printf

function

```
int printf ( const char * format, ... );
```
<cstdio>

Print formatted data to stdout

Writes to the standard output (stdout) a sequence of data formatted as the *format* argument specifies. After the *format* parameter, the function expects at least as many additional arguments as specified in *format*.

Parameters

format

String that contains the text to be written to `stdout`. It can optionally contain embedded format tags that are substituted by the values specified in subsequent argument(s) and formatted as requested. The number of arguments following the *format* parameters should at least be as much as the number of format tags. The format tags follow this prototype:

%[flags][width][.precision][length]specifier

Where *specifier* is the most significant one and defines the type and the interpretation of the value of the corresponding argument:

specifier	**Output**	**Example**
`c`	Character	`a`
`d` or `i`	Signed decimal integer	`392`
`e`	Scientific notation (mantise/exponent) using `e` character	`3.9265e+2`
`E`	Scientific notation (mantise/exponent) using `E` character	`3.9265E+2`
`f`	Decimal floating point	`392.65`
`g`	Use the shorter of `%e` or `%f`	`392.65`
`G`	Use the shorter of `%E` or `%f`	`392.65`
`o`	Signed octal	`610`
`s`	String of characters	`sample`
`u`	Unsigned decimal integer	`7235`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (capital letters)	`7FA`
`p`	Pointer address	`B800:0000`
`n`	Nothing printed. The argument must be a pointer to a signed `int`, where the number of characters written so far is stored.	
`%`	A `%` followed by another `%` character will write `%` to `stdout`.	

The tag can also contain ***flags***, ***width***, ***.precision*** and ***modifiers*** sub-specifiers, which are optional and follow these specifications:

flags	**description**
`-`	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
`+`	Forces to preceed the result with a plus or minus sign (`+` or `−`) even for positive numbers. By default, only negative numbers are preceded with a `−` sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
`#`	Used with `o`, `x` or `X` specifiers the value is preceeded with `0`, `0x` or `0X` respectively for values different than zero. Used with `e`, `E` and `f`, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with `g` or `G` the result is the same as with `e` or `E` but trailing zeros are not removed.
`0`	Left-pads the number with zeroes (`0`) instead of spaces, where padding is specified (see *width* sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
`*`	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument thas has to be formatted.
.precision	**description**
.number	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of `0` means that no character is written for the value `0`. For `e`, `E` and `f` specifiers: this is the number of rounded digits to be printed after the decimal point. For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For `c` type: it has no effect. When no *precision* is specified, the default is `1`. If the period is specified without an explicit value for *precision*, `0` is assumed.
`.*`	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument thas has to be formatted.

length	description
h	The argument is interpreted as a `short int` or `unsigned short int` (only applies to integer specifiers: `i`, `d`, `o`, `u`, `x` and `X`).
l	The argument is interpreted as a `long int` or `unsigned long int` for integer specifiers (`i`, `d`, `o`, `u`, `x` and `X`), and as a wide character or wide character string for specifiers `c` and `s`.
L	The argument is interpreted as a `long double` (only applies to floating point specifiers: `e`, `E`, `f`, `g` and `G`).

additional arguments

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each `%`-tag specified in the *format* parameter, if any. There should be the same number of these arguments as the number of `%`-tags that expect a value.

Return Value

On success, the total number of characters written is returned. On failure, a negative number is returned.

scanf

function

```
int  scanf ( const char * format, ... );
```
<cstdio>

Read formatted data from stdin

Reads data from `stdin` and stores them according to the parameter *format* into the locations pointed by the additional arguments. The additional arguments should point to already allocated objects of the type specified by their corresponding format tag within the *format* string.

Parameters

format

C string that contains one or more of the following items:

- **Whitespace character:** the function will read and ignore any whitespace characters (this includes blank spaces and the newline and tab characters) which are encountered before the next non-whitespace character. This includes any quantity of whitespace characters, or none.
- **Non-whitespace character, except percentage signs (%):** Any character that is not either a whitespace character (blank, newline or tab) or part of a format specifier (which begin with a `%` character) causes the function to read the next character from `stdin`, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of `stdin` unread.
- **Format specifiers:** A sequence formed by an initial percentage sign (`%`) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from `stdin` and stored in the locations pointed to by the additional arguments. A format specifier follows this prototype:

%[*][width][modifiers]type

where:

`*`	An optional starting asterisk indicates that the data is to be retrieved from `stdin` but ignored, i.e. it is not stored in the corresponding argument.
width	Specifies the maximum number of characters to be read in the current reading operation
modifiers	Specifies a size different from `int` (in the case of `d`, `i` and `n`), `unsigned int` (in the case of `o`, `u` and `x`) or `float` (in the case of `e`, `f` and `g`) for the data pointed by the corresponding additional argument: **h :** short int (for d, i and n), or unsigned short int (for o, u and x) **l :** long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) **L :** long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

scanf type specifiers:

type	Qualifying Input	Type of argument
`c`	**Single character:** Reads the next character. If a *width* different from 1 is specified, the function reads *width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	`char *`
`d`	**Decimal integer:** Number optionally preceeded with a + or − sign.	`int *`
`e,E,f,g,G`	**Floating point:** Decimal number containing a decimal point, optionally preceeded by a + or − sign and optionally folowed by the `e` or `E` character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	`float *`
`o`	**Octal integer**.	`int *`

s	**String of characters**. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x,X	Hexadecimal integer.	int *

additional arguments

The function expects a sequence of references as additional arguments, each one pointing to an object of the type specified by their corresponding %-tag within the *format* string, in the same order.
For each format specifier in the *format* string that retrieves data, an additional argument should be specified. These arguments are expected to be references (pointers): if you want to store the result of a fscanf operation on a regular variable you should precede its identifier with the *reference operator*, i.e. an ampersand sign (&), like in:

```
int n;
scanf ("%d",&n);
```

Return Value

On success, the function returns the number of items succesfully read. This count can match the expected number of readings or fewer, even zero, if a matching failure happens. In the case of an input failure before any data could be successfully read, EOF is returned.

printf Example

```
/* printf example */
#include <cstdio>
using namespace std;

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radixes: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
   printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
   return 0;
}
```

And here is the output:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radixes: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

scanf Example

```
/* scanf example */
#include <cstdio>
using namespace std;

int main ()
{
  char str [80];
  int i;

  printf ("Enter your family name: ");
  scanf  ("%s",str);

  printf ("Enter your age: ");
  scanf  ("%d",&i);

  printf ("Mr. %s , %d years old.\n",str,i);

  printf ("Enter a hexadecimal number: ");
  scanf  ("%x",&i);
  printf ("You have entered %#x (%d).\n",i,i);

  return 0;
}
```

This example demonstrates some of the types that can be read with `scanf`:

```
Enter your family name: Soulie
Enter your age: 29
Mr. Soulie , 29 years old.
Enter a hexadecimal number: ff
You have entered 0xff (255).
```