NAME

scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf − input format conversion

SYNOPSIS

#include <stdio.h>
int scanf(const char * *format*, **...*);***
int fscanf(FILE * *stream*, **const char *** *format*, **...*);***
int sscanf(const char * *str*, **const char *** *format*, **...*);***

#include <stdarg.h>
int vscanf(const char * *format*, **va_list** *ap***);**
int vsscanf(const char * *str*, **const char *** *format*, **va_list** *ap***);**
int vfscanf(FILE * *stream*, **const char *** *format*, **va_list** *ap***);**

DESCRIPTION

The **scanf** family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The **scanf** function reads input from the standard input stream *stdin*, **fscanf** reads input from the stream pointer *stream*, and **sscanf** reads its input from the character string pointed to by *str*.

The **vfscanf** function is analogous to **vfprintf**(3) and reads input from the stream pointer *stream* using a variable argument list of pointers (see **stdarg**(3). The **vscanf** function scans a variable argument list from the standard input and the **vsscanf** function scans it from a string; these are analogous to the **vprintf** and **vsprintf** functions respectively.

Each successive *pointer* argument must correspond properly with each successive conversion specifier (but see 'suppression' below). All conversions are introduced by the **%** (percent sign) character. The *format* string may also contain other characters. White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the **%** character introducing a conversion there may be a number of *flag* characters, as follows:

***** Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

a Indicates that the conversion will be **s**, the needed memory space for the string will be malloc'ed and the pointer to it will be assigned to the *char* pointer variable, which does not have to be initialised before. This flag does not exist in *ANSI C*.

h Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *short int* (rather than *int*).

l Indicates either that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long int* (rather than *int*), or that the conversion will be one of **efg** and the next pointer is a pointer to *double* (rather than *float*). Specifying two **l** flags is equivalent to the **L** flag.

L Indicates that the conversion will be either **efg** and the next pointer is a pointer to *long double* or the conversion will be **dioux** and the next pointer is a pointer to *long long*. (Note that long long is not an *ANSI C* type. Any program using this will not be portable to all architectures).

q equivalent to L. This flag does not exist in *ANSI C*.

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the **%** and the conversion. If no width is given, a default of 'infinity' is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

% Matches a literal '%'. That is, '%%' in the format string matches a single input '%' character. No conversion is done, and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

D Equivalent to **ld**; this exists only for backwards compatibility. (Note: thus only in libc4. In libc5 and glibc the %D is silently ignored, causing old programs to fail mysteriously.)

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with '0x' or '0X', in base 8 if it begins with '0', and in base 10 otherwise. Only characters that correspond to the base are used.

o Matches an unsigned octal integer; the next pointer must be a pointer to *unsigned int*.

u Matches an unsigned decimal integer; the next pointer must be a pointer to *unsigned int*.

x Matches an unsigned hexadecimal integer; the next pointer must be a pointer to *unsigned int*.

X Equivalent to **x**

f Matches an optionally signed floating-point number; the next pointer must be a pointer to *float*.

e Equivalent to **f**.

g Equivalent to **f**.

E Equivalent to **f**

s Matches a sequence of non-white-space characters; the next pointer must be a pointer to *char*, and the array must be large enough to accept all the sequence and the terminating **NUL** character. The input string stops at white space or at the maximum field width, whichever occurs first.

c Matches a sequence of *width* count characters (default 1); the next pointer must be a pointer to *char*, and there must be enough room for all the characters (no terminating **NUL** is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

[Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to *char*, and there must be enough room for all the characters in the string, plus a terminating **NUL** character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket [character and a close bracket] character. The set *excludes* those characters if the first character after the open bracket is a circumflex ^. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character **-** is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, '[^]0-9-]' means the set 'everything except close bracket, zero through nine, and hyphen'. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

p Matches a pointer value (as printed by '%p' in **printf**(3); the next pointer must be a pointer to *void*.

n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to *int*. This is *not* a conversion, although it can be suppressed with the ***** flag. The C standard says: 'Execution of a

%n directive does not increment the assignment count returned at the completion of execution' but the Corrigendum seems to contradict this. Probably it is wise not to make any assumptions on the effect of %n conversions on the return value.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '%d' conversion. The value **EOF** is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

strtol(3), **strtoul**(3), **strtod**(3), **getc**(3), **printf**(3)

STANDARDS

The functions **fscanf**, **scanf**, and **sscanf** conform to ANSI X3.159-1989 ("ANSI C").

The **q** flag is the *BSD 4.4* notation for *long long*, while **ll** or the usage of **L** in integer conversions is the GNU notation.

The Linux version of these functions is based on the *GNU libio* library. Take a look at the *info* documentation of *GNU libc (glibc-1.08)* for a more concise description.

BUGS

All functions are fully ANSI X3.159-1989 conformant, but provide the additional flags **q** and **a** as well as an additional behaviour of the **L** and **l** flags. The latter may be considered to be a bug, as it changes the behaviour of flags defined in ANSI X3.159-1989.

Some combinations of flags defined by *ANSI C* are not making sense in *ANSI C* (e.g. **%Ld**). While they may have a well-defined behaviour on Linux, this need not to be so on other architectures. Therefore it usually is better to use flags that are not defined by *ANSI C* at all, i.e. use **q** instead of **L** in combination with **diouxX** conversions or **ll**.

The usage of **q** is not the same as on *BSD 4.4*, as it may be used in float conversions equivalently to **L**.