The read builtin command

read something about read here!

Synopsis

```
read [-ers] [-u <FD>] [-t <TIMEOUT>] [-p <PROMPT>] [-a <ARRAY>] [-n <
NCHARS>] [-N <NCHARS>] [-d <DELIM>] [-i <TEXT>] [<NAME...>]
```

Description

The `read` builtin reads **one line** of data (text, user input, …) from standard input or a supplied filedescriptor number into one or more variables named by `<NAME…>` .

Since Bash 4.3-alpha, `read` skips any `NUL` (ASCII () code 0) characters in input.

If `<NAME…>` is given, the line is word-split using IFS variable, and every word is assigned to one `<NAME>` . The remaining words are all assigned to the last `<NAME>` if more words than variable names are present.

> If no `<NAME>` is given, the whole line read (without performing word-splitting!) is assigned to the shell variable REPLY. Then, `REPLY` really contains the line as it was read, without stripping pre- and postfix spaces and other things!
>
> ```
> while read -r; do
>   printf '"%s"\n' "$REPLY"
> done <<<"  a line with prefix and postfix space  "
> ```

If a timeout is given, or if the shell variable TMOUT is set, it is counted from initially waiting for input until the completion of input (i.e. until the complete line is read). That means the timeout can occur during input, too.

Options

Option	Description

Option	Description
`-a <ARRAY>`	read the data word-wise into the specified array `<ARRAY>` instead of normal variables
`-d <DELIM>`	recognize `<DELIM>` as data-end, rather than `<newline>`
`-e`	on interactive shells: use Bash's readline interface to read the data. Since version 5.1-alpha, this can also be used on specified file descriptors using `-u`
`-i <STRING>`	preloads the input buffer with text from `<STRING>` , only works when Readline (`-e`) is used
`-n <NCHARS>`	reads `<NCHARS>` characters of input, then quits
`-N <NCHARS>`	reads `<NCHARS>` characters of input, *ignoring any delimiter*, then quits
`-p <PROMPT>`	the prompt string `<PROMPT>` is output (without a trailing automatic newline) before the read is performed
`-r`	raw input - **disables** interpretion of **backslash escapes** and **line-continuation** in the read data
`-s`	secure input - don't echo input if on a terminal (passwords!)
`-t <TIMEOUT>`	wait for data `<TIMEOUT>` seconds, then quit (exit code 1). Fractional seconds ("5.33") are allowed since Bash 4. A value of 0 immediately returns and indicates if data is waiting in the exit code. Timeout is indicated by an exit code greater than 128. If timeout arrives before data is read completely (before end-of-line), the partial data is saved.
`-u <FD>`	use the filedescriptor number `<FD>` rather than `stdin` (0)

When both, `-a <ARRAY>` and a variable name `<NAME>` is given, then the array is set, but not the variable.

Of course it's valid to set individual array elements without using `-a` :

```
read MYARRAY[5]
```

> Reading into array elements using the syntax above **may cause pathname expansion to occur**.
>
> Example: You are in a directory with a file named `x1` , and you want to read into an array `x` , index `1` with
>
> ```
> read x[1]
> ```
>
> then pathname expansion will expand to the filename `x1` and break your processing!

Even worse, if `nullglob` is set, your array/index will disappear.

To avoid this, either **disable pathname expansion** or **quote** the array name and index:

```
read 'x[1]'
```

Return status

Status	Reason
0	no error
0	error when assigning to a read-only variable [1)]
2	invalid option
>128	timeout (see `-t`)
!=0	invalid filedescriptor supplied to `-u`
!=0	end-of-file reached

read without -r

Essentially all you need to know about `-r` is to **ALWAYS** use it. The exact behavior you get without `-r` is completely useless even for weird purposes. It basically allows the escaping of input which matches something in IFS, and also escapes line continuations. It's explained pretty well in the POSIX read (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/read.html#tag_20_109) spec ().

```
2012-05-23 13:48:31     geirha  it should only remove the backslashe
s, not change \n and \t and such into newlines and tabs
2012-05-23 13:49:00     ormaaj  so that's what read without -r does?
2012-05-23 13:49:16     geirha  no, -r doesn't remove the backslashes
2012-05-23 13:49:34     ormaaj  I thought read <<<'str' was equivalen
t to read -r <<<$'str'
2012-05-23 13:49:38     geirha  # read x y <<< 'foo\ bar baz'; echo "
<$x><$y>"
2012-05-23 13:49:40     shbot   geirha: <foo bar><baz>
2012-05-23 13:50:32     geirha  no, read without -r is mostly pointle
ss. Damn bourne
2012-05-23 13:51:08     ormaaj  So it's mostly (entirely) used to esc
ape spaces
2012-05-23 13:51:24     ormaaj  and insert newlines
2012-05-23 13:51:47     geirha  ormaaj: you mostly get the same effec
t as using \ at the prompt
2012-05-23 13:52:04     geirha  echo \"  outputs a " ,  read x <<<
'\"'  reads a "
2012-05-23 13:52:32     ormaaj  oh weird
2012-05-23 13:52:46      *      ormaaj struggles to think of a point
to that...
2012-05-23 13:53:01     geirha  ormaaj: ask Bourne :P
2012-05-23 13:53:20     geirha  (not Jason)
2012-05-23 13:53:56     ormaaj  hm thanks anyway :)
```

Examples

Rudimentary cat replacement

A rudimentary replacement for the `cat` command: read lines of input from a file and print them on the terminal.

```
opossum() {
  while read -r; do
    printf "%s\n" "$REPLY"
  done <"$1"
}
```

Note: Here, `read -r` and the default `REPLY` is used, because we want to have the real literal line, without any mangeling. `printf` is used, because (depending on settings), `echo` may interpret some baskslash-escapes or switches (like `-n`).

Press any key...

Remember the MSDOS `pause` command? Here's something similar:

```
pause() {
  local dummy
  read -s -r -p "Press any key to continue..." -n 1 dummy
}
```

Notes:

- `-s` to suppress terminal echo (printing)
- `-r` to not interpret special characters (like waiting for a second character if somebody presses the backslash)

Reading Columns

Simple Split

Read can be used to split a string:

```
var="one two three"
read -r col1 col2 col3 <<< "$var"
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$col3"
```

Take care that you cannot use a pipe:

```
echo "$var" | read col1 col2 col3 # does not work!
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$col3"
```

Why? because the commands of the pipe run in subshells that cannot modify the parent shell. As a result, the variables `col1`, `col2` and `col3` of the parent shell are not modified (see article: Bash and the process tree).

If the variable has more fields than there are variables, the last variable get the remaining of the line:

```
read col1 col2 col3 <<< "one two three four"
printf "%s\n" "$col3" #prints three four
```

Changing The Separator

By default reads separates the line in fields using spaces or tabs. You can modify this using the *special variable* IFS, the Internal Field Separator.

```
IFS=":" read -r col1 col2 <<< "hello:world"
printf "col1: %s col2: %s\n" "$col1" "$col2"
```

Here we use the `var=value command` syntax to set the environment of `read` temporarily. We could have set `IFS` normally, but then we would have to take care to save its value and restore it afterward (`OLD=$IFS IFS=":"; read ….;IFS=$OLD`).

The default `IFS` is special in that 2 fields can be separated by one or more space or tab. When you set `IFS` to something besides whitespace (space or tab), the fields are separated by **exactly** one character:

```
IFS=":" read -r col1 col2 col3 <<< "hello::world"
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$col3"
```

See how the `::` in the middle infact defines an additional *empty field*.

The fields are separated by exactly one character, but the character can be different between each field:

```
IFS=":|@" read -r col1 col2 col3 col4 <<< "hello:world|in@bash"
printf "col1: %s col2: %s col3 %s col4 %s\n" "$col1" "$col2" "$col3"
"$col4"
```

Are you sure?

```
asksure() {
echo -n "Are you sure (Y/N)? "
while read -r -n 1 -s answer; do
  if [[ $answer = [YyNn] ]]; then
    [[ $answer = [Yy] ]] && retval=0
    [[ $answer = [Nn] ]] && retval=1
    break
  fi
done

echo # just a final linefeed, optics...

return $retval
}

### using it
if asksure; then
  echo "Okay, performing rm -rf / then, master...."
else
  echo "Pfff..."
fi
```

Ask for a path with a default value

Note: The `-i` option was introduced with Bash 4

```
read -e -p "Enter the path to the file: " -i "/usr/local/etc/" FILEPA
TH
```

The user will be prompted, he can just accept the default, or edit it.

Multichar-IFS: Parsing a simple date/time string

Here, `IFS` contains both, a colon and a space. The fields of the date/time string are recognized correctly.

```
datetime="2008:07:04 00:34:45"
IFS=": " read -r year month day hour minute second <<< "$datetime"
```

Portability considerations

- POSIX® only specified the `-r` option (raw read); `-r` is not only POSIX, you can find it in earlier Bourne source code
- POSIX® doesn't support arrays
- `REPLY` is not POSIX®, you need to set `IFS` to the empty string to get the whole line for shells that don't know `REPLY`.

```
while IFS= read -r line; do
  ...
done < text.txt
```

See also

- Internal: The printf builtin command

[1)] fixed in 4.2-rc1

Discussion

Dan Douglas, 2011/09/13 06:07 ()

There's something slightly fishy going on with how the read arguments are parsed. In reality it's arbitrary expansions.

```
$ x='['; y=']'; z='+++'; while read -r "a${x}v${z%+}$y"; do :; do
ne < <(printf '%s\n' {a..d}); echo "${a[@]}";
a b c d
```

The string is expanded, passed into read, and then eval'd. So expansions are happening twice here assuming it's double not single quoted.

Another example pre-initializing the variable in the arithmetic context:

```
$ while read -r "var[${x:=5}, x++]"; do :; done < <(printf '%s\n'
{a..c}); for i in "${!var[@]}"; do printf '(%d, %s), ' "$i" "${va
r[$i]}"; done;
(5, a), (6, b), (7, c), (8, ),
```

However, if that's all there was to it you wouldn't expect this to work:

```
while read -r "var[$((x++))]"; do ...
```

There's some unique evaluation mode going on here since Bash knows not to perform arithmetic expansion before passing the resulting string into read otherwise the result would simply be to make all assignments to var[0].

PS... the preview is deleting all the "pluses". Hopefully the above is understandable.

Jan Schampera, 2011/09/20 17:38 ()

Hi,

sorry for the delay

This is not "double-expansion" per se, the behaviour is completely understandable (but I think you know it).

Thanks for this very interesting spotlight, I never thought that much about it!

Grisha, 2015/09/04 06:45 ()

> Bash knows not to perform arithmetic expansion before passing the resulting string into read

That's because `read` is part of the `while` command, so expansions do not happen until the command is executed for each loop iteration.

Andor, 2012/06/06 14:46 ()

Read, has some gotchas.

http://www.redhat.com/mirrors/LDP/LDP/abs/html/gotchas.html (http://www.redhat.com/mirrors/LDP/LDP/abs/html/gotchas.html)

Jan Schampera, 2012/07/01 11:43 ()

Yes, it's not related directly to `read`, but to Bash's design regarding pipelining. AFAIR, there was a discussion on the mailing list to align the behaviour to what Korn does - running the last element of a pipeline in the current execution environment.

ABS is a good collection and a good work showing how to use the shell, but don't over-estimate it's technical side.

scavenger, 2013/02/18 09:49 (), 2013/12/31 09:32 ()

thank you bash for running a sub-shell when piping, now we cannot read anymore multiple variables at the same time !

```
grep -w regexp file | read a b c
```

there is no solution to replace this KSH functionality. The command: `read -r a b c «<$(command)` solution is bourne and korn shell (88 & 93) incompatible.

dannysauer, 2016/05/26 15:42 ()

Both bash and ksh93 can do process substitution like `read a b c < <(some command)` (note that there is a space between the first <, which redirects STDIN, and the second <, which indicates to run the command and connect it to a named pipe, providing the pipe in place of the `<(cmd)` expression).

You can use that in place of a pipe pretty much any time. I use it all the time for things like `diff <(sort file1 | grep -v '^#') <(sort file2 | grep -v '^#')` , which compares two files, ignoring comments and sorting the lines in the files before comparing. :)

BobD (http://ezilidanto.com), 2015/01/12 17:42 ()

scavenger, how about?–

echo a b c | read -r -a a ; echo ${a[@]} ${#a[@]}

a b c 3