}, {} ]",z.x,z.y);
10   };
11 };
```