

# Input/output

Charlie Dey, Virginia Trueheart

Spring 2020

## Formatted output

# Formatted output

- cout uses default formatting
- Possible: pad a number, use limited precision, format as hex/base2, etc
- Many of these output modifiers need

```
#include <iomanip>
```

# Default unformatted output

## Code:

```
for (int i=1; i<2000000000; i*=10)
    cout << "Number: " << i << endl;
cout << endl;
```

## Output

[io] cunformat:

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

# Reserve space

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

## Code:

```
cout << "Width is 6:" << endl;
for (int i=1; i<200000000; i*=10)
    cout << "Number: "
        << setw(6) << i << endl;
cout << endl;

cout << "Width is 6:" << endl;
cout << "."
    << setw(6) << 1 << 2 << 3 << endl;
cout << endl;
```

## 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
```

# 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<2000000000; i*=10)
    cout << "Number: "
         << setfill('.')
         << setw(6) << i
         << endl;
```

**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.

# 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 << endl;
```

**Output**

**[io] formatleft:**

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

# 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 << endl;
}
```

**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	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
```

## 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) << endl;
```

**Output**

**[string] quasifix:**

make[4]: \*\*\* No rule to make tar

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

# Hexadecimal

Hex output is useful for pointers (chapter ??):

**Code:**

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

**Output**

**[pointer] coutpoint:**

```
address of i, decimal: 14073269  
address of i, hex      : 0x7ffee2
```

Back to decimal:

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

# Floating point formatting

# 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 <<
    endl;
    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
```

(Notice the rounding)

# 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 << endl;  
    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
```

# 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
        << endl;
    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
```

# Scientific notation

```
cout << "Combine width and precision:" << endl;
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4) << x << endl;
    x *= 10;
}
```



# Output

Combine width and precision:

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

# Text output to file

Streams are general: work the same for console out and file out.

```
#include <fstream>
```

Use:

Code:

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

Output

[io] fio:

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

# Binary output

## 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
```

**Cout on classes**

# 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 << endl;
```