Formatted I/O: printf and scanf

CSC 100 Introduction to programming in C/C++, Summer 2022

Table of Contents

- 1. README
- 2. printf
- 3. Conversion specification
- 4. Examples:
- 5. Integer decimal "d"
- 6. Floating point exponential "e"
- 7. Floating point fixed decimal "f"
- 8. Variable floating point "g"
- 9. scanf
- 10. First example
- 11. Main traps
- 12. How scanf works
- 13. Walk through example
- 14. Ordinary characters in format strings
- 15. Example with ordinary characters
- 16. Common mistakes:

1. README

- There is much more to `scanf` and `printf` than we've seen
- I/O is where the pedal hits the metal - where man meets machine
- In this notebook: conversion specifications for printf and scanf

2. printf

When it is called, `printf` must be supplied with:

1. a format string, like `"The output is: %d\n`
2. any number of values to be inserted into the string at printing
3. the values can also be computed

3. Conversion specification

- A **conversion specification** is a placeholder like `d`
- Binary (machine) format is converted to printed (human) format
- General form: `%m.pX` where

	WHAT	EXAMPLE
m	minimum field width	`%4d` prints `123` as `_123`
p	precision after point	`%.3f` prints `3.141593` as `3.142`

	WHAT	EXAMPLE
X	conversion specifier	`d`, `e`, `f`, `g`

4. Examples:

```
printf("....|....|....|\n");
printf("%8d\n", 123); // print 123 on 8 places (right-aligned)
printf("%-8d\n", 123); // print 123 on 8 places (left-aligned)
printf("%10.3f\n", 3.141593); // print 3 decimals on 10 places (right)
printf("%-10.3f\n", 3.141593); // print 3 decimals on 10 places (left)
```

```
....|....|....|
     123
123
     3.142
3.142
```

5. Integer decimal "d"

- `d` displays an integer in decimal (= base 10) form. `p` is the minimum number of digits to display the integer. Default is `p=1`.
- For example, the code below 1 prints numbers with different precision values:
  - `%d` displays `int` in decimal form (minimum amount of space)
  - `%5d` displays `int` in decimal form using 5 characters
  - `%-5d` displays `int` on 5 characters, left-justified
  - `%5.3d` displays `int` on 5 characters, at least 3 digits

```
int i = 40;
printf("....|....|\n");
printf("%d\n",i); // decimal form (minimum amount of space)
printf("%5d\n",i); // decimal form using 5 characters
printf("%-5d\n",i); // on 5 characters, left-justified
printf("%5.4d\n",i); // on 5 characters, at least 3 digits
```

```
....|....|
40
   40
40
 0040
```

6. Floating point exponential "e"

- `e` displays a floating-point number in exponential ("scientific") notation, e.g. `10. * 10. * 10. = 1000. = 1.0e+03`.
- `p` indicates the digits after decimal point. If `p=0`, no decimal point is displayed.

What went wrong in the first two statements?

```
printf("....|....|....|\n");
printf("%e\n", 1);
printf("%100.3e\n", 1000.);
printf("%-.1e\n", 1.);
printf("%e\n", 1000000000000000.);
printf("%15.f\n", 1000000000000000.);
```

```
....|....|....|
4.940656e-324

1.0e+00
1.000000e+15
1000000000000000
```

7. Floating point fixed decimal "f"

That's `f` as we already know it from many other examples. The precision p is defined as for `e`. Trailing zeroes are shown.

```
printf("....|....|\n");
printf("%10.3f\n", 100.1);
```

```
....|....|
   100.100
```

8. Variable floating point "g"

- `g` displays a floating point number in either exponential format or fixed decimal format depending on the number's size.
- `p` is the maximum number of **significant** digits (**not** digits after the decimal point!) to be displayed.
- No trailing zeroes are shown. If there are no decimal digits after the decimal point, no decimal point is shown.

```
printf("%g\n%g\n%g\n", 299792458., 1.45e+03, 8000);
```

299792000.0
1450
3.9525e-320

9. scanf

- A `scanf` **format string** may contain ordinary characters and

conversion specifications like `d`, `e`, `f`, `g`

- **The *conversions** allowed with `scanf` are essentially the same as those used with `printf`

- The `scanf` format string tends to contain **only** conversion specs

10. First example

- Example input:

```
1  -20  .3   -4.0e3
```

  Example program to read this input:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);

printf("|%5d|%5d|%5.1f|%10.1f|\n", i, j, x, y);
```

```
|    1|  -20|  0.3|   -4000.0|
```

  Create the input file:

```
echo "1  -20 .3 -4.0e+3" > ./data/io_scanf_input
```

11. Main traps

- The compiler will not check that specs and input match
- The `&` symbol may not miss in front of the input variable

12. How scanf works

- `scanf` is a pattern-matching function: it tries to math input groups with conversion specifications in the format string
- For each spec, it tries to locate an item in input
- It reads the item, and stops when it can't match
- If an item is not read successfully, `scanf` aborts

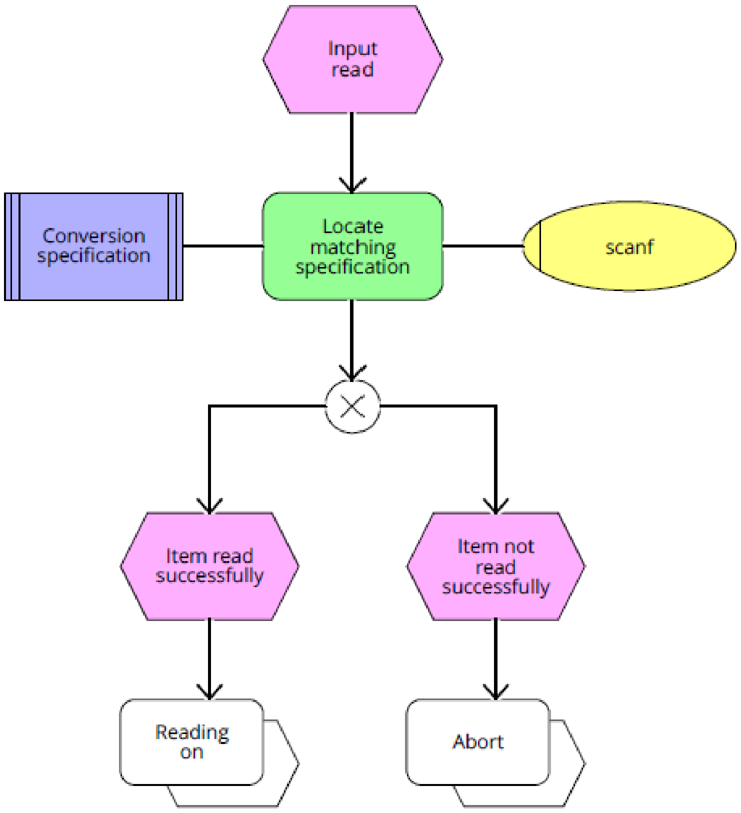
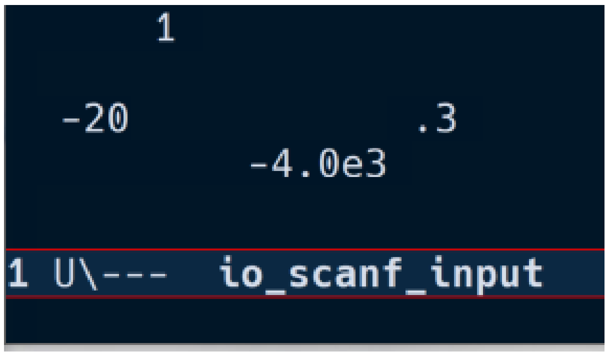


Figure 1: How scanf works (Event-controlled Process Chain diagram)

- White-space characters are ignored: space, TAB (`\t`), new-line (`\n`)
- In [1](#) above, the lines can be on one line or spread over

several lines:



```
      1

 -20                .3
        -4.0e3

1 U\---  io_scanf_input
```

Figure 2: Input file for tscanf

- `scanf` sees a character stream (`¤` = new-line, `s` = skip'd, `r` = read):

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```

- When asked to read an **integer** (`%d` or `%i`), `scanf` searches for a digit, or a +/- sign, then reads until it encounters a non-digit
- When asked to read a **float** (`%f`, `%g`, `%e`), `scanf` looks for +/- sign, digits, decimal point, or an exponent (`e+02`, `e-02`)
- When used with `scanf`, `%e`, `%f`, `%g` are interchangeable
- When it finds a character that cannot be part of the current item, the character is returned to be read again during the scanning of the next input item or the next call of `scanf`

13. Walk through example

The extended example below has the same spec as 1 - `"%d%d%f%f",&i,&j&x&y`

```
1-20.3-4.0e3¤
```

1. Expects `%d`. Stores `1` in `i`, returns `-`
2. Expects `%d`. Stores `-20` in `j`, returns `.`
3. Expects `%f`. Stores `0.3` in `x`, returns `-`
4. Expects `%f`. Stores `-4.0 x 10^3` in `y`, returns `¤`

14. Ordinary characters in format strings

- `scanf` reads white-space until it reaches a symbol
- When it reaches a symbol, it tries to match to next input
- It now either continues processing or aborts

15. Example with ordinary characters

- If the format string is `"%d/%d"` and the input is `•5/•96`, `scanf` succeeds.
- If the input is `•5•/•96` , `scanf` fails, because the `/` in the format string doesn't match the space in the input.
- Upon encountering the `/` in `•5•/•96`, `scanf` will abort, since it expects a digit or a +/- sign.
- To allow spaces after the first number, use `"%d /%d"` instead.

16. Common mistakes:

1. putting `&` in front of variables in a `printf` call

```
printf("%d %d\n", &i, &j);  /*** WRONG ***/
```

2. assuming that `scanf` should resemble `printf` formats

```
scanf("%d, %d", &i, &j);
```

- After storing `i`, `scanf` will try to match a comma with the next input character. If it's a space, it will abort.

- Only this input will work: `100, 100` but not `100 100`

3. putting a `\n` character at the end of `scanf` string

```
scanf("%d\n", &i);
```

   - To `scanf`, the new-line is *white-space*. It will advance to the next white-space character and not finding one will hang forever

Author: Marcus Birkenkrahe

Created: 2022-06-07 Tue 12:03