NAME

curl_maprintf, curl_mfprintf, curl_mprintf, curl_msnprintf, curl_msprintf curl_mvaprintf, curl_mvfprintf, curl_mvprintf, curl_mvsnprintf, curl_mvsprintf - formatted output conversion

SYNOPSIS

```
#include <curl/mprintf.h>

int curl_mprintf(const char *format, ...);
int curl_mfprintf(FILE *fd, const char *format, ...);
int curl_msprintf(char *buffer, const char *format, ...);
int curl_msnprintf(char *buffer, size_t maxlength, const char *format, ...);
int curl_mvprintf(const char *format, va_list args);
int curl_mvfprintf(FILE *fd, const char *format, va_list args);
int curl_mvsprintf(char *buffer, const char *format, va_list args);
int curl_mvsnprintf(char *buffer, size_t maxlength, const char *format,
                    va_list args);
char *curl_maprintf(const char *format , ...);
char *curl_mvaprintf(const char *format, va_list args);
```

DESCRIPTION

These functions produce output according to the format string and given arguments. They are mostly clones of the well-known C-style functions but there are slight differences in behavior.

We discourage users from using any of these functions in new applications.

Functions in the curl_mprintf() family produce output according to a format as described below. The functions **curl_mprintf()** and **curl_mvprintf()** write output to stdout, the standard output stream; **curl_mfprintf()** and **curl_mvfprintf()** write output to the given output stream; **curl_msprintf()**, **curl_msnprintf()**, **curl_mvsprintf()**, and **curl_mvsnprintf()** write to the character string **buffer**.

The functions **curl_msnprintf()** and **curl_mvsnprintf()** write at most *maxlength* bytes (including the terminating null byte ('\0')) to *buffer*.

The functions **curl_mvprintf()**, **curl_mvfprintf()**, **curl_mvsprintf()**, **curl_mvsnprintf()** are equivalent to the functions **curl_mprintf()**, **curl_mfprintf()**, **curl_msprintf()**, **curl_msnprintf()**, respectively, except that they are called with a *va_list* instead of a variable number of arguments. These functions do not call the *va_end* macro. Because they invoke the *va_arg* macro, the value of *ap* is undefined after the call.

The functions **curl_maprintf()** and **curl_mvaprintf()** return the output string as pointer to a newly

allocated memory area. The returned string must be *curl_free(3)*ed by the receiver.

All of these functions write the output under the control of a format string that specifies how subsequent arguments are converted for output.

FORMAT STRING

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %, and ends with a conversion specifier. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The $ modifier

The arguments must correspond properly with the conversion specifier. By default, the arguments are used in the order given, where each ’*’ (see Field width and Precision below) and each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given). One can also specify explicitly which argument is taken, at each place where an argument is required, by writing "%m$" instead of ’%’ and "*m$" instead of ’*’, where the decimal integer m denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

```
curl_mprintf("%*d", width, num);
```

and

```
curl_mprintf("%2$*1$d", width, num);
```

are equivalent. The second style allows repeated references to the same argument.

If the style using ’$’ is used, it must be used throughout for all conversions taking an argument and all width and precision arguments, but it may be mixed with "%%" formats, which do not consume an argument. There may be no gaps in the numbers of arguments specified using ’$’; for example, if arguments 1 and 3 are specified, argument 2 must also be specified somewhere in the format string.

Flag characters

The character % is followed by zero or more of the following flags:

The value should be converted to its "alternate form".

0 The value should be zero padded.

- The converted value is to be left adjusted on the field boundary. (The default is right justification.)

The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A ’-’ overrides a ’0’ if both are given.

’ ’ (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

+ A sign (+ or -) should always be placed before a number produced by a signed conversion. By default, a sign is used only for negative numbers. A ’+’ overrides a space if both are used.

Field width

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it gets padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write "*" or "*m$" (for some decimal integer m) to specify that the field width is given in the next argument, or in the *m-th* argument, respectively, which must be of type int. A negative field width is taken as a ’-’ flag followed by a positive field width. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Precision

An optional precision in the form of a period (’.’) followed by an optional decimal digit string. Instead of a decimal digit string one may write "*" or "*m$" (for some decimal integer m) to specify that the precision is given in the next argument, or in the *m-th* argument, respectively, which must be of type int. If the precision is given as just ’.’, the precision is taken to be zero. A negative precision is taken as if the precision were omitted. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the radix character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** and **S** conversions.

Length modifier

h A following integer conversion corresponds to a *short* or *unsigned short* argument.

l (ell) A following integer conversion corresponds to a *long* or *unsigned long* argument, or a following n conversion corresponds to a pointer to a long argument

ll (ell-ell). A following integer conversion corresponds to a *long long* or *unsigned long long* argument, or a following n conversion corresponds to a pointer to a long long argument.

q A synonym for **ll**.

L A following a, A, e, E, f, F, g, or G conversion corresponds to a long double argument.

z A following integer conversion corresponds to a *size_t* or *ssize_t* argument.

Conversion specifiers

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, i The int argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

o, u, x, X

The unsigned int argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (**x** and **X**) notation. The letters *abcdef* are used for **x** conversions; the letters *ABCDEF* are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

e, E

The double argument is rounded and output in the style **"[-]d.ddde+-dd"**

f, F The double argument is rounded and output to decimal notation in the style **"[-]ddd.ddd"**.

g, G

The double argument is converted in style f or e.

c The int argument is converted to an unsigned char, and the resulting character is written.

s The *const char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte. If a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

p The *void* * pointer argument is printed in hexadecimal.

n The number of characters written so far is stored into the integer pointed to by the corresponding argument.

% A '%' is written. No argument is converted.

EXAMPLE

```
const char *name = "John";

int main(void)
{
  curl_mprintf("My name is %s\n", name);
  curl_mprintf("Pi is almost %f\n", (double)25.0/8);
}
```

AVAILABILITY

These functions might be removed from the public libcurl API in the future. Do not use them in new programs or projects.

RETURN VALUE

The **curl_maprintf** and **curl_mvaprintf** functions return a pointer to a newly allocated string, or NULL if it failed.

All other functions return the number of characters actually printed (excluding the null byte used to end output to strings). Note that this sometimes differ from how the POSIX versions of these functions work.

SEE ALSO

printf(3), **sprintf**(3), **fprintf**(3), **vprintf**(3)