The printf command

Fix Me! Stranger, this is a very big topic that needs experience - please fill in missing information, extend the descriptions, and correct the details if you can!

Attention:
This is about the Bash-builtin command printf - however, the description should be nearly identical for an external command that follows POSIX®.

GNU Awk expects a comma after the format string and between each of the arguments of a **printf** command. For examples, see: **code snippet**.

Unlike other documentations, I don't want to redirect you to the manual page for the printf() C function family. However, if you're more experienced, that should be the most detailed description for the format strings and modifiers.

Due to conflicting historical implementations of the echo command, POSIX® recommends that printf is preferred over echo.

General

The printf command provides a method to print preformatted text similar to the printf() system interface (C function). It's meant as successor for echo and has far more features and possibilities.

Beside other reasons, POSIX® has a very good argument to recommend it: Both historical main flavours of the echo command are mutual exclusive, they collide. A "new" command had to be invented to solve the issue.

Syntax

```
printf <FORMAT> <ARGUMENTS...>
```

The text format is given in <FORMAT>, while all arguments the formatstring may point to are given after that, here, indicated by <ARGUMENTS…>.

Thus, a typical printf-call looks like:

```
printf "Surname: %s\nName: %s\n" "$SURNAME" "$FIRSTNAME"
```

where "Surname: %s\nName: %s\n" is the format specification, and the two variables are passed as arguments, the %s in the formatstring points to (for every format specifier you give, printf awaits one argument!).

Options

-v VAR	If given, the output is assigned to the variable VAR instead of printed to stdout (comparable to sprintf() in some way)

The -v Option can't assign directly to array indexes in Bash versions older than Bash 4.1.

In versions newer than 4.1, one must be careful when performing expansions into the first non-option argument of printf as this opens up the possibility of an easy code injection vulnerability.

```
$ var='-vx[$(echo hi >&2)]'; printf "$var" hi; declare -p x
hi
declare -a x='([0]="hi")'
```

…where the echo can of course be replaced with any arbitrary command. If you must, either specify a hard-coded format string or use -- to signal the end of options. The exact same issue also applies to read, and a similar one to mapfile, though performing expansions into their arguments is less common.

Arguments

Of course in shell-meaning the arguments are just strings, however, the common C-notations plus some additions for number-constants are recognized to give a number-argument to printf:

Number-Format	Description
N	A normal decimal number
0N	An octal number
0xN	A hexadecimal number
0XN	A hexadecimal number
"X	(a literal double-quote infront of a character): interpreted as number (underlying codeset) **don't forget escaping**
'X	(a literal single-quote infront of a character): interpreted as number (underlying codeset) **don't forget escaping**

If more arguments than format specifiers are present, then the format string is re-used until the last argument is interpreted. If fewer format specifiers than arguments are present, then number-formats are set to zero, while string-formats are set to null (empty).

Take care to avoid word splitting, as accidentally passing the wrong number of arguments can produce wildly different and unexpected results. See this article.

Again, attention: When a numerical format expects a number, the internal printf-command will use the common Bash arithmetic rules regarding the base. A command like the following example **will** throw an error, since 08 is not a valid octal number (00 to 07!):

```
printf '%d\n' 08
```

Format strings

The format string interpretion is derived from the C printf() function family. Only format specifiers that end in one of the letters diouxXfeEgGaAcs are recognized.

To print a literal % (percent-sign), use %% in the format string.

Again: Every format specifier expects an associated argument provided!

These specifiers have different names, depending who you ask. But they all mean the same: A placeholder for data with a specified format:

- format placeholder
- conversion specification
- formatting token
- …

Format	Description
%b	Print the associated argument while interpreting backslash escapes in there
%q	Print the associated argument **shell-quoted**, reusable as input
%d	Print the associated argument as **signed decimal** number
%i	Same as %d
%o	Print the associated argument as **unsigned octal** number
%u	Print the associated argument as **unsigned decimal** number
%x	Print the associated argument as **unsigned hexadecimal** number with lower-case hex-digits (a-f)
%X	Same as %x, but with upper-case hex-digits (A-F)
%f	Interpret and print the associated argument as **floating point** number
%e	Interpret the associated argument as **double**, and print it in <N>±e<N> format
%E	Same as %e, but with an upper-case E in the printed format
%g	Interprets the associated argument as **double**, but prints it like %f or %e
%G	Same as %g, but print it like %E
%c	Interprets the associated argument as **char**: only the first character of a given argument is printed
%s	Interprets the associated argument literally as string
%n	Assigns the number of characters printed so far to the variable named in the corresponding argument. Can't specify an array index. If the given name is already an array, the value is assigned to the zeroth element.
%a	Interprets the associated argument as **double**, and prints it in the form of a C99 hexadecimal floating-point literal.
%A	Same as %a, but print it like %E
%(FORMAT)T	output the date-time string resulting from using FORMAT as a format string for strftime(3). The associated argument is the number of seconds since Epoch, or -1 (current time) or -2 (shell startup time). If no corresponding argument is supplies, the current time is used as default
%%	No conversion is done. Produces a % (percent sign)

Some of the mentioned format specifiers can modify their behaviour by getting a format modifier:

Modifiers

To be more flexible in the output of numbers and strings, the printf command allows format modifiers. These are specified **between** the introductory % and the character that specifies the format:

```
printf "%50s\n" "This field is 50 characters wide..."
```

Field and printing modifiers

Field output format	
<N>	**Any number**: Specifies a **minimum field width**, if the text to print is shorter, it's padded with spaces, if the text is longer, the field is expanded
.	**The dot**: Together with a field width, the field is **not** expanded when the text is longer, the text is truncated instead. "%.s" is an undocumented equivalent for "%.0s", which will force a field width of zero, effectively hiding the field from output
*	**The asterisk**: the width is given as argument before the string or number. Usage (the "*" corresponds to the "20"): printf "%*s\n" 20 "test string"
#	"Alternative format" for numbers: see table below
-	**Left-bound** text printing in the field (standard is **right-bound**)
0	Pads numbers with zeros, not spaces
<space>	Pad a positive number with a space, where a minus (-) is for negative numbers
+	Prints all numbers **signed** (+ for positive, - for negative)
'	For decimal conversions, the thousands grouping separator is applied to the integer portion of the output according to the current LC_NUMERIC

The "alternative format" modifier #:

Alternative Format	
%#o	The octal number is printed with a leading zero, unless it's zero itself
%#x, %#X	The hex number is printed with a leading "0x"/"0X", unless it's zero
%#g, %#G	The float number is printed with **trailing zeros** until the number of digits for the current precision is reached (usually trailing zeros are not printed)
all number formats except %d, %o, %x, %X	Always print a decimal point in the output, even if no digits follow it

Precision

The precision for a floating- or double-number can be specified by using .<DIGITS>, where <DIGITS> is the number of digits for precision. If <DIGITS> is an asterisk (*), the precision is read from the argument that precedes the number to print, like (prints 4,3000000000):

```
printf "%.*f\n" 10 4,3
```

The format .*N to specify the N'th argument for precision does not work in Bash.

For strings, the precision specifies the maximum number of characters to print (i.e., the maximum field width). For integers, it specifies the number of digits to print (zero-padding!).

Escape codes

These are interpreted if used anywhere in the format string, or in an argument corresponding to a %b format.

Code	Description
\\	Prints the character \ (backslash)
\a	Prints the alert character (ASCII code 7 decimal)
\b	Prints a backspace
\f	Prints a form-feed
\n	Prints a newline
\r	Prints a carriage-return
\t	Prints a horizontal tabulator
\v	Prints a vertical tabulator
\"	Prints a '
\?	Prints a ?
\<NNN>	Interprets <NNN> as **octal** number and prints the corresponding character from the character set
\0<NNN>	same as \<NNN>
\x<NNN>	Interprets <NNN> as **hexadecimal** number and prints the corresponding character from the character set (**3 digits**)
\u<NNNN>	same as \x<NNN>, but **4 digits**
\U<NNNNNNNN>	same as \x<NNN>, but **8 digits**

The following additional escape and extra rules apply only to arguments associated with a %b format:

\c	Terminate output similarly to the \c escape used by echo -e. printf produces no additional output after coming across a \c escape in a %b argument.

- Backslashes in the escapes: \', \", and \? are not removed.
- Octal escapes beginning with \0 may contain up to four digits. (POSIX specifies up to three).

These are also respects in which %b differs from the escapes used by $'...' style quoting.

Examples

Snipplets

- print the decimal representation of a hexadecimal number (preserve the sign)
  - `printf "%d\n" 0x41`
  - `printf "%d\n" -0x41`
  - `printf "%+d\n" 0x41`
- print the octal representation of a decimal number
  - `printf "%o\n" 65`
  - `printf "%05o\n" 65` (5 characters width, padded with zeros)
- this prints a 0, since no argument is specified
  - `printf "%d\n"`
- print the code number of the character A
  - `printf "%d\n" \'A`
  - `printf "%d\n" "'A"`
- Generate a greeting banner and assign it to the variable GREETER
  - `printf -v GREETER "Hello %s" "$LOGNAME"`
- Print a text at the end of the line, using tput to get the current line width
  - `printf "%*s\n" $(tput cols) "Hello world!"`

Small code table

This small loop prints all numbers from 0 to 127 in

- decimal
- octal
- hex

```
for ((x=0; x <= 127; x++)); do
  printf '%3d | %04o | 0x%02x\n' "$x" "$x" "$x"
done
```

Ensure well-formatted MAC address

This code here will take a common MAC address and rewrite it into a well-known format (regarding leading zeros or upper/lowercase of the hex digits, …):

```
the_mac="0:13:ce:7:7a:ad"

# lowercase hex digits
the_mac="$(printf "%02x:%02x:%02x:%02x:%02x:%02x" 0x${the_mac//:/ 0x})"

# or the uppercase-digits variant
the_mac="$(printf "%02X:%02X:%02X:%02X:%02X:%02X" 0x${the_mac//:/ 0x})"
```

Replacement echo

This code was found in Solaris manpage for echo(1).

Solaris version of /usr/bin/echo is equivalent to:

```
printf "%b\n" "$*"
```

Solaris /usr/ucb/echo is equivalent to:

```
if [ "X$1" = "X-n" ]
then
     shift
     printf "%s" "$*"
else
     printf "%s\n" "$*"
fi
```

prargs Implementation

Working off the replacement echo, here is a terse implementation of prargs:

```
printf '"%b"\n' "$0" "$@" | nl -v0 -s": "
```

repeating a character (for example to print a line)

A small trick: Combining printf and parameter expansion to draw a line

```
length=40
printf -v line '%*s' "$length"
echo ${line// /-}
```

or:

```
length=40
eval printf -v line '%.0s-' {1..$length}
```

Replacement for some calls to date(1)

The %(…)T format string is a direct interface to strftime(3).

```
$ printf 'This is week %(%U/%Y)T.\n' -1
This is week 52/2010.
```

Please read the manpage of strftime(3) to get more information about the supported formats.

differences from awk printf

Awk also derives its *printf()* function from C, and therefore has similar format specifiers. However, in all versions of awk the space character is used as a string concatenation operator, so it cannot be used as an argument separator. **Arguments to awk printf must be separated by commas.** Some versions of awk do not require printf arguments to be surrounded by parentheses, but you should use them anyway to provide portability.

In the following example, the two strings are concatenated by the intervening space so that no argument remains to fill the format.

```
$ echo "Foo" | awk '{ printf "%s\n" $1 }'
awk: (FILENAME=- FNR=1) fatal: not enough arguments to satisfy format string
        `%s
Foo'
         ^ ran out for this one
```

Simply replacing the space with a comma and adding parentheses yields correct awk syntax.

```
$ echo "Foo" | awk '{ printf( "%s\n", $1 ) }'
Foo
```

With appropriate metacharacter escaping the bash printf can be called from inside awk (as from perl and other languages that support shell callout) as long as you don't care about program efficiency or readability.

```
echo "Foo" | awk '{ system( "printf \"%s\\n \" \"" $1 "\""  ) }'
Foo
```

Differences from C, and portability considerations

- The a, A, e, E, f, F, g, and G conversions are supported by Bash, but not required by POSIX.
- There is no wide-character support (wprintf). For instance, if you use %c, you're actually asking for the first byte of the argument. Likewise, the maximum field width modifier (dot) in combination with %s goes by bytes, not characters. This limits some of printf's functionality to working with ascii only. ksh93's printf supports the L modifier with %s and %c (but so far not %S or %C) in order to treat precision as character width, not byte count. zsh appears to adjust itself dynamically based upon LANG and LC_CTYPE. If LC_CTYPE=C, zsh will throw "character not in range" errors, and otherwise supports wide characters automatically if a variable-width encoding is set for the current locale.
- Bash recognizes and skips over any characters present in the length modifiers specified by POSIX during format string parsing.

builtins/printf.def

```
#define LENMODS "hjlLtz"
...
/* skip possible format modifiers */
modstart = fmt;
while (*fmt && strchr (LENMODS, *fmt))
fmt++;
```

- mksh has no built-in printf by default (usually). There is an unsupported compile-time option to include a very poor, basically unusable implementation. For the most part you must rely upon the system's /usr/bin/printf or equivalent. The mksh maintainer recommends using print. The development version (post- R40f) adds a new parameter expansion in the form of ${name@Q} which fills the role of printf %q – expanding in a shell-escaped format.
- ksh93 optimizes builtins run from within a command substitution and which have no redirections to run in the shell's process. Therefore the printf -v functionality can be closely matched by var=$(printf …) without a big performance hit.

```
# Illustrates Bash-like behavior. Redefining printf is usually unnecessary / not recommended.
function printf {
    case $1 in
        -v)
            shift
            nameref x=$1
            shift
            x=$(command printf "$@")
            ;;
        *)
            command printf "$@"
    esac
}
builtin cut
print $$
printf -v 'foo[2]' '%d\n' "$(cut -d ' ' -f 1 /proc/self/stat)"
typeset -p foo
# 22461
# typeset -a foo=([2]=22461)
```

- The optional Bash loadable print may be useful for ksh compatibility and to overcome some of echo's portability pitfalls. Bash, ksh93, and zsh's print have an -f option which takes a printf format string and applies it to the remaining arguments. Bash lists the synopsis as: print: print [-Rnprs] [-u unit] [-f format] [arguments]. However, only -Rrnfu are actually functional. Internally, -p is a noop (it doesn't tie in with Bash coprocs at all), and -s only sets a flag but has no effect. -Cev are unimplemented.
- Assigning to variables: The printf -v way is slightly different to the way using command-substitution. Command substitution removes trailing newlines before substituting the text, printf -v preserves all output.

See also

- SUS: printf utility and printf() function
- Code snip: Print a horizontal line uses some printf examples
- Greg's BashFAQ 18: How can I use numbers with leading zeros in a loop, e.g., 01, 02?

Discussion

Dan Douglas, 2011/09/10 03:36

A somewhat glaring omission from every shell I have to test with other than ksh93 is the numbered argument conversion specifiers. %n$ or *n$ http://pubs.opengroup.org/onlinepubs/9699919799/functions/printf.html

The idea is to allow either rearranging the order of the arguments, or reusing a width modifier (with *), by addressing which conversions apply to which args. Unfortunately these don't really save you any typing because numbering any of the conversions causes the behavior when there are more arguments than conversions to go away. The spec says behavior is unspecified in this case. Ksh simply segfaults (as the 3rd example below). The first and second are essentially equivalent.

```
~ $ printf '%.*s\n' 3 'foobar' 3 'foobarbaz' #Bash
foo
foo
~ $ ksh
$ printf '%.*1$s\n%3$.*1$s\n' 3 'foobar' 'foobarbaz'
foo
foo
$ printf '%.*1$s\n' 3 'foobar' 'foobarbaz'
Segmentation fault
~ $ printf '%.*1$s\n%3$.*1$s\n' 3 'foobar' 'foobarbaz' #Back in Bash
-bash: printf: `1': invalid format character
~ $
```

Kind of dumb the way it's specified but I can imagine a few scenarios where it might be useful. And given that Bash simply points to printf(3) as documentation it's a rather odd thing to miss even though it's also missing in almost every other shell's builtins plus printf(1) of gnu coreutils. (It's described in printf(3) of the linux-manpages too, which mentions it's SUS, but not C99.)

Altair IV, 2012/05/18 14:19

Umm, what? That last section has no business being here. awk's built-in printf function is a completely different entity from the shell's version. The syntax for awk's printf is comma-delimited, i.e. 'printf("<format>" , "<arguments>")' , with the parentheses being optional. So it's actually the first "fix" that's using it correctly.

Dan Douglas, 2012/07/12 06:13

So fix it, this is a wiki. :)

MJF, 2012/07/12 12:18

> That last section has no business being here. awk's built-in printf function is a completely different entity from the shell's version.

Yeah, it's bloody obvious that 'printf' and 'printf' are "completely different" things. The fact that the syntax for the two is only slightly different is just a massive coincidence, and can't possibly lead to any confusion. And why bother to document that difference? People should just read the source code to figure out why awk is puking out errors, right?

R.W. Emerson II, 2012/12/08 18:07

I've found that the following lines produce different results:

```
  var=$(printf ...)
  printf -v var ...
```

For example, the first line below omits the trailing \n, but the second line retains it:

```
declare vT ; vT=$(printf "%s\n" "ABC") ; echo "vT($vT)"
declare vT ; printf -v vT "%s\n" "ABC" ; echo "vT($vT)"
```

It's nice to finally find a site that documents these bash commands in depth. I've been searching for hours, in vain, for an article or post that mentions the above printf bug/feature. I'm surprised to find that no one else has mentioned it.

Jan Schampera, 2012/12/16 13:19

The reason is that a command substitution $() cuts a trailing newline, as mentioned in the article about command substitution.

Thus, your notice is absolutely correct. These two commands produce slightly different results and I should mention it above.

You could leave a comment if you were logged in.