UNIVERSITY OF DEBRECEN
FACULTY OF INFORMATICS

Formatted input/output in the C language

Imre Varga PhD

Output

The most important tools for formatted output in C are the `printf()`, `fprintf()`, and `sprintf()` functions. Their return value is the number of characters printed successfully, or a negative value in case of error. A character represents one byte, but there are also multi-character symbols (e.g., escape characters). These functions have a variable number of parameters. Each has a so-called format string parameter, which determines what the character sequence to be written to the output data stream should look like. Values of different types can be inserted in the appropriate form at specific locations of this format string. In this case, the function always converts the given type of value into a string. The completed character sequence (determined from the format string and additional values) is put into the output data stream. In the case of `printf()`, it is printed to the screen (`stdout`), in the case of `fprintf()`, to a text file opened for writing, and in the case of `sprintf()`, to a string. The definitions of the functions (which are given in the `stdio.h` header file) are as follows:

```
int printf (const char * formatstring, ...);
int fprintf (FILE* stream, const char * formatstring, ...);
int sprintf (char* str, const char * formatstring, ...);
```

Below you will find a collection of examples that summarize the special expressions used in the format string.

Print a constant string to the current position of the "cursor". (In this simple case, the `puts()` function can be used as an alternative solution, but this also automatically places a line break at the end of the text, i.e., it always prints a separate line.)

```
printf("Hello World!");
```

← C code snippet

```
Hello World!
```

← output

Special characters

`\n`: New Line (Line Feed, LF). Starting a new line during output. `'\n'` is also a present character (a single byte), but since it is "non-printable", it is apparently missing.

```
printf("Hello\n World!");
```

```
Hello
World!
```

`\r`: Carriage Return (CR). Sets the current cursor position to the beginning of the current line, so that further writing occurs at the beginning of this line. In some cases, new lines in text files are implemented with a CR+LF character combination.

```
printf("abcdef\rABC");
```

```
ABCdef
```

\t: Tab. Appears as a horizontal break. Variable size, extending from a current position to a given next position. Single character, not the same as a sequence of spaces.

```
printf("A\tZ\nabc\txyz");
```

```
A       Z
abc     xyz
```

\a: Bell or Alert. Plays a default warning sound. Non-printable character, so it is not displayed visually.

```
printf("\a");
```

”Ding”

\b: Backspace. Deletes the character before the current cursor position in the line, moving the cursor back one character.

```
printf("abcdx\rdef");
```

```
abcdef
```

\": Double quote character. The beginning and end of a string are marked with a double quote character. If we want to print the double quote character itself, this escape sequence must be used.

```
printf("The word \"read\" is a verb.");
```

```
The word "read" is a verb.
```

\\: Backslash. It has a special meaning; it marks the beginning of an escape sequence. If we want to print the backslash character itself, this special escape sequence must be used.

```
printf("Windows path: C:\\Users\\John\\x.txt");
```

```
Windows path: C:\Users\John\x.txt
```

%%: Percent sign character. In the format string, the percent sign character has a special meaning, so if you want to print this character itself, you must write it twice.

```
printf("15%%duck 85%%chicken");
```

```
15%duck 85%chicken
```

Substituting fixed-point representation values

%d: Insert signed integer in decimal format. The %i can also be used.

```
int NumMoons=95;
printf("Jupiter has %d moons.",NumMoons);
```

```
Jupiter has 95 moons.
```

`%6d`: Inserts a signed integer using at least 6 characters with leading spaces if necessary. The displayed length can be modified. For example, it is useful for right-aligned columns, so that the place-value notation is written correctly.

```
int A=12, B=345, C=6, D=7890;
printf("%6d\n%6d\n%6d\n%6d\n",A,B,C,D);
```

```
    12
   345
     6
  7890
```

`%02d`: Insert a signed integer using at least 2 characters with a leading zero if necessary. The displayed length can be modified. Important if a specific number of digits is required (for example, a bank account number or a phone number is stored as a number).

```
int h=8, m=3, s=0;
printf("%02d:%02d:%02d",h,m,s);
```

```
08:03:00
```

`%-4d`: Inserts a signed integer using at least 4 characters (with spaces after it if necessary). The displayed length can be modified. For example, useful for left-aligned columns.

```
int a=21, b=9, c=136;
printf("%-4dapples\n%-4dbananas\n%-4dcherries",a,b,c);
```

```
21  apples
9   bananas
136 cherries
```

`%+d`: Insert a signed integer so that the positive sign is always explicitly printed. (If the sign is negative, it is always displayed.)

```
int N=210;
printf("%+d vs %d",N,N);
```

```
+210 vs 210
```

`%u`: Inserting an integer value with unsigned interpretation definitely.

```
int N=-1;
printf("%u vs %d",N,N);
```

```
4294967296 vs -1
```

`%x`: Inserts an integer in hexadecimal notation. The digits `a-f` are displayed in lowercase.

```
int A=700, B=0x700;
printf("%x != %x",A,B);
```

```
2bc != 700
```

`%X`: Inserts an integer in hexadecimal notation. The digits `a-f` are displayed in uppercase.

```
int A=700, B=0x700;
printf("%X != %X",A,B);
```

```
2BC != 700
```

`%#x`: Insert an integer in hexadecimal notation with "`0x`" prefix.

```
int A=700, B=0x700;
printf("%#x != %#x",A,B);
```

```
0x2BC != 0x700
```

`%o`: Insert an integer in octal notation.

```
int A=436, B=0664;
printf("%o == %o",A,B);
```

```
664 == 664
```

`%#o`: Insert an integer in octal notation with the usual "`0`" prefix.

```
int A=436, B=0664;
printf("%#o == %#o",A,B);
```

```
0664 == 0664
```

`%ld`: Inserts an 8-byte signed integer (`long int`) specified in decimal. (For `long long int` type, `%lld` is required.)

```
long int BigNumber=1000000000000;
printf("%ld != %d",BigNumber,BigNumber);
```

```
1000000000000 != -727379968
```

`%hi`: Insert a 2-byte short integer specified in decimal.

```
short int A=32767;
printf("%hi and %hi",A,A+1);
```

```
32767 and -32768
```

`%c`: Insert a character. The character can also be specified as a single-byte integer (a character code).

```
char c='A';
printf("%c and %c",c,66);
```

```
A and B
```

`%p`: Insert a memory address as a prefixed hexadecimal value. (Pointer values are also fixed-point data.)

```
int i=123;
int *P=&i;
printf("address of i: %p",P);
```

```
address of i: 0x7fffe6add9b8
```

Substituting floating-point values

`%f`: Inserts a single-precision floating-point value (`float`) displayed to 6 fractional digits, rounding to the last digit. (The imprecision of floating-point representation must be considered. For example, the float value `9876.54321` is displayed as `9876.542969`.)

```
float x=-12.3, y=0.000000567;
printf("%f %f",x,y);
```

```
-12.300000 0.000001
```

`%11f`: Inserts a single-precision floating-point value (float) of at least 11 characters, including 6 fractional digits by default, with rounding. The displayed length can be modified. For example, it is useful for right-aligned columns so that the place-value is written correctly.

```
float x=-12.34, y=0.000000567;
printf("%11f\n%11f",x,y);
```

```
  -12.340000
    0.000001
```

`%11.3f`: Inserts a single-precision floating-point value (float) of at least 11 characters, including 3 fractional digits, with rounding. The displayed length can be modified. For example, it is useful for right-aligned columns so that the place-value is written correctly.

```
float x=-12.345678, y=0.0004321;
printf("%11.3f\n%11.3f",x,y);
```

```
    -12.346
      0.000
```

`%-11.3f`: Insert a single-precision floating-point value (float) of at least 11 characters, displayed with 3 fractional digits, with rounding and left-justified. The displayed length can be modified.

```
float x=123.45;
printf("%11.3fUSD",x);
```

```
123.450   USD
```

`%011f`: Inserts a single-precision floating-point value (float) at least 11 characters right-justified, with leading zeros, if necessary, displayed with 6 fractional digits by default with rounding. The displayed length can be modified.

```
float x=12.34, y=0.12345678;
printf("%11f\n%11f",x,y);
```

```
00012.340000
00000.123457
```

`%lf`: Inserts a double-precision floating-point value (`double`) displayed with 6 fractional digits with rounding.

```
float x=123.45678987654;
printf("%f %f",x,y);
```

```
123.456790
```

`%e`: Insert a floating-point value in scientific form with `'e'` notation. This is a kind of normal form: `1.234560e+03` → $1.23456 \times 10^{+3}$.

```
float x=1234.56,0.000987;
printf("%e %e",x,y);
```

```
1.234560e+03 9.869999e+04
```

`%E`: Insert a floating-point value in scientific form with `'E'` notation. This is a kind of normal form: `9.869999E-04` → 9.87×10^{-4}.

```
float x=1234.56,0.000987;
printf("%E %E",x,y);
```

```
1.234560E+03 9.869999E+04
```

`%g`: Insert a floating-point value so that it is displayed with the minimum number of decimal places compared to the value (i.e., it omits trailing zeros in the fractional part). If the fractional part is 0, then do not even print the decimal point. (If it is more convenient, it uses scientific notation. In this case, it uses the notation `'e'` for `%g` and the notation `'E'` for `%G`.)

```
float w=123.0;
float x=123.450000;
float y=12.3456789;
float z=12345678.9;
printf("%g\n%g\n%g\n%g",w,x,y,z);
```

```
123
123.45
12.3457
1.23457e+07
```

Substituting strings

`%s`: Substitutes a character sequence into the format string. Inserts as many characters as the length of the given string.

```
char name[10]="John";
printf("Hello %s!",name);
```

```
Hello John!
```

`%8s`: Substitute a character sequence into the format string, at least 8 characters long, right-aligned, with spaces inserted from the left. The displayed length can be modified.

```
char name[10]="John";
printf("Hello %s!",name);
```

```
Hello     John!
```

`%-8s`: Substitute a character sequence into the format string, at least 8 characters long, left-aligned, with spaces inserted to the right. The displayed length can be modified.

```
char name[10]="John";
printf("Hello %s!",name);
```

```
H e l l o   J o h n           !
```

Managing colors and styles

`\033[0m`: This ANSI escape code (re)sets the default color and style in the consol. (The previously changed color and style will still have an effect until this.)

```
printf("\033[0mColored text!");
```

```
C o l o r e d   t e x t !
```

`\033[1m`: The number after the `'['` specifies the style of the text. For example, `1` is bold. Unless a different value is specified, each subsequent call of `printf()` will use the last setting. The meaning of the digits describing the style is summarized in the following table.

style digit	description
0	normal (reset)
1	bold
2	faint
3	italic
4	underline
5, 6	blink
7	inverted (swapped FG and BG color)
8	hide
9	crossed-out (strike)

```
printf("\033[1mColored text!");
```

```
C o l o r e d   t e x t !
```

`\033[;31m`: The number after `';'` specifies the foreground color of the text. For example, `31` means red. Until a different value is set, each subsequent call of `printf()` will use the last setting. The meanings of the numbers describing the foreground color are summarized in the following table.

code	foreground color
30	black
31	red
32	green
33	yellow
34	blue
35	magenta
36	cyan
37	white

```
printf("\033[;31mColored text!");
```

```
C o l o r e d   t e x t !
```

`\033[;;42m`: The number after the second `';'` specifies the background color. For example, `42` means green. Until a different value is set, each subsequent call of `printf()` will use the last setting. The meanings of the numbers describing the background color are summarized in the following table.

code	background color
40	black
41	red
42	green
43	yellow
44	blue
45	magenta
46	cyan
47	white

```
printf("\033[;;42mColored text!");
```

`\033[3;34;43m`: Italicized, blue text on a yellow background. After displaying the text, everything returns to the default settings. (After the `"\033["` and before the `'m'` előtt three numbers can be specified: style, foreground color, background color. (These must be separated by semicolons. If any of the values are missing, the last setting remains active.)

```
printf("\033[4;34;43mColored text!\033[0m");
```

Input

The most important tools for formatted input in the C language are `scanf()`, `fscanf()` and `sscanf()`. These functions have a variable number of parameters. Each of them has a so-called format string parameter, which determines the formal expectations for the input character string. Based on the format string, the individual substrings of the input data stream are converted to the data type corresponding to the format and are saved in the variables defined by the parameters after the format string. (These parameters must therefore be memory addresses, i.e., pseudo-address-based parameter passing must be used.) If the content of the input does not correspond to the prescribed format, the reading stops, and the function terminates. The return value of the functions is the number of successfully read values, i.e. the integer that tells how many of the variables after the format parameter were successfully initialized based on the input. This can be used to decide whether the reading was successful and how many variable values come from the input data stream. The input stream for `scanf()` is the keyboard (`stdin`), for `fscanf()` it is a text file opened for reading, and for `sscanf()` it is a string. The definitions of the functions (which are specified in the `stdio.h` header file) are as follows:

```
int scanf (const char * formatstring, ...);
int fscanf (FILE* stream, const char * formatstring, ...);
int sscanf (char* str, const char * formatstring, ...);
```

Below you will find a collection of examples that summarize the special expressions used in the format string and the different options for reading data.

Reading simple data

`%s`: Read a word into an array of characters. The reading continues until the first whitespace (space, tab, new line) or the end of the input stream (`stdin`), whichever comes first. (If the word to be read is longer than the array of characters specified to hold it, this can cause problems.)

```
J o h n   S m i t h
char name[10];
scanf("%s",name);
```

name	"John"

← input

← C code snippet

← new value of variable

`%4s`: Read a word, but a maximum of 4 characters, into a character array. The length can be modified.

```
t e m p e r a t u r e
char word[10];
scanf("%4s",word);
```

word	"temp"

`%c`: Reading a character (1 byte).

```
+ * % &
char b;
scanf("%c",&b);
```

b	'+'

`%d`: Reading a string that can be interpreted as an integer, converts it to an `int` value, and stores it in the specified variable.

```
1 2 3 4 . 5 U S D
int i;
scanf("%d",&i);
```

i	1234

`%2d`: Reading a string of up to 2 characters long, interpreted as an integer, from the input stream, converts it to an `int` value, and stores it in the specified variable. The length can be modified.

```
1 2 3 4 . 5 U S D
int i;
scanf("%d",&i);
```

i	12

`%f`: Reading a string of characters that can be interpreted as a real (fractional decimal) number from the input data stream, converts it to a `float` value, and stores it in the

specified variable. (The real number can also be specified in scientific notation. In the case of a double-precision floating-point variable, the `%lf` format specifier should be used.) If the string does not contain a `'.'` character (i.e., the value looks like an integer), it will still be stored as a floating-point number.

```
1234.5USD
float price;
scanf("%f",&price);
```

price	1234.5

`%5f`: Reading a string of up to 5 characters long, interpreted as a real (fractional decimal) number, from the input data stream, converts it to a `float` value, and stores it in the specified variable. (The real number can also be specified in scientific notation.) The length can be modified. (The length includes any sign and decimal point as well.)

```
123.45USD
float price;
scanf("%5f",&price);
```

price	123.4

`%*c, %*d, %*f, %*s`: The value of a given type (character, integer, real number, string) is read but not saved into a variable. It is essentially omitting value. The length can also be specified.

```
1st 2nd 3rd
int a,b,c;
scanf("%d%*2c %d%*2c %d%*2c",&a,&b,&c);
```

a	1
b	2
c	3

Examples for complex reading formats

e-mail: Retrieving an email address with the username and domain stored in separate strings.

```
john.doe@gmail.com
char user[20], domain[20];
scanf("%s@%s",user,domain);
```

user	"john.doe"
domain	"gmail.com"

phone number: Splitting a Hungarian landline phone number into parts. (The parts will be treated as whole numbers.)

```
06525129007516
int areacode,phonenumber,extension;
scanf("06%2d%6d%d",&areacode,&phonenumber,&extension);
```

areacode	52
phonenumber	512900

extension	75016

weather: Reading a tab-separated set of temperature and humidity data into separate variables.

```
23.5°C  49%
float temperature; int humidity;
scanf("%f°C\t%d",&temperature,&humidity);
```

temperature	23.5
humidity	49

Using regular expressions (regex)

`%[+-*/%]`: Read a string containing only basic arithmetic operators. The "allowed" characters specified in square brackets are read; if any other characters are found in the input stream, the reading is terminated.

```
++operator
char s[10];
scanf("%[+-*/%]",s);
```

s	"++"

`%[a-z]`: Reading a string containing only lowercase English letters. Reading continues until a character is found in the input stream that is outside the range of `'a'` and `'z'` (ASCII 97-122). The regular expression `%[A-Z]` is used to read a string containing uppercase English letters (`'A'-'Z'`).

```
var=2.5;
char letters[10];
scanf("%[a-z]",letters);
```

letters	"var"

`%[0-9]`: Read a string containing only decimal digit characters. (The following example reads a string of up to 16 characters long.)

```
1234567890123456789
char BankCardNumber[17];
scanf("%16[a-z]",BankCardNumber);
```

BankCardNumber	"1234567890123456"

`%[a-zA-z0-9_]`: Reading a contiguous string containing only alphanumeric or underscore characters.

```
Get_Age("John Doe");
char subroutin[20];
scanf("%[a-zA-Z0-9_]",subroutin);
```

subroutin	"Get_Age"

`%[^,]`: Reading a string that lasts until the next comma character. So, the reading continues until the character specified after ^ is reached in the input stream. The character that marks

the end of the reading is not read. For example, it can be useful when processing *.csv files. Another useful example is the `%[\n]` notation, which is used when reading line by line.

```
John Doe,New York
char name[20];
scanf("%[^,]",name);
```

name	"John Doe"

`%*[+-]`: Sign discarding, i.e. if the next character is `'+'` or `'-'`, it will be read but not saved. In the example below, an integer absolute value will be read regardless of sign.

```
-123
int absval;
scanf("%*[+-]%d",&absval);
```

absval	123