function

scanf

<cstdio>

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

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 specifier within the *format* string.

Parameters

`format`

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- **Whitespace character:** the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- see isspace). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- **Non-whitespace character, except format specifier (%):** 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 the stream, 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 the stream 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 the *stream* and stored into the locations pointed by the additional arguments.

A *format specifier* for `scanf` follows this prototype:

```
%[*][width][length]specifier
```

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

specifier	Description	Characters extracted
`i`	Integer	Any number of digits, optionally preceded by a sign (`+` or `-`).

		Decimal digits assumed by default (`0-9`), but a `0` prefix introduces octal digits (`0-7`), and `0x` hexadecimal digits (`0-f`). *Signed* argument.
`d` *or* `u`	Decimal integer	Any number of decimal digits (`0-9`), optionally preceded by a sign (`+` or `-`). `d` is for a *signed* argument, and `u` for an *unsigned*.
`o`	Octal integer	Any number of octal digits (`0-7`), optionally preceded by a sign (`+` or `-`). *Unsigned* argument.
`x`	Hexadecimal integer	Any number of hexadecimal digits (`0-9`, `a-f`, `A-F`), optionally preceded by `0x` or `0X`, and all optionally preceded by a sign (`+` or `-`). *Unsigned* argument.
`f`, `e`, `g`	Floating point number	A series of decimal digits, optionally containing a decimal point, optionally preceeded by a sign (`+` or `-`) and optionally followed by the `e` or `E` character and a decimal integer (or some of the other sequences supported by `strtod`). Implementations complying with C99 also support hexadecimal floating-point format when preceded by `0x` or `0X`.
`a`		
`c`	Character	The next character. If a *width* other than `1` is specified, the function reads exactly *width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
`s`	String of characters	Any number of non-whitespace characters, stopping at the first whitespace character found. A terminating null character is automatically added at the end of the stored sequence.
`p`	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format `%p` in fprintf.
`[`*characters*`]`	Scanset	Any number of the characters specified between the brackets. A dash (`-`) that is not the first character may produce non-portable behavior in some library implementations.
`[^`*characters*`]`	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.
`n`	Count	No input is consumed. The number of characters read so far from `stdin` is stored in the pointed location.
`%`	`%`	A `%` followed by another `%` matches a single `%`.

Except for `n`, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: *asterisk* (`*`), *width* and *length* (in that order), which are optional and follow these specifications:

sub-specifier	**description**
`*`	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of `hh`, `h`, `l`, `ll`, `j`, `z`, `t`, `L` (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a *length* sub-specifier):

	specifiers					
length	**d i**	**u o x**	**f e g a**	**c s [] [^]**	**p**	**n**
(none)	`int*`	`unsigned int*`	`float*`	`char*`	`void**`	`int*`

hh	signed char*	unsigned char*				signed char*
h	short int*	unsigned short int*				short int*
l	long int*	unsigned long int*	double*	wchar_t*		long int*
ll	long long int*	unsigned long long int*				long long int*
j	intmax_t*	uintmax_t*				intmax_t*
z	size_t*	size_t*				size_t*
t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*
L			long double*			

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.

... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.
There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.
These arguments are expected to be pointers: to store the result of a `scanf` operation on a regular variable, its name should be preceded by the *reference operator* (`&`) (see example).

Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (feof or ferror). And, if either happens before any data could be successfully read, EOF is returned.

If an encoding error happens interpreting wide characters, the function sets errno to `EILSEQ`.

Example

```c
/* scanf example */
#include <stdio.h>

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

  printf ("Enter your family name: ");
  scanf ("%79s",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;
}
```

Edit & Run

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).
```

Compatibility

Particular library implementations may support additional specifiers and sub-specifiers.
Those listed here are supported by the latest C and C++ standards (both published in 2011), but those in yellow were introduced by C99 (only required for C++ implementations since C++11), and may not be supported by libraries that comply with older standards.

See also

fscanf	Read formatted data from stream (function)
printf	Print formatted data to stdout (function)
gets	Get string from stdin (function)
fopen	Open file (function)