NAME

format() − generates formatted output as in printf

SYNOPSIS

int format(cfun, farg, formatstr, formatargs)
int (*cfun)();
void *farg;
char *formatstr;
va_list formatargs;

DESCRIPTION

format() generates ASCII data, based on a specified input string. format() knows how to create ASCII representations of all C data types.

This subroutine takes four arguments:

cfun is the address of a function declared in C as

int cfun(ch, farg)
char ch;
void *farg;

cfun is called with each character generated by format(). farg is an arbitrary value passed as the second argument to cfun.

formatstr
is a format string that controls the output, as described below.

formatargs
points to a list of arguments that are to be formatted.

Formatting Options

The format string contains a string to be processed and places where other strings can be inserted. These places are denoted by a % followed by characters which specify how the next argument from formatargs is to be placed in the string. For example, %c inserts a character. The format string "I am printing the letter %c." with the character ’f’ results in the string, "I am printing the letter f."

The simplest formatting options are as follows:

%c	outputs byte as a character
%d	outputs integer as a decimal number
%i	outputs integer as a decimal number
%D	outputs long as a decimal number
%o	outputs integer as an octal number
%O	outputs long as an octal number
%u	outputs unsigned integer as a decimal number
%x	outputs integer as a hexadecimal number
%X	outputs long as a hexadecimal number
%l	outputs long decimal
%ld	outputs long decimal
%li	outputs long decimal
%lo	outputs long octal
%lu	outputs unsigned long as a decimal number

%lx outputs long hexadecimal

For more specialized formatting, use the portable set. The syntax for the portable set is:

%<unsigned><intype><outtype>

<unsigned>, <intype>, and *<outtype>* are one character options, as follows:

Option	**Character**	**Means**
<unsigned>	U	unsigned
	none	signed
<intype>	C	char
	I	int
	S	short
	L	long
<outtype>	O	octal
	X	hexadecimal
	D	decimal

<intype> indicates the type of argument, while *<outtype>* indicates the type of format you want *<intype>* printed in.

For example, use %CO to output a character in octal or %US to output an unsigned short in decimal.

Note that in *The C Programming Language,* Kernighan and Ritchie document %x and %d; in the portable set, %IX performs the same function as %x, while %ID performs the same function as %d.

Using U for the *<unsigned>* option is only useful when D is used as the *<outtype>* option because octal and hexadecimal numbers are never signed.

Specifying Field Width

Any type of formatting option can also have a number which further specifies the formatting. The number appears after the percent sign and before any characters. The number is of the form *y.z* and can be negative.

y specifies the minimum number of characters that the formatted argument will fill. If the number is positive, the result will be right justified. If it is negative, the result will be left justified. This only specifies a minimum, so if there are more characters than can fit into *y* characters, the rest will be printed. Thus, "%6c" with the character 'f' results in the following:

```
"%6c"   "     f"
"%-6c"  "f     "
```

A leading 0 in the *y* specification causes zero filling for the right justification, so 3 printed in %06d yields "000003". The zero does not mean octal.

z specifies a maximum number of characters or significant digits and is not used for the above types because they are all fixed length. The *z* is, however, used in %s, %b, %e, %f, and %g formats.

A field width or the significant digits or both may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or the significant digits. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or the significant digits must appear *before* the *arg* (if any) to be converted. A negative field width argument is taken as a '–' flag followed by a positive field width. If the significant digits argument is negative, it will be changed to zero.

%s Inserts a string.

Useful for printing strings that have no null terminator. The minimum width of the string is *y* and the maximum width of the string is *z.*

NOTE: When string formatting reaches a null, the formatting stops. If the *y* field has specified more characters, the result is padded with blanks. Thus, formatting "test" results in the following:

```
"%6s"   "  test"
```

"%-6s"	**"test "**
"%.2"	**"te"**
"%6.2"	**" te"**

%b Inserts a string bounded by a length.

This actually uses TWO arguments: the first is a string, and the second is a number. For instance:

"%b", x, 3 is the same as **"%.3s", x**

The number argument is used as the maximum number of characters in the string. %b is used when the number of characters used might change with different calls.

%e Inserts a floating number in 'e format'.
This is of the form "˜.####e*&&":

˜	either a minus sign or nothing
#'s	the digits of the mantissa
*	either plus or minus
&'s	the digits of the exponent

Thus, 43.5 is formatted as .435e+02 and −.00435 is formatted as −.435e-02. There are always two digits of exponent. The *y* field gives a minimum width for the entire number, and the *z* field gives the maximum number of digits in the mantissa. *z* defaults to 6. Thus, formatting 43.5 results in the following:

"%10e"	**" .435e+02"**
"%-10e"	**".435e+02 "**
"%5e"	**".435e+02"**
"%10.1e"	**" .4e+02"**

%E Inserts a floating number in 'E format'.
This is of the form "˜.####E*&&", and identical to the **%e** format except that **E** is used instead of **e** in the exponent.

%f Inserts a floating number in 'f format'.
This is of the form "˜####.&&":

˜	either a minus sign or nothing
#'s	the digits before the decimal point
&'s	the digits after the decimal point

The number of #'s is always the number needed. Again, the *y* field gives a minimum on the size of the entire number, while the *z* field is the number of significant digits to keep after the decimal point. *z* defaults to 6. Thus, formatting 1.23456 results in the following:

"%10f"	**" 1.234560"**
"%-10f"	**"1.234560 "**
"%.2f"	**"1.23"**
"%5.2f"	**" 1.23"**

%F Inserts a floating number in 'F format'.
This is mainly the same as the **%f** format.

%g Inserts a floating number in optional format.

Prints in either %e or %f format, whichever is shorter, with the minimum number of digits needed to the right of the decimal. Whole numbers will have no digits to the right of the decimal because trailing zeroes are suppressed. *z* gives the maximum number of significant digits including digits before the decimal point. *z* defaults to 6. For example, the format string "%10.4g" results in the following:

435.0	**" 435."**

```
4.35        "    4.35"
.0435       " .435e-01"
.000435     " .435e-03"
```

%G Inserts a floating number in 'G format'.
This outputs the form "~.####E*&&" in case that the exponent variant is used.

%r Recursive or remote format.

This takes two arguments: the first is a format string, and the second is a pointer to the argument list. The new format string is formatted and put out in place of the %r.

The field width and significant digit specification is ignored within the %r format. The recursive format makes it possible to write variable arg printing routines without using format(3) directly.

%% Inserts %.

A double percent sign in the format string becomes a single percent sign in the output. This does not use an argument.

%<SP>

Inserts spaces.

A space character as a format control causes a field of *y* spaces to be output. This does not use an argument. Note that an actual space should appear where <SP> appears above.

RETURNS

the number of characters transmitted excluding the trailing null byte.

EXAMPLES

1. **fprintf**() (formatted print to a file) could be implemented this way:

```
#include <stdio.h>
#include <varargs.h>
/* VARARGS2 */
fprintf(f, fmt, va_alist)
        FILE    *f;
        char    *fmt;
        va_dcl
{
        va_list args;
        extern int fputc();

        va_start(args);
        format(&fputc, (long)f, fmt, args);
        va_end(args);
}
```

2. **comerr()** could be implemented this way:

```
#include <stdio.h>
#include <varargs.h>
/* VARARGS1 */
comerr(fmt, va_alist)
        char    *fmt;
        va_dcl
{
        va_list args;
        int     err;
        char    errbuf[20];
        char    *errnam;
        extern int      errno;
```

```
                extern int          sys_nerr;
                extern char         *sys_errlist[];

                err = errno;
                va_start(args);
                if (err < 0) {
                            fprintf(stderr, "Progname: %r", fmt, args);
                } else {
                            if (err >= sys_nerr) {
                                        sprintf(errbuf, "Error %d", err);
                                        errnam = errbuf;
                            } else {
                                        errnam = sys_errlist[err];
                            }
                            fprintf(stderr, "Progname: %s. %r",
                                                                errnam, fmt, args);
                }
                va_end(args);
                exit(err);
      }
```

SEE ALSO

comerr(3), error(3), fprintf(3), printf(3), sprintf(3)

NOTES

If there are no floating point operations anywhere in the program, the floating point package is not loaded and therefore floating point printing commands will possibly not work correctly.

To just print a string without any formatting, use the %s format to ensure that the string is correctly interpreted. This is especially important if the string itself contains % characters, because format() will attempt to interpret them, which causes unwanted additions in the output string. The number of characters inserted for either %s or %b should be less than 512.