Bash Tips and Tricks

News	Bash customization	Recommended Links	Unix Sysadmin Tips	Command history reuse	Customizing Shell Dot Files: .profile, RC-file, and history	Advanced filesystem navigation
Neatbash -- a simple bash prettyprinter	Midnight Commander as Bash IDE	BASH Debugging	Shell Prompts	Shell Aliases	Examples of .bashrc files	dirname and basename
.screenrc examples	Attaching to and detaching from screen sessions	How to rename files with special characters in names	Midnight Commander Tips and Tricks	WinSCP Tips	cdpath	Piping Vim Buffer Through Unix Filters: ! and !! Commands
Arithmetic Expressions in BASH	String Operations	if statements in shell	Usage of pipes with loops in shell	sort command	tr command	exec command
dd	cut command	System Activity Reporter (sar)	Shell Input and Output Redirection	Unix Find Tutorial. Using -exec option with find	Unix find tutorial Finding files using file name or path	Brace Expansion
AWK Tips	AWK one liners	GNU Screen Tips	VIM Tips	pv	Command completion	BASH Debugging
IFS	SSH Tips	SCP Tips	Readline and inputrc	Shell Input and Output Redirection	Bash Built-in Variables	Directory favorites
Brace Expansion	Process Substitution in Shell	Sequences of commands in Unix shell	Subshells	Shell scripts collections	nmap_tips	Annotated List of Bash Enhancements
Pipes in Loops	Pushd, popd and dirs	Sysadmin Horror Stories	Unix shells history	Unix Shell Tips and Tricks	Humor	Etc

See also Unix Shell Tips and Tricks

The introduction below was adapted from article "Unix Scripting: some Traps, Pitfalls and Recommendations" by Marc Dobson

Introduction

Bash has several gotchas

- **Gotcha 1** BASH has some very unintuitive behavior if you source a script and do not provide a path to it: by default, when the command "source filename" does not contain a slash (*i.e.* does not include a path for the file), BASH searches the $PATH environment variable for the filename, and ***only if it does not find one there*** it searches the current directory!!!! ***Furthermore this happens independently whether the file has the executable bit set or not*** . This is counter intuitive to say the least and as an example TCSH does NOT do this search. One can ***disable this behavior*** with the BASH "shopt" built-in command:

- shopt -u sourcepath

- **Gotcha 2: pipeline in bash by default does not write in the current shell.** That was a blatant design erroir and it was corrected only in Bash 3.0 or so. Now you can use shopt to fix that

- **Gotcha 3: Editing long line that span several lines on your terminal in many versions of bash is a pain in the neck. It discards some symbols and does other crazy staff.** Copy this line into a file, edit with vi or your favorite editor and source the file.

Recommendation 1: ***Create a special bash related log file or notebook where you write your findings.*** Environment is now so complex that you will definitely forget some of the most useful findings, if you do not write them down and periodically browse the content.

> For the same purpose create and maintain separate file with aliases (say .aliases) and a file with functions (say .functions), where you can write all the best ideas you have found or invented yourself. Which actually might never visit you the second time unless you write them the first time. Just don't overdo it, too many aliases are as bad as too few. Here excessive zeal is really destructive. But even if you do not use them resizing your .aliases and .functions file is a very useful exercise that refresh some of long forgotten skills that at one point of time you used to have ;-)

Recommendation 2: ***In order to force bash to write lines in history on exit you need to put the line***

shopt -s histappend

into your .bash_profile or a similar file (e.g. .profile) that executes for interactive sessions only.

Recommendation 3: **Add to your Prompt command** history -a to preserve history from multiple terminals. ***This is a very neat trick !!!***

> **Bash history handling with multiple terminals**
>
> The **bash** session that is saved is the one for the **terminal** that is closed the latest. If you want to save the commands for every session, you could use the trick explained here.
>
> export PROMPT_COMMAND='history -a'

To quote the manpage: “If set, the value is executed as a command prior to issuing each primary prompt.”

So every time my command has finished, it appends the unwritten **history** item to ~/.bash

ATTENTION: If you use multiple shell sessions and do not use this trick, you need to write the history manually to preserver it using the command history -a

See also:

- https://unix.stackexchange.com/questions/1288/preserve-bash-history-in-multiple-terminal-windows
- http://northernmost.org/blog/flush-bash_history-after-each-command/comment-page-1/index.html#comment-640
- Keeping persistent history in bash - Eli Bendersky's website

Recommendation 4: ls command has option -h which like in df produces "human readable" size of the file. So the most famous shell alias

```
alias ll='ls -la'
```

Might better be written as

```
alias ll='ls -halF'
```

Recommendation 5: If you prefer light color for your terminal, you are generally screwed: it is very difficult select proper colors for light background. Default colors work well on black of dark blue background, but that's it. For light background you need to limit yourself to three basic colors (black, red and blue) and forget about all other nuances. Actually they do not matter much anyway, too many colors it is just another sign of overcomplexity of the Linux environment as people simply stop paying attention to them. To disable or simplify color scheme create your own DIR_COLORS file or use option --nocolor.

Recommendation 6: When sourcing a script always use a path name for the file or at lease the prefix "./". By default Bash first searches regular names in PATH first. You can disable this behaviour with shopt -u sourcepath but it you work on multiple boxes where you are not primary administrator you can't just put this option into /etc/bashrc. Sourcing script from a wrong directory might lead to disasters/horror stories, especially, if you are working as root.

Recommendation 7: always choose a unique script name. Ther is nothing wrong with long names, if they help to prevent a SNAFU. Unique script names can easily be obtained by prefixing the name with the project name to the script name (*e.g.* gpfs_setup). Bad generic names, where multiple scripts with the same name might exist in multiple directories are for example, setup, configure, install etc...

Recommendation 8: While this page is about clever, ingenious tips and tricks, you should never try to be too clever or too bold. Always play safe and test your commands, such as find with -exec option by printing set of files they operate on before applying it to a production server (especially if this is a remote server). System administration is a very conservative profession and absence of SNAFU is more important that demonstration of excessive cleverness, boldness...

Sourcing versus Executing

In the sourced file, an EXIT command, will terminate the whole script in which it was issues (the shell that invoked this script) not just sourced sub-script where the exit command was executed.

In contrast in standalone scripts which are executed in subshell the EXIT command in this case will exit the shell/interpreter which was started to execute the "main" script. Therefore the executed script file will just stop and return to the shell which called it.

As an example take the following two scripts. Script 1 is:

```
#!/bin/bash

echo "Executing script2"
./script2
if [ $? -eq 0 ]; then
  echo "Executing ls in /tmp/md"
  ls -l /tmp/md
else
  echo "Exiting"
  exit 1
fi
```

And script 2 is:

```
#!/bin/bash

echo "In script 2"
if [ -e "/tmp/md" ]; then
  echo "/tmp/md exists"
else
  echo "/tmp/md does not exist"
  exit 1
fi
```

Both scripts should have the execute bit set. Start a BASH shell by typing bash, and at the next prompt execute script 1. The following output is produced:

If directory /tmp/md exists:

```
Executing script2
In script 2
/tmp/md exists
Executing ls in /tmp/md
total 0
```

If directory /tmp/md does not exist:

```
Executing script2
In script 2
/tmp/md does not exist
Exiting
```

Now change script 1 to source script 2 instead of executing it (source ./script2 instead of ./script2). When the script 1 is executed the following output will be produced:

If directory /tmp/md exists:

If directory /tmp/md does not exist:

```
Executing script2
In script 2
/tmp/md exists
Executing ls in /tmp/md
total 0
```

```
Executing script2
In script 2
/tmp/md does not exist
```

If the directory /tmp/md exists then the output is the same and exactly the same commands were executed. If however the directory /tmp/md does not exist then the script 2 has an EXIT and as it was sourced from script 1, it is actually script 1 which exits without the desired effect, *i.e.* printing "Exiting". In this case it is not very important but it could have very profound consequences with complex scripts.

The ambiguity in this case is compounded by the difference in coding in the two branches of the IF statement of script 2. For the case when the directory exists the EXIT command is implicit (the script goes to the end and exits normally), whereas for the case when the directory does not exist the EXIT command is explicit (this is the one which causes the exit from script 1).

If the programmer wishes to exit from a sourced script file (as he would with the EXIT command in an executed script), he may do so with:

```
return [n]
```

where "[n]" is the return value that can be tested for in the script/shell which sourced the script file (as with the EXIT command). Beware though that the RETURN command is also used to exit a function, therefore make sure that the RETURN command is placed in the appropriate place for the desired effect.

Do not use source functionality as a poor man subroutines

If the same functionality is required (*i.e.* the same commands) to be executed multiple times it is better to use shell functions or standalone scripts, then to source the same fragment multiple times. If you use this "multiple sourcing" as a poor man subroutine always put a banner to remind yourself what is happening:

```
#!/bin/bash

echo "We have been executed"
echo "Sourcing the external commands from the file /root/bin/standard_gpfs_setup_actions..."
. /root/bin/standard_gpfs_setup_actions
echo "Exiting"
```

If the set of commands is written as a file that needs to be sourced use the full path or least specify "dot-slash prefix if it reside in the current directory. Never use "naked", non-qualified names. For example

```
. ./sourced_script
```


Top Visited

- [May 01, 2022] **Films depicting female sociopaths** : **Understanding Micromanagers and Control Freaks**
- [Apr 30, 2022] **Insufficient Retirement Funds as Immanent Problem of Neoliberal Regime**
- [Apr 29, 2022] **The Mythical Man-Month**
- [Apr 28, 2022] **TAoCP and its Influence of Computer Science** : **The second love -- typography** : **Literate programming**
- [Apr 27, 2022] **The AWK Programming Language**
- [Apr 26, 2022] **Slightly Skeptical View on John K. Ousterhout and Tcl**
- [Apr 25, 2022] **Slightly Skeptical View on Larry Wall and Perl**
- [Apr 24, 2022] **Donald Knuth**
- [Apr 23, 2022] **Financial Humor** : **Politically Incorrect Humor** : **Computer and System Administration Humor**
- [Apr 22, 2022] **Introduction to the Unix shell history**
- [Apr 21, 2022] **Unix History with some Emphasis on Scripting** :
- [Apr 20, 2022] **A Slightly Skeptical View on Linus Torvalds** : **Selling Bazaar to Cathedral: Linux Gold Rush**
- [Apr 19, 2022] **Multix OS**
- [Apr 18, 2022] **Programming Languages History**
- [Apr 17, 2022] **Fifty glorious years (1950-2000): the triumph of the US computer engineering**
- [Apr 16, 2022] **Financial Humor** : **Politically Incorrect Humor** : **Computer and System Administration Humor**
- [Apr 15, 2022] **Is DevOps a yet another "for profit" technocult?**

Switchboard | **Latest** | **Past week** | **Past month**

NEWS CONTENTS

- 20200701 : Stupid Bash tricks- History, reusing arguments, files and directories, functions, and more *by Valentin Bajrami* (Jul 01, 2020 , www.redhat.com)
- 20200305 : How to tell if you're using a bash builtin in Linux (Mar 05, 2020 , www.networkworld.com)
- 20200305 : Bash IDE - Visual Studio Marketplace (Mar 05, 2020 , marketplace.visualstudio.com)
- 20191108 : Bash aliases you can't live without *by Seth Kenlon* (Jul 31, 2019 , opensource.com)
- 20191108 : Winterize your Bash prompt in Linux (Nov 08, 2019 , opensource.com)
- 20191108 : How to change the default shell prompt (Jun 29, 2014 , access.redhat.com)
- 20191108 : How to escape unicode characters in bash prompt correctly - Stack Overflow (Nov 08, 2019 , stackoverflow.com)
- 20191023 : Apply Tags To Linux Commands To Easily Retrieve Them From History (Oct 23, 2019 , www.ostechnix.com)
- 20190828 : Echo Command in Linux with Examples (Aug 28, 2019 , linoxide.com)
- 20190822 : How To Display Bash History Without Line Numbers - OSTechNix (Aug 22, 2019 , www.ostechnix.com)
- 20190814 : bash - PID background process - Unix Linux Stack Exchange (Aug 14, 2019 , unix.stackexchange.com)
- 20190814 : unix - How to get PID of process by specifying process name and store it in a variable to use further - Stack Overflow (Aug 14, 2019 , stackoverflow.com)
- ***20190814**** linux - How to get PID of background process - Stack Overflow ***(Aug 14, 2019 , stackoverflow.com) [Recommended]***
- 20190726 : Cheat.sh Shows Cheat Sheets On The Command Line Or In Your Code Editor> (Jul 26, 2019 , www.linuxuprising.com)
- 20190726 : What Is /dev/null in Linux *by Alexandru Andrei* (Jul 23, 2019 , www.maketecheasier.com)
- 20190129 : hstr -- Bash and zsh shell history suggest box - easily view, navigate, search and manage your command history (Nov 17, 2018 , dvorka.github.io)
- ***20190126**** Ten Things I Wish I'd Known About about bash ***(Jan 06, 2018 , zwischenzugs.com) [Recommended]***
- 20181117 : hh command man page (Nov 17, 2018 , www.mankier.com)
- 20181017 : How to use arrays in bash script - LinuxConfig.org (Oct 17, 2018 , linuxconfig.org)
- 20181010 : Bash History Display Date And Time For Each Command (Oct 10, 2018 , www.cyberciti.biz)
- 20180609 : How to use the history command in Linux Opensource.com (Jun 09, 2018 , opensource.com)
- 20180601 : Introduction to Bash arrays *by Robert Aboukhalil* (Jun 01, 2018 , opensource.com)
- 20180528 : Useful Linux Command Line Bash Shortcuts You Should Know (May 28, 2018 , www.tecmint.com)
- 20171031 : High-speed Bash *by Tom Ryder* (Jan 24, 2012 , sanctum.geek.nz)
- 20171031 : Learning the content of /bin and /usr/bin *by Tom Ryder* (Mar 16, 2012 , sanctum.geek.nz)
- 20171031 : Shell config subfiles *by Tom Ryder* (Jan 30, 2015 , sanctum.geek.nz)
- 20171031 : Better Bash history *by Tom Ryder* (Feb 21, 2012 , sanctum.geek.nz)
- 20171031 : Bash history expansion *by Tom Ryder* (Aug 16, 2012 , sanctum.geek.nz)
- 20171031 : Prompt directory shortening *by Tom Ryder* (Nov 07, 2014 , sanctum.geek.nz)
- 20171020 : Simple logical operators in Bash - Stack Overflow (Oct 20, 2017 , stackoverflow.com)
- 20171019 : Bash One-Liners bashoneliners.com (Oct 19, 2017 , www.bashoneliners.com)
- 20171016 : Indenting Here-Documents - bash Cookbook (Oct 16, 2017 , www.safaribooksonline.com)
- 20171016 : Indenting bourne shell here documents (Oct 16, 2017 , prefetch.net)
- 20171009 : TMOUT - Auto Logout Linux Shell When There Isn't Any Activity *by Aaron Kili* (Oct 07, 2017 , www.tecmint.com)
- 20170927 : Arithmetic Evaluation (Sep 27, 2017 , mywiki.wooledge.org)
- 20170927 : Integer ASCII value to character in BASH using printf (Sep 27, 2017 , stackoverflow.com)
- 20170927 : linux - How to convert DOS-Windows newline (CRLF) to Unix newline in a Bash script (linux - How to convert DOS-Windows newline (CRLF) to Unix newline in a Bash script, Sep 27, 2017)
- 20170829 : How to view the `.bash_history` file via command line (Aug 29, 2017 , askubuntu.com)
- 20170729 : Preserve bash history in multiple terminal windows - Unix Linux Stack Exchange (Jul 29, 2017 , unix.stackexchange.com)
- 20170729 : shell - How does this bash code detect an interactive session - Stack Overflow (Jul 29, 2017 , stackoverflow.com)
- 20170726 : I feel stupid declare not found in bash scripting (www.linuxquestions.org)
- 20170726 : Associative array declaration gotcha (Jul 26, 2017 , unix.stackexchange.com)
- 20170726 : Typing variables: declare or typeset (Jul 26, 2017 , www.tldp.org)
- 20170725 : Arrays in bash 4.x (Jul 25, 2017 , wiki.bash-hackers.org)
- 20170725 : Local variables (Jul 25, 2017 , wiki.bash-hackers.org)
- 20170725 : Environment variables (Jul 25, 2017 , wiki.bash-hackers.org)
- 20170725 : Block commenting (Jul 25, 2017 , wiki.bash-hackers.org)
- 20170725 : Doing specific tasks: concepts, methods, ideas (Jul 25, 2017 , wiki.bash-hackers.org)
- 20170725 : Keeping persistent history in bash (Jul 25, 2017 , eli.thegreenplace.net)
- 20170724 : Bash history handling with multiple terminals (get=)
- 20170313 : 6.3 Arrays (Mar 13, 2017 , name="KSH-CH-6-SECT-3">)
- 20170313 : Leaning the Korn shell: Chapter 6 Integer Variables and Arithmetic (Mar 13, 2017 , docstore.mik.ua)
- 20170214 : Ms Dos style aliases for linux (Feb 14, 2017 , bash.cyberciti.biz)
- 20170204 : Quickly find differences between two directories (Feb 04, 2017 , www.cyberciti.biz)
- 20170204 : Restoring deleted /tmp folder (Jan 13, 2015 , cyberciti.biz)
- 20170204 : Use CDPATH to access frequent directories in bash - Mac OS X Hints (Feb 04, 2017 , hints.macworld.com)
- 20170204 : Copy file into multiple directories (Feb 04, 2017 , www.cyberciti.biz)
- 20170204 : 20 Unix Command Line Tricks – Part I (Feb 04, 2017 , www.cyberciti.biz)
- 20170204 : List all files or directories on your system (Feb 04, 2017 , www.cyberciti.biz)
- 20170204 : basic ~~.bashrc ~~.bash_profile tips thread (Arch Linux Forums)
- 20170204 : What are some useful Bash tricks - Quora (What are some useful Bash tricks - Quora,)
- 20170204 : What are some useful .bash_profile and .bashrc tips - Quora (What are some useful .bash_profile and .bashrc tips - Quora,)
- 20170204 : My Favorite bash Tips and Tricks Linux Journal (My Favorite bash Tips and Tricks Linux Journal,)
- 20161219 : Unknown Bash Tips and Tricks For Linux Linux.com The source for Linux information (Unknown Bash Tips and Tricks For Linux Linux.com The source for Linux information, Dec 19, 2016)
- 20161219 : Bash Tricks " Linux Magazine (Bash Tricks " Linux Magazine, Dec 19, 2016)
- 20161219 : Linux secrets most users dont know about ITworld (Linux secrets most users don't know about ITworld,)
- 20151206 : Bash For Loop Examples (June 24, 2015 , cyberciti.biz)
- 20151108 : Get timestamps on Bash's History (nickgeoghegan.net)
- 20140508 : 25 Even More – Sick Linux Commands UrFixs Blog (25 Even More – Sick Linux Commands UrFix's Blog, May 08, 2014)
- 20140508 : 25 Best Linux Commands UrFixs Blog (25 Best Linux Commands UrFix's Blog, May 08, 2014)
- 20121216 : bash - how do I list the functions defined in my shell - Stack Overflow (bash - how do I list the functions defined in my shell - Stack Overflow, Dec 16, 2012)
- 20121216 : Unknown Bash Tips and Tricks For Linux Linux.com (Unknown Bash Tips and Tricks For Linux Linux.com,)
- 20121216 : My Favorite Bash Substitution Tricks Drastic Code (My Favorite Bash Substitution Tricks Drastic Code,)

Old News ;-)

[Jun 12, 2021] Ctrl-R -- Find and run a previous command

Jun 12, 2021 | anto.online

What if you needed to execute a specific command again, one which you used a while back? And you can't remember the first character, but you can remember you used the word "serve".

You can use the up key and keep on pressing up until you find your command. (That could take some time)

Or, you can enter CTRL + R and type few keywords you used in your last command. Linux will help locate your command, requiring you to press enter once you found your command. The example below shows how you can enter CTRL + R and then type "ser" to find the previously run "PHP artisan serve" command. For sure, this tip will help you speed up your command-line experience.

```
anto@odin:~$
(reverse-i-search)`ser': php artisan serve
```

You can also use the history command to output all the previously stored commands. The history command will give a list that is ordered in ascending relative to its execution.

[Jun 12, 2021] The use of PS4= LINENO in debugging bash scripts

Jun 10, 2021 | www.redhat.com

Exit status

In Bash scripting, $? prints the exit status. If it returns zero, it means there is no error. If it is non-zero, then you can conclude the earlier task has some issue.

A basic example is as follows:

```
$ cat myscript.sh
        #!/bin/bash
        mkdir learning
        echo $?
```

If you run the above script once, it will print 0 because the directory does not exist, therefore the script will create it. Naturally, you will get a non-zero value if you run the script a second time, as seen below:

```
$ sh myscript.sh
mkdir: cannot create directory 'learning': File exists
1
```

In the cloud

- Understanding cloud computing
- Free course: Red Hat OpenStack Technical Overview
- Free e-book: Hybrid Cloud Strategy for Dummies

Best practices

It is always recommended to enable the debug mode by adding the -e option to your shell script as below:

```
$ cat test3.sh
!/bin/bash
set -x
echo "hello World"
mkdiir testing
 ./test3.sh
+ echo 'hello World'
hello World
+ mkdiir testing
./test3.sh: line 4: mkdiir: command not found
```

You can write a debug function as below, which helps to call it anytime, using the example below:

```
$ cat debug.sh
#!/bin/bash
_DEBUG="on"
function DEBUG()
{
 [ "$_DEBUG" == "on" ] && $@
}
DEBUG echo 'Testing Debudding'
DEBUG set -x
a=2
b=3
c=$(( $a + $b ))
DEBUG set +x
echo "$a + $b = $c"
```

Which prints:

```
$ ./debug.sh
Testing Debudding
+ a=2
+ b=3
+ c=5
+ DEBUG set +x
+ '[' on == on ']'
+ set +x
2 + 3 = 5
```

Standard error redirection

You can redirect all the system errors to a custom file using standard errors, which can be denoted by the number 2 . Execute it in normal Bash commands, as demonstrated below:

```
$ mkdir users 2> errors.txt
$ cat errors.txt
mkdir: cannot create directory "~users': File exists
```

Most of the time, it is difficult to find the exact line number in scripts. To print the line number with the error, use the PS4 option (supported with Bash 4.1 or later). Example below:

```
$ cat test3.sh
#!/bin/bash
PS4='LINENO:'

set -x
echo "hello World"
mkdiir testing
```

You can easily see the line number while reading the errors:

```
$ /test3.sh
5: echo 'hello World'
hello World
6: mkdiir testing
./test3.sh: line 6: mkdiir: command not found
```

[Jun 08, 2021] Bang commands: two potentially useful shortcuts for command line -- !! and !$ by Nikolai Bezroukov

softpanorama.org

Those shortcuts belong to the class of commands known as ***bang commands*** . Internet search for this term provides a wealth of additional information (which probably you do not need ;-), I will concentrate on just most common and potentially useful in the current command line environment bang commands. Of them !$ is probably the most useful and definitely is the most widely used. For many sysadmins it is the only bang command that is regularly used.

1. **!!** is the bang command that re-executes the last command . This command is used mainly as a shortcut sudo !! -- elevation of privileges after your command failed on your user account. For example:

   ```
   fgrep 'kernel' /var/log/messages # it will fail due to unsufficient privileges, as /var/log directory is not readable by ordinary user
   sudo !! # now we re-execute the command with elevated privileges
   ```

2. !$ puts into the current command line the last argument from previous command . For example:

   ```
   mkdir -p /tmp/Bezroun/Workdir
   cd !$
   ```

   In this example the last command is equivalent to the command cd /tmp/Bezroun/Workdir. Please try this example. It is a pretty neat trick.

NOTE: You can also work with individual arguments using numbers.

- !:1 is the previous command and its options
- !:2 is the first argument of the previous command
- !:3 is the second
- And so on

For example:

```
cp !:2 !:3 # picks up  the first and the second argument from the previous command
```

For this and other bang command capabilities, copying fragments of the previous command line using mouse is much more convenient, and you do not need to remember extra staff. After all, band commands were created before mouse was available, and most of them reflect the realities and needs of this bygone era. Still I met sysadmins that use this and some additional capabilities like !!:s^<old>^<new> (which replaces the string 'old' with the string 'new" and re-executes previous command) even now.

The same is true for !* -- all arguments of the last command. I do not use them and have had troubles writing this part of this post, correcting it several times to make it right 4/0

Nowadays CTRL+R activates reverse search, which provides an easier way to navigate through your history then capabilities in the past provided by band commands.

[May 28, 2021] 10 Amazing and Mysterious Uses of (!) Symbol or Operator in Linux Commands

Images removed. See the original for the full text.

Notable quotes:

"... You might also mention !? It finds the last command with its' string argument. For example, if"¦ ..."

"... I didn't see a mention of historical context in the article, so I'll give some here in the comments. This form of history command substitution originated with the C Shell (csh), created by Bill Joy for the BSD flavor of UNIX back in the late 70's. It was later carried into tcsh, and bash (Bourne-Again SHell). ..."

linuxiac.com

The The '!' symbol or operator in Linux can be used as Logical Negation operator as well as to fetch commands from history with tweaks or to run previously run command with modification. All the commands below have been checked explicitly in bash Shell. Though I have not checked but a major of these won't run in other shell. Here we go into the amazing and mysterious uses of '!' symbol or operator in Linux commands.

4. How to handle two or more arguments using (!)

Let's say I created a text file 1.txt on the Desktop.

```
$ touch /home/avi/Desktop/1.txt
```

and then copy it to "~ /home/avi/Downloads "~ using complete path on either side with cp command.

```
$ cp /home/avi/Desktop/1.txt /home/avi/downloads
```

Now we have passed two arguments with cp command. First is "~ /home/avi/Desktop/1.txt "~ and second is "~ /home/avi/Downloads "~, lets handle them differently, just execute echo [arguments] to print both arguments differently.

```
$ echo "1st Argument is : !^"
$ echo "2nd Argument is : !cp:2"
```

Note 1st argument can be printed as "!^" and rest of the arguments can be printed by executing "![Name_of_Command]:[Number_of_argument]" .

In the above example the first command was "~ cp "~ and 2nd argument was needed to print. Hence "!cp:2" , if any command say xyz is run with 5 arguments and you need to get 4th argument, you may use "!xyz:4" , and use it as you like. All the arguments can be accessed by "!*" .

5. Execute last command on the basis of keywords

We can execute the last executed command on the basis of keywords. We can understand it as follows:

```
$ ls /home > /dev/null                  [Command 1]
$ ls -l /home/avi/Desktop > /dev/null       [Command 2]
$ ls -la /home/avi/Downloads > /dev/null    [Command 3]
$ ls -lA /usr/bin > /dev/null           [Command 4]
```

Here we have used same command (ls) but with different switches and for different folders. Moreover we have sent to output of each command to "~ /dev/null "~ as we are not going to deal with the output of the command also the console remains clean.

Now Execute last run command on the basis of keywords.

```
$ ! ls          [Command 1]
$ ! ls -l       [Command 2]
$ ! ls -la      [Command 3]
$ ! ls -lA      [Command 4]
```

Check the output and you will be astonished that you are running already executed commands just by ls keywords.

Run Commands Based on Keywords

6. The power of !! Operator

You can run/alter your last run command using (!!) . It will call the last run command with alter/tweak in the current command. Lets show you the scenario

Last day I run a one-liner script to get my private IP so I run,

```
$ ip addr show | grep inet | grep -v 'inet6'| grep -v '127.0.0.1' | awk '{print $2}' | cut -f1 -d/
```

Then suddenly I figured out that I need to redirect the output of the above script to a file ip.txt , so what should I do? Should I retype the whole command again and redirect the output to a file? Well an easy solution is to use UP navigation key and add '> ip.txt' to redirect the output to a file as.

```
$ ip addr show | grep inet | grep -v 'inet6'| grep -v '127.0.0.1' | awk '{print $2}' | cut -f1 -d/ > ip.txt
```

Thanks to the life Savior UP navigation key here. Now consider the below condition, the next time I run below one-liner script.

```
$ ifconfig | grep "inet addr:" | awk '{print $2}' | grep -v '127.0.0.1' | cut -f2 -d:
```

As soon as I run script, the bash prompt returned an error with the message "bash: ifconfig: command not found" , It was not difficult for me to guess I run this command as user where it should be run as root.

So what's the solution? It is difficult to login to root and then type the whole command again! Also (UP Navigation Key) in last example didn't came to rescue here. So? We need to call "!!" without quotes, which will call the last command for that user.

```
$ su -c "!!" root
```

Here su is switch user which is root, -c is to run the specific command as the user and the most important part !! will be replaced by command and last run command will be substituted here. Yeah! You need to provide root password.

I make use of !! mostly in following scenarios,

1. When I run apt-get command as normal user, I usually get an error saying you don't have permission to execute.

```
$ apt-get upgrade && apt-get dist-upgrade
```

Opps error"¦don't worry execute below command to get it successful..

```
$ su -c !!
```

Same way I do for,

```
$ service apache2 start
or
$ /etc/init.d/apache2 start
or
$ systemctl start apache2
```

OOPS User not authorized to carry such task, so I run..

```
$ su -c 'service apache2 start'
or
$ su -c '/etc/init.d/apache2 start'
```

or
$ su -c 'systemctl start apache2'

7. Run a command that affects all the file except ![FILE_NAME]

The ! (Logical NOT) can be used to run the command on all the files/extension except that is behind '!' .

A. Remove all the files from a directory except the one the name of which is 2.txt .

$ rm !(2.txt)

B. Remove all the file type from the folder except the one the extension of which is "˜ pdf "˜.

$ $ rm !(*.pdf)

... ...

- ***Edgar Allen*** May 19, 2015 at 10:30 pm

  You might also mention !? It finds the last command with its' string argument. For example, if"¦

  1013 grep tornado /usr/share/dict/words
  1014 grep hurricane /usr/share/dict/words
  1015 wc -l /usr/share/dict/words

  are all in the history then !?torn will grep for tornado again where !torn would search in vain for a command starting with torn.

  And `wc !?torn?:2` works to select argument two from the command containing tornado and run `wc` on it.

- ***Stephen*** May 19, 2015 at 6:07 pm

  I didn't see a mention of historical context in the article, so I'll give some here in the comments. This form of history command substitution originated with the C Shell (csh), created by Bill Joy for the BSD flavor of UNIX back in the late 70's. It was later carried into tcsh, and bash (Bourne-Again SHell).

  Personally, I've always preferred the C-shell history substitution mechanism, and never really took to the fc command (that I first encountered in the Korne shell).

- ***suzy*** May 16, 2015 at 11:45 am

  4th command. You can access it much simpler. There are actually regular expressions:

  - ^ -- is at the begging expression
  - $ -- is at the end expression
  - :number -- any number parameter

  Examples:

  touch a.txt b.txt c.txt
  echo !^ ""> display first parameter
  echo !:1 ""> also display first parameter
  echo !:2 ""> display second parameter
  echo !:3 ""> display third parameter
  echo !$ ""> display last (in our case 3th) parameter
  echo !* ""> display all parameters

- ***Tomasz Wiszkowski*** May 16, 2015 at 10:50 am

  I think (5) works differently than you pointed out, and redirection to devnull hides it, but ZSh still prints the command.

  When you invoke "! ls"¦"□, it always picks the last ls command you executed, just appends your switches at the end (after /dev/null).

  One extra cool thing is the !# operator, which picks arguments from current line. Particularly good if you need to retype long path names you already typed in current line. Just say, for example

  cp /some/long/path/to/file.abc !#:1

  And press tab. It's going to replace last argument with entire path and file name.

  - ***Avishek Kumar*** May 18, 2015 at 11:37 pm

    Tomasz,

    For your first part of feedback: It doesn't pick the last command executed and just to prove this we have used 4 different switches for same command. ($! ls $! ls -l $! ls -la $! ls -lA). Now you may check it by entering the keywords in any order and in each case it will output the same result.

    As far as it is not working in ZSH as expected, i have already mentioned that it i have tested it on BASH and most of these won't work in other shell.

    For the second part, what you mentioned is a HASH TAG in Linux Command Line and we have included it in one of our article. You may like to read it here: https://www.tecmint.com/linux-commandline-chat-server-and-remove-unwanted-packages/

[May 28, 2021] Bash scripting- Moving from backtick operator to $ parentheses

May 20, 2021 | www.redhat.com

You can achieve the same result by replacing the backticks with the $ parens, like in the example below:

â¯ echo "There are $(ls | wc -l) files in this directory"
There are 3 files in this directory

Here's another example, still very simple but a little more realistic. I need to troubleshoot *something* in my network connections, so I decide to show my total and waiting connections minute by minute.

```
â¯ cat netinfo.sh
#!/bin/bash
while true
do
  ss -an > netinfo.txt
  connections_total=$(cat netinfo.txt | wc -l)
  connections_waiting=$(grep WAIT netinfo.txt | wc -l)
  printf "$(date +%R) - Total=%6d Waiting=%6d\n" $connections_total $connections_waiting
  sleep 60
done

â¯ ./netinfo.sh
22:59 - Total=  2930 Waiting=   977
23:00 - Total=  2923 Waiting=   963
23:01 - Total=  2346 Waiting=   397
23:02 - Total=  2497 Waiting=   541
```

It doesn't seem like a *huge* difference, right? I just had to adjust the syntax. Well, there are some implications involving the two approaches. If you are like me, who automatically uses the backticks without even blinking, keep reading.

Deprecation and recommendations

Deprecation sounds like a bad word, and in many cases, it might really be bad.

When I was researching the explanations for the backtick operator, I found some discussions about "are the backtick operators deprecated?"

The short answer is: Not in the sense of "on the verge of becoming unsupported and stop working." However, backticks should be avoided and replaced by the $ parens syntax.

The main reasons for that are (in no particular order):

1. Backticks operators can become messy if the internal commands also use backticks.

- You will need to escape the internal backticks, and if you have single quotes as part of the commands or part of the results, reading and troubleshooting the script can become difficult.
- If you start thinking about *nesting* backtick operators inside other backtick operators, things will not work as expected or not work at all. Don't bother.

2. The $ parens operator is safer and more predictable.

- What you code inside the $ parens operator is treated as a shell script. Syntactically it is the same thing as having that code in a text file, so you can expect that everything you would code in an isolated shell script would work here.

Here are some examples of the behavioral differences between backticks and $ parens:

```
â¯ echo '\$x'
\$x

â¯ echo `echo '\$x'`
$x

â¯ echo $(echo '\$x')
\$x
```

You can find additional examples of the differences between backticks and $ parens behavior here .

[Free cheat sheet: Get a list of Linux utilities and commands for managing servers and networks .]

Wrapping up

If you compare the two approaches, it seems logical to think that you should always/only use the $ parens approach. And you might think that the backtick operators are only used by sysadmins from an older era .

Well, that might be true, as sometimes I use things that I learned long ago, and in simple situations, my "muscle memory" just codes it for me. For those ad-hoc commands that you know that do not contain any nasty characters, you might be OK using backticks. But for anything that is more perennial or more complex/sophisticated, please go with the $ parens approach.

[May 23, 2021] Adding arguments and options to your Bash scripts

May 23, 2021 | www.redhat.com

“Handling options”

The ability for a Bash script to handle command line options such as -h to display help gives you some powerful capabilities to direct the program and modify what it does. In the case of your -h option, you want the program to print the help text to the terminal session and then quit without running the rest of the program. The ability to process options entered at the command line can be added to the Bash script using the while command in conjunction with the getops and case commands.

The getops command reads any and all options specified at the command line and creates a list of those options. The while command loops through the list of options by setting the variable $options for each in the code below. The case statement is used to evaluate each option in turn and execute the statements in the corresponding stanza. The while statement will continue to assess the list of options until they have all been processed or an exit statement is encountered, which terminates the program.

Be sure to delete the help function call just before the echo "Hello world!" statement so that the main body of the program now looks like this.

```
###############################################################
###############################################################
# Main program                                                #
###############################################################
###############################################################
###############################################################
# Process the input options. Add options as needed.        #
###############################################################
```

```
# Get the options
while getopts ":h" option; do
  case $option in
    h) # display Help
       Help
       exit;;
  esac
done

echo "Hello world!"
```

Notice the double semicolon at the end of the exit statement in the case option for -h . This is required for each option. Add to this case statement to delineate the end of each option.

Testing is now a little more complex. You need to test your program with several different options -- and no options -- to see how it responds. First, check to ensure that with no options that it prints "Hello world!" as it should.

```
[student@testvm1 ~]$ hello.sh
Hello world!
```

That works, so now test the logic that displays the help text.

```
[student@testvm1 ~]$ hello.sh -h
Add a description of the script functions here.

Syntax: scriptTemplate [-g|h|t|v|V]
options:
g     Print the GPL license notification.
h     Print this Help.
v     Verbose mode.
V     Print software version and exit.
```

That works as expected, so now try some testing to see what happens when you enter some unexpected options.

```
[student@testvm1 ~]$ hello.sh -x
Hello world!

[student@testvm1 ~]$ hello.sh -q
Hello world!

[student@testvm1 ~]$ hello.sh -lkjsahdf
Add a description of the script functions here.

Syntax: scriptTemplate [-g|h|t|v|V]
options:
g     Print the GPL license notification.
h     Print this Help.
v     Verbose mode.
V     Print software version and exit.

[student@testvm1 ~]$
```

“Handling invalid options”

The program just ignores the options for which you haven't created specific responses without generating any errors. Although in the last entry with the -lkjsahdf options, because there is an "h" in the list, the program did recognize it and print the help text. Testing has shown that one thing that is missing is the ability to handle incorrect input and terminate the program if any is detected.

You can add another case stanza to the case statement that will match any option for which there is no explicit match. This general case will match anything you haven't provided a specific match for. The *case* statement now looks like this.

```
while getopts ":h" option; do
  case $option in
    h) # display Help
       Help
       exit;;
   \?) # Invalid option
       echo "Error: Invalid option"
       exit;;
  esac
done
```

Kubernetes and OpenShift

- Free cheatsheet: Kubernetes and Minikube
- Free ebook: Designing Cloud-Native Applications
- Interactive course: Getting Started with OpenShift
- Free ebook: Build Applications with Kubernetes and Openshift

This bit of code deserves an explanation about how it works. It seems complex but is fairly easy to understand. The while – done structure defines a loop that executes once for each option in the getopts – option structure. The ":h" string -- which requires the quotes -- lists the possible input options that will be evaluated by the case – esac structure. Each option listed must have a corresponding stanza in the case statement. In this case, there are two. One is the h) stanza which calls the Help procedure. After the Help procedure completes, execution returns to the next program statement, exit;; which exits from the program without executing any more code even if some exists. The option processing loop is also terminated, so no additional options would be checked.

Notice the catch-all match of \? as the last stanza in the case statement. If any options are entered that are not recognized, this stanza prints a short error message and exits from the program.

Any additional specific cases must precede the final catch-all. I like to place the case stanzas in alphabetical order, but there will be circumstances where you want to ensure that a particular case is processed before certain other ones. The case statement is sequence sensitive, so be aware of that when you construct yours.

The last statement of each stanza in the case construct must end with the double semicolon (;;), which is used to mark the end of each stanza explicitly. This allows those programmers who like to use explicit semicolons for the end of each statement instead of implicit ones to continue to do so for each statement within each case stanza.

Test the program again using the same options as before and see how this works now.

The Bash script now looks like this.

```
#!/bin/bash
############################################################
# Help                                     #
############################################################
Help()
{
  # Display Help
  echo "Add description of the script functions here."
  echo
  echo "Syntax: scriptTemplate [-g|h|v|V]"
  echo "options:"
  echo "g     Print the GPL license notification."
  echo "h     Print this Help."
  echo "v     Verbose mode."
  echo "V     Print software version and exit."
  echo
}

############################################################
############################################################
# Main program                                     #
############################################################
############################################################
############################################################
# Process the input options. Add options as needed.        #
############################################################
# Get the options
while getopts ":h" option; do
  case $option in
    h) # display Help
       Help
       exit;;
   \?) # Invalid option
       echo "Error: Invalid option"
       exit;;
  esac
done


echo "hello world!"
```

Be sure to test this version of your program very thoroughly. Use random input and see what happens. You should also try testing valid and invalid options without using the dash (-) in front.

“Using options to enter data”

First, add a variable and initialize it. Add the two lines shown in bold in the segment of the program shown below. This initializes the $Name variable to "world" as the default.

```
<snip>
############################################################
############################################################
# Main program                                     #
############################################################
############################################################

# Set variables
Name="world"

############################################################
# Process the input options. Add options as needed.        #
<snip>
```

Change the last line of the program, the echo command, to this.

```
echo "hello $Name!"
```

Add the logic to input a name in a moment but first test the program again. The result should be exactly the same as before.

```
[dboth@david ~]$ hello.sh
hello world!
[dboth@david ~]$

# Get the options
while getopts ":hn:" option; do
  case $option in
    h) # display Help
       Help
       exit;;
    n) # Enter a name
       Name=$OPTARG;;
   \?) # Invalid option
       echo "Error: Invalid option"
       exit;;
  esac
done
```

$OPTARG is always the variable name used for each new option argument, no matter how many there are. You must assign the value in $OPTARG to a variable name that will be used in the rest of the program. This new stanza does not have an exit statement. This changes the program flow so that after processing all valid options in the case statement, execution moves on to the next statement after the case construct.

Test the revised program.

```
[dboth@david ~]$ hello.sh
hello world!

[dboth@david ~]$ hello.sh -n LinuxGeek46
hello LinuxGeek46!

[dboth@david ~]$ hello.sh -n "David Both"
hello David Both!
[dboth@david ~]$
```

The completed program looks like this.

```
#!/bin/bash
############################################################
# Help                                                     #
############################################################
Help()
{
  # Display Help
  echo "Add description of the script functions here."
  echo
  echo "Syntax: scriptTemplate [-g|h|v|V]"
  echo "options:"
  echo "g     Print the GPL license notification."
  echo "h     Print this Help."
  echo "v     Verbose mode."
  echo "V     Print software version and exit."
  echo
}

############################################################
############################################################
# Main program                                             #
############################################################
############################################################

# Set variables
Name="world"

############################################################
# Process the input options. Add options as needed.        #
############################################################
# Get the options
while getopts ":hn:" option; do
  case $option in
    h) # display Help
       Help
       exit;;
    n) # Enter a name
       Name=$OPTARG;;
   \?) # Invalid option
       echo "Error: Invalid option"
       exit;;
  esac
done


echo "hello $Name!"
```

Be sure to test the help facility and how the program reacts to invalid input to verify that its ability to process those has not been compromised. If that all works as it should, then you have successfully learned how to use options and option arguments.

[Dec 10, 2020] Linux Subshells for Beginners With Examples - LinuxConfig.org

Dec 10, 2020 | linuxconfig.org

Bash allows two different subshell syntaxes, namely $() and back-tick surrounded statements. Let's look at some easy examples to start:

```
$ echo '$(echo 'a')'
$(echo a)
$ echo "$(echo 'a')"
a
$ echo "a$(echo 'b')c"
abc
$ echo "a`echo 'b'`c"
abc
```

SUBSCRIBE TO NEWSLETTER
Subscribe to Linux Career NEWSLETTER and receive latest Linux news, jobs, career advice and tutorials.

https://googleads.g.doubleclick.net/pagead/ads?guci=2.2.0.0.2.2.0.0&gdpr=0&us_privacy=1---&client=ca-pub-4906753266448300&output=html&h=189&slotname=5703296903&adk=1248373483&adf=1566064928&pi=t.ma~as.5703296903&w=754&fwrn=4&lmt=1606768699&rafmt=11subshells-for-beginners-with-examples&flash=0&wgl=1&tt_state=W3siaXNzdWVyT3JpZ2luIjoiaHR0cHM6Ly9hZHNlcnZpY2UuZ29vZ2xlLmNvbSIsInN0YXRlIjowfSx7ImIzc3VIck9yaWdpbiI6Imh0dHBzOi8vYM&shv=r20201112&cbv=r20190131&ptt=9&saldr=aa&abxe=1&cookie=ID%3Da3d6872c3b570d2f-2256edf9acc400fe%3AT%3D1604637667%3ART%3D1604637667%3AS%3DALNI_MboWqYGLjuR1MmbPrvzRe-G7T4AZw&correlator=5015138629854&frm=20&pv=2&ga_vid=1677892679.1604637667&ga_sid=1606768711&ga_hid=1690704763&ga_fc=0&iag=0&icsg=57724359842431

In the first command, as an example, we used ' single quotes. This resulted in our subshell command, inside the single quotes, to be interpreted as literal text instead of a command. This is standard Bash: ' indicates literal, " indicates that the string will be parsed for subshells and variables.

In the second command we swap the ' to " and thus the string is parsed for actual commands and variables. The result is that a subshell is being started, thanks to our subshell syntax ($()), and the command inside the subshell (echo 'a') is being executed literally, and thus an a is produced, which is then inserted in the overarching / top level echo . The command at that stage can be read as echo "a" and thus the output is a .

In the third command, we further expand this to make it clearer how subshells work in-context. We echo the letter b inside the subshell, and this is joined on the left and the right by the letters a and c yielding the overall output to be abc in a similar fashion to the second command.

In the fourth and last command, we exemplify the alternative Bash subshell syntax of using back-ticks instead of $() . It is important to know that $() is the preferred syntax, and that in some remote cases the back-tick based syntax may yield some parsing errors where the $() does not. I would thus strongly encourage you to always use the $() syntax for subshells, and this is also what we will be using in the following examples.

Example 2: A little more complex

```
$ touch a
$ echo "-$(ls [a-z])"
-a
$ echo "-=-||$(ls [a-z] | xargs ls -l)||-=-"
-=-||-rw-rw-r-- 1 roel roel 0 Sep  5 09:26 a||-=-
```

Here, we first create an empty file by using the touch a command. Subsequently, we use echo to output something which our subshell $(ls [a-z]) will generate. Sure, we can execute the ls directly and yield more or less the same result, but note how we are adding - to the output as a prefix.

In the final command, we insert some characters at the front and end of the echo command which makes the output look a bit nicer. We use a subshell to first find the a file we created earlier (ls [a-z]) and then - still inside the subshell - pass the results of this command (which would be only a literally - i.e. the file we created in the first command) to the ls -l using the pipe (|) and the xargs command. For more information on xargs, please see our articles xargs for beginners with examples and multi threaded xargs with examples .

Example 3: Double quotes inside subshells and sub-subshells!

```
echo "$(echo "$(echo "it works")" | sed 's|it|it surely|')"
it surely works
```

---

https://googleads.g.doubleclick.net/pagead/ads?guci=2.2.0.0.2.2.0.0&gdpr=0&us_privacy=1---&client=ca-pub-4906753266448300&output=html&h=189&slotname=5703296903&adk=1248373483&adf=2724449972&pi=t.ma~as.5703296903&w=754&fwrn=4&lmt=1606768699&rafmt=11subshells-for-beginners-with-examples&flash=0&wgl=1&adsid=ChEIgM2S_gUQzN_42M_QwuOnARIqAL1WgU0IPKMTPLYMrAFnUAY1w18hzIzNy0CGR82uXn3xCpt9jLaEISQY&tt_state=W3siaXNzdWVy2256edf9acc400fe%3AT%3D1604637667%3ART%3D1604637667%3AS%3DALNI_MboWqYGLjuR1MmbPrvzRe-G7T4AZw&prev_fmts=754x189%2C0x0&nras=1&correlator=5015138629854&frm=20&pv=1&ga_vid=1677892679.1604637667&ga_sid=1606768711&ga_hid=1690704763&gH_kCAb-qHdw3GuwXq6RB3MJbCIRq9VISu7n8l1rpQZCm8sfL6sdfh-BMTltKIaB0w&pvsid=3023641763965231&pem=477&ref=https%3A%2F%2Fwww.linuxtoday.com%2F&rx=0&eae=0&fc=896&brdim=1536%2C0%2C1536%2C0%2C1536%2C11-30-05&ifi=2&uci=a!2&btvi=2&fsb=1&xpc=0ozDx6KTCC&p=https%3A//linuxconfig.org&dtd=4517

Cool, no? Here we see that double quotes can be used inside the subshell without generating any parsing errors. We also see how a subshell can be nested inside another subshell. Are you able to parse the syntax? The easiest way is to start "in the middle or core of all subshells" which is in this case would be the simple echo "it works" .

This command will output it works as a result of the subshell call $(echo "it works") . Picture it works in place of the subshell, i.e.

```
echo "$(echo "it works" | sed 's|it|it surely|')"
it surely works
```

This looks simpler already. Next it is helpful to know that the sed command will do a substitute (thanks to the s command just before the | command separator) of the text it to it surely . You can read the sed command as replace __it__ with __it surely__. The output of the subshell will thus be it surely works`, i.e.

```
echo "it surely works"
it surely works
```

Conclusion

In this article, we have seen that subshells surely work (pun intended), and that they can be used in wide variety of circumstances, due to their ability to be inserted inline and within the context of the overarching command. Subshells are very powerful and once you start using them, well, there will likely be no stopping. Very soon you will be writing something like:

```
$ VAR="goodbye"; echo "thank $(echo "${VAR}" | sed 's|^| and |')" | sed 's|k |k you|'
```

This one is for you to try and play around with! Thank you and goodbye

[Sep 05, 2020] documentation - How do I get the list of exit codes (and-or return codes) and meaning for a command-utility

Sep 05, 2020 | unix.stackexchange.com

What exit code should I use?

- 1 - Catchall for general errors. The exit code is 1 as the operation was not successful.
- 2 - Misuse of shell builtins (according to Bash documentation)
- 126 - Command invoked cannot execute.
- 127 - "command not found".
- 128 - Invalid argument to exit.
- 128+n - Fatal error signal "n".

- 130 - Script terminated by Control-C.
- 255* - Exit status out of range.

There is no "recipe" to get the meanings of an exit status of a given terminal command.

My first attempt would be the manpage:

```
user@host:~# man ls
  Exit status:
     0     if OK,

     1     if minor problems (e.g., cannot access subdirectory),

     2     if serious trouble (e.g., cannot access command-line argument).
```

Second : Google . See wget as an example.

Third : The exit statuses of the shell, for example bash. Bash and it's builtins may use values above 125 specially. 127 for command not found, 126 for command not executable. For more information see the bash exit codes .

Some list of sysexits on both Linux and BSD/OS X with preferable exit codes for programs (64-78) can be found in /usr/include/sysexits.h (or: man sysexits on BSD):

```
0   /* successful termination */
64  /* base value for error messages */
64  /* command line usage error */
65  /* data format error */
66  /* cannot open input */
67  /* addressee unknown */
68  /* host name unknown */
69  /* service unavailable */
70  /* internal software error */
71  /* system error (e.g., can't fork) */
72  /* critical OS file missing */
73  /* can't create (user) output file */
74  /* input/output error */
75  /* temp failure; user is invited to retry */
76  /* remote error in protocol */
77  /* permission denied */
78  /* configuration error */
/* maximum listed value */
```

The above list allocates previously unused exit codes from 64-78. The range of unallotted exit codes will be further restricted in the future.

However above values are mainly used in sendmail and used by pretty much nobody else, so they aren't anything remotely close to a standard (as pointed by @Gilles).

In shell the exit status are as follow (based on Bash):

- 1 - 125 - Command did not complete successfully. Check the command's man page for the meaning of the status, few examples below:
- 1 - Catchall for general errors

  Miscellaneous errors, such as "divide by zero" and other impermissible operations.

  Example:

  ```
  $ let "var1 = 1/0"; echo $?
  -bash: let: var1 = 1/0: division by 0 (error token is "0")
  1
  ```

- 2 - Misuse of shell builtins (according to Bash documentation)

  Missing keyword or command, or permission problem (and diff return code on a failed binary file comparison).

  Example:

  ```
  empty_function() {}
  ```

- 6 - No such device or address

  Example:

  ```
  $ curl foo; echo $?
  curl: (6) Could not resolve host: foo
  6
  ```

- 124 - command times out
- 125 - if a command itself fails see: coreutils
- 126 - if command is found but cannot be invoked (e.g. is not executable)

  Permission problem or command is not an executable.

  Example:

  ```
  $ /dev/null
  $ /etc/hosts; echo $?
  -bash: /etc/hosts: Permission denied
  126
  ```

- 127 - if a command cannot be found, the child process created to execute it returns that status

  Possible problem with $PATH or a typo.

Example:

```
$ foo; echo $?
-bash: foo: command not found
127
```

- 128 - Invalid argument to exit

  exit takes only integer args in the range 0 - 255.

  Example:

  ```
  $ exit 3.14159
  -bash: exit: 3.14159: numeric argument required
  ```

- 128 - 254 - fatal error signal "n" - command died due to receiving a signal. The signal code is added to 128 (128 + SIGNAL) to get the status (Linux: man 7 signal , BSD: man signal), few examples below:

- 130 - command terminated due to Ctrl-C being pressed, 130-128=2 (SIGINT)

  Example:

  ```
  $ cat
  ^C
  $ echo $?
  130
  ```

- 137 - if command is sent the KILL(9) signal (128+9), the exit status of command otherwise

  kill -9 $PPID of script.

- 141 - SIGPIPE - write on a pipe with no reader

  Example:

  ```
  $ hexdump -n100000 /dev/urandom | tee &>/dev/null >(cat > file1.txt) >(cat > file2.txt) >(cat > file3.txt) >(cat > file4.txt) >(cat > file5.txt)
  $ find . -name '*.txt' -print0 | xargs -r0 cat | tee &>/dev/null >(head /dev/stdin > head.out) >(tail /dev/stdin > tail.out)
  xargs: cat: terminated by signal 13
  $ echo ${PIPESTATUS[@]}
  0 125 141
  ```

- 143 - command terminated by signal code 15 (128+15=143)

  Example:

  ```
  $ sleep 5 && killall sleep &
  [1] 19891
  $ sleep 100; echo $?
  Terminated: 15
  143
  ```

- 255 * - exit status out of range.

  exit takes only integer args in the range 0 - 255.

  Example:

  ```
  $ sh -c 'exit 3.14159'; echo $?
  sh: line 0: exit: 3.14159: numeric argument required
  255
  ```

  According to the above table, exit codes 1 - 2, 126 - 165, and 255 have special meanings, and should therefore be avoided for user-specified exit parameters.

Please note that out of range exit values can result in unexpected exit codes (e.g. exit 3809 gives an exit code of 225, 3809 % 256 = 225).

See:

- Appendix E. Exit Codes With Special Meanings at Advanced Bash-Scripting Guide
- Writing Better Shell Scripts – Part 2 at Innovationsts

You will have to look into the code/documentation. However the thing that comes closest to a "standardization" is errno.h share improve this answer follow answered Jan 22 '14 at 7:35 Thorsten Staerk 2,885 1 1 gold badge 17 17 silver badges 25 25 bronze badges

PSkocik ,

thanks for pointing the header file.. tried looking into the documentation of a few utils.. hard time finding the exit codes, seems most will be the stderrs... – precise Jan 22 '14 at 9:13

[Jul 12, 2020] How to add a Help facility to your Bash program - Opensource.com

Jul 12, 2020 | opensource.com

How to add a Help facility to your Bash program In the third article in this series, learn about using functions as you create a simple Help facility for your Bash script. 20 Dec 2019 David Both (Correspondent) Feed 53 up Image by : Opensource.com x **Subscribe now**

Get the highlights in your inbox every week.

https://opensource.com/eloqua-embedded-email-capture-block.html?offer_id=70160000000QzXNAA0

In the first article in this series, you created a very small, one-line Bash script and explored the reasons for creating shell scripts and why they are the most efficient option for the system administrator, rather than compiled programs. In the second article , you began the task of creating a fairly simple template that you can use as a starting point for other Bash programs, then explored ways to test it.

This third of the four articles in this series explains how to create and use a simple Help function. While creating your Help facility, you will also learn about using functions and how to handle command-line options such as -h .

Why Help? More on Bash

- Bash cheat sheet
- An introduction to programming with Bash
- A sysadmin's guide to Bash scripting
- Latest Bash articles

Even fairly simple Bash programs should have some sort of Help facility, even if it is fairly rudimentary. Many of the Bash shell programs I write are used so infrequently that I forget the exact syntax of the command I need. Others are so complex that I need to review the options and arguments even when I use them frequently.

Having a built-in Help function allows you to view those things without having to inspect the code itself. A good and complete Help facility is also a part of program documentation.

About functions

Shell functions are lists of Bash program statements that are stored in the shell's environment and can be executed, like any other command, by typing their name at the command line. Shell functions may also be known as procedures or subroutines, depending upon which other programming language you are using.

Functions are called in scripts or from the command-line interface (CLI) by using their names, just as you would for any other command. In a CLI program or a script, the commands in the function execute when they are called, then the program flow sequence returns to the calling entity, and the next series of program statements in that entity executes.

The syntax of a function is:

```
FunctionName(){program statements}
```

Explore this by creating a simple function at the CLI. (The function is stored in the shell environment for the shell instance in which it is created.) You are going to create a function called hw , which stands for "hello world." Enter the following code at the CLI and press Enter . Then enter hw as you would any other shell command:

```
[ student @ testvm1 ~ ] $ hw (){ echo "Hi there kiddo" ; }
[ student @ testvm1 ~ ] $ hw
Hi there kiddo
[ student @ testvm1 ~ ] $
```

OK, so I am a little tired of the standard "Hello world" starter. Now, list all of the currently defined functions. There are a lot of them, so I am showing just the new hw function. When it is called from the command line or within a program, a function performs its programmed task and then exits and returns control to the calling entity, the command line, or the next Bash program statement in a script after the calling statement:

```
[ student @ testvm1 ~ ] $ declare -f | less
< snip >
hw ()
{
echo "Hi there kiddo"
}
< snip >
```

Remove that function because you do not need it anymore. You can do that with the unset command:

```
[ student @ testvm1 ~ ] $ unset -f hw ; hw
bash: hw: command not found
[ student @ testvm1 ~ ] $ 
```

Creating the Help function

Open the hello program in an editor and add the Help function below to the hello program code after the copyright statement but before the echo "Hello world!" statement. This Help function will display a short description of the program, a syntax diagram, and short descriptions of the available options. Add a call to the Help function to test it and some comment lines that provide a visual demarcation between the functions and the main portion of the program:

```
##############################################################################
# Help #
##############################################################################
Help ()
{
# Display Help
echo "Add description of the script functions here."
echo
echo "Syntax: scriptTemplate [-g|h|v|V]"
echo "options:"
echo "g Print the GPL license notification."
echo "h Print this Help."
echo "v Verbose mode."
echo "V Print software version and exit."
echo
}

##############################################################################
##############################################################################
# Main program #
##############################################################################
##############################################################################

Help
echo "Hello world!"
```

The options described in this Help function are typical for the programs I write, although none are in the code yet. Run the program to test it:

```
[ student @ testvm1 ~ ] $ . / hello
Add description of the script functions here.

Syntax: scriptTemplate [ -g | h | v | V ]
options:
g Print the GPL license notification.
h Print this Help.
v Verbose mode.
V Print software version and exit.

Hello world !
[ student @ testvm1 ~ ] $
```

Because you have not added any logic to display Help only when you need it, the program will always display the Help. Since the function is working correctly, read on to add some logic to display the Help only when the -h option is used when you invoke the program at the command line.

Handling options

A Bash script's ability to handle command-line options such as -h gives some powerful capabilities to direct the program and modify what it does. In the case of the -h option, you want the program to print the Help text to the terminal session and then quit without running the rest of the program. The ability to process options entered at the command line can be added to the Bash script using the while command (see *How to program with Bash: Loops* to learn more about while) in conjunction with the getops and case commands.

The getops command reads any and all options specified at the command line and creates a list of those options. In the code below, the while command loops through the list of options by setting the variable $options for each. The case statement is used to evaluate each option in turn and execute the statements in the corresponding stanza. The while statement will continue to evaluate the list of options until they have all been processed or it encounters an exit statement, which terminates the program.

Be sure to delete the Help function call just before the echo "Hello world!" statement so that the main body of the program now looks like this:

```
###############################################################################
###############################################################################
# Main program #
###############################################################################
###############################################################################
###############################################################################
# Process the input options. Add options as needed. #
###############################################################################
# Get the options
while getopts ":h" option; do
case $option in
h ) # display Help
Help
exit ;;
esac
done

echo "Hello world!"
```

Notice the double semicolon at the end of the exit statement in the case option for -h . This is required for each option added to this case statement to delineate the end of each option.

Testing

Testing is now a little more complex. You need to test your program with a number of different options -- and no options -- to see how it responds. First, test with no options to ensure that it prints "Hello world!" as it should:

```
[ student @ testvm1 ~ ] $ . / hello
Hello world !
```

That works, so now test the logic that displays the Help text:

```
[ student @ testvm1 ~ ] $ . / hello -h
Add description of the script functions here.

Syntax: scriptTemplate [ -g | h | t | v | V ]
options:
g Print the GPL license notification.
h Print this Help.
v Verbose mode.
V Print software version and exit.
```

That works as expected, so try some testing to see what happens when you enter some unexpected options:

```
[ student @ testvm1 ~ ] $ . / hello -x
Hello world !
[ student @ testvm1 ~ ] $ . / hello -q
Hello world !
[ student @ testvm1 ~ ] $ . / hello -lkjsahdf
Add description of the script functions here.

Syntax: scriptTemplate [ -g | h | t | v | V ]
options:
g Print the GPL license notification.
h Print this Help.
v Verbose mode.
V Print software version and exit.

[ student @ testvm1 ~ ] $
```

The program just ignores any options without specific responses without generating any errors. But notice the last entry (with -lkjsahdf for options): because there is an h in the list of options, the program recognizes it and prints the Help text. This testing has shown that the program doesn't have the ability to handle incorrect input and terminate the program if any is detected.

You can add another case stanza to the case statement to match any option that doesn't have an explicit match. This general case will match anything you have not provided a specific match for. The case statement now looks like this, with the catch-all match of \? as the last case. Any additional specific cases must precede this final one:

```
while getopts ":h" option; do
case $option in
h ) # display Help
Help
exit ;;
\? ) # incorrect option
echo "Error: Invalid option"
exit ;;
esac
done
```

Test the program again using the same options as before and see how it works now.

Where you are

You have accomplished a good amount in this article by adding the capability to process command-line options and a Help procedure. Your Bash script now looks like this:

```
#!/usr/bin/bash
###############################################################################
# scriptTemplate #
# #
# Use this template as the beginning of a new program. Place a short #
# description of the script here. #
# #
# Change History #
# 11/11/2019 David Both Original code. This is a template for creating #
```

```
# new Bash shell scripts. #
# Add new history entries as needed. #
# #
# #
################################################################################
################################################################################
################################################################################
# #
# Copyright (C) 2007, 2019 David Both #
# LinuxGeek46@both.org #
# #
# This program is free software; you can redistribute it and/or modify #
# it under the terms of the GNU General Public License as published by #
# the Free Software Foundation; either version 2 of the License, or #
# (at your option) any later version. #
# #
# This program is distributed in the hope that it will be useful, #
# but WITHOUT ANY WARRANTY; without even the implied warranty of #
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the #
# GNU General Public License for more details. #
# #
# You should have received a copy of the GNU General Public License #
# along with this program; if not, write to the Free Software #
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA #
# #
################################################################################
################################################################################
################################################################################

################################################################################
# Help #
################################################################################
Help ()
{
# Display Help
echo "Add description of the script functions here."
echo
echo "Syntax: scriptTemplate [-g|h|t|v|V]"
echo "options:"
echo "g Print the GPL license notification."
echo "h Print this Help."
echo "v Verbose mode."
echo "V Print software version and exit."
echo
}

################################################################################
################################################################################
# Main program #
################################################################################
################################################################################
################################################################################
# Process the input options. Add options as needed. #
################################################################################
# Get the options
while getopts ":h" option; do
case $option in
h ) # display Help
Help
exit ;;
\? ) # incorrect option
echo "Error: Invalid option"
exit ;;
esac
done

echo "Hello world!"
```

Be sure to test this version of the program very thoroughly. Use random inputs and see what happens. You should also try testing valid and invalid options without using the dash (-) in front.

Next time

In this article, you added a Help function as well as the ability to process command-line options to display it selectively. The program is getting a little more complex, so testing is becoming more important and requires more test paths in order to be complete.

The next article will look at initializing variables and doing a bit of sanity checking to ensure that the program will run under the correct set of conditions.

[Jul 12, 2020] Navigating the Bash shell with pushd and popd - Opensource.com

Notable quotes:

"... directory stack ..."

Jul 12, 2020 | opensource.com

Navigating the Bash shell with pushd and popd Pushd and popd are the fastest navigational commands you've never heard of. 07 Aug 2019 Seth Kenlon (Red Hat) Feed 71 up 7 comments Image by : Opensource.com x **Subscribe now**

Get the highlights in your inbox every week.

https://opensource.com/eloqua-embedded-email-capture-block.html?offer_id=70160000000QzXNAA0

The pushd and popd commands are built-in features of the Bash shell to help you "bookmark" directories for quick navigation between locations on your hard drive. You might already feel that the terminal is an impossibly fast way to navigate your computer; in just a few key presses, you can go anywhere on your hard drive, attached storage, or network share. But that speed can break down when you find yourself going back and forth between directories, or when you get "lost" within your filesystem. Those are precisely the problems pushd and popd can help you solve.

pushd

At its most basic, pushd is a lot like cd . It takes you from one directory to another. Assume you have a directory called one , which contains a subdirectory called two , which contains a subdirectory called three , and so on. If your current working directory is one , then you can move to two or three or anywhere with the cd command:

$ pwd
one
$ cd two / three
$ pwd
three

You can do the same with pushd :

$ pwd
one
$ pushd two / three
~ / one / two / three ~ / one
$ pwd
three

The end result of pushd is the same as cd , but there's an additional intermediate result: pushd echos your destination directory and your point of origin. This is your ***directory stack*** , and it is what makes pushd unique.

Stacks

A stack, in computer terminology, refers to a collection of elements. In the context of this command, the elements are directories you have recently visited by using the pushd command. You can think of it as a history or a breadcrumb trail.

You can move all over your filesystem with pushd ; each time, your previous and new locations are added to the stack:

$ pushd four
~ / one / two / three / four ~ / one / two / three ~ / one
$ pushd five
~ / one / two / three / four / five ~ / one / two / three / four ~ / one / two / three ~ / one **Navigating the stack**

Once you've built up a stack, you can use it as a collection of bookmarks or fast-travel waypoints. For instance, assume that during a session you're doing a lot of work within the ~/one/two/three/four/five directory structure of this example. You know you've been to one recently, but you can't remember where it's located in your pushd stack. You can view your stack with the +0 (that's a plus sign followed by a zero) argument, which tells pushd not to change to any directory in your stack, but also prompts pushd to echo your current stack:

$ pushd + 0
~ / one / two / three / four ~ / one / two / three ~ / one ~ / one / two / three / four / five

Alternatively, you can view the stack with the dirs command, and you can see the index number for each directory by using the -v option:

$ dirs -v
0 ~ / one / two / three / four
1 ~ / one / two / three
2 ~ / one
3 ~ / one / two / three / four / five

The first entry in your stack is your current location. You can confirm that with pwd as usual:

$ pwd
~ / one / two / three / four

Starting at 0 (your current location and the first entry of your stack), the ***second*** element in your stack is ~/one , which is your desired destination. You can move forward in your stack using the +2 option:

$ pushd + 2
~ / one ~ / one / two / three / four / five ~ / one / two / three / four ~ / one / two / three
$ pwd
~ / one

This changes your working directory to ~/one and also has shifted the stack so that your new location is at the front.

You can also move backward in your stack. For instance, to quickly get to ~/one/two/three given the example output, you can move back by one, keeping in mind that pushd starts with 0:

$ pushd -0
~ / one / two / three ~ / one ~ / one / two / three / four / five ~ / one / two / three / four **Adding to the stack**

You can continue to navigate your stack in this way, and it will remain a static listing of your recently visited directories. If you want to add a directory, just provide the directory's path. If a directory is new to the stack, it's added to the list just as you'd expect:

$ pushd / tmp
/ tmp ~ / one / two / three ~ / one ~ / one / two / three / four / five ~ / one / two / three / four

But if it already exists in the stack, it's added a second time:

$ pushd ~ / one
~ / one / tmp ~ / one / two / three ~ / one ~ / one / two / three / four / five ~ / one / two / three / four

While the stack is often used as a list of directories you want quick access to, it is really a true history of where you've been. If you don't want a directory added redundantly to the stack, you must use the +N and -N notation.

Removing directories from the stack

Your stack is, obviously, not immutable. You can add to it with pushd or remove items from it with popd .

For instance, assume you have just used pushd to add ~/one to your stack, making ~/one your current working directory. To remove the first (or "zeroeth," if you prefer) element:

$ pwd
~ / one
$ popd + 0
/ tmp ~ / one / two / three ~ / one ~ / one / two / three / four / five ~ / one / two / three / four
$ pwd
~ / one

Of course, you can remove any element, starting your count at 0:

$ pwd ~ / one
$ popd + 2
/ tmp ~ / one / two / three ~ / one / two / three / four / five ~ / one / two / three / four
$ pwd ~ / one

You can also use popd from the back of your stack, again starting with 0. For example, to remove the final directory from your stack:

$ popd -0
/ tmp ~ / one / two / three ~ / one / two / three / four / five

When used like this, popd does not change your working directory. It only manipulates your stack.

Navigating with popd

The default behavior of popd , given no arguments, is to remove the first (zeroeth) item from your stack and make the next item your current working directory.

This is most useful as a quick-change command, when you are, for instance, working in two different directories and just need to duck away for a moment to some other location. You don't have to think about your directory stack if you don't need an elaborate history:

$ pwd
~ / one
$ pushd ~ / one / two / three / four / five
$ popd
$ pwd
~ / one

You're also not required to use pushd and popd in rapid succession. If you use pushd to visit a different location, then get distracted for three hours chasing down a bug or doing research, you'll find your directory stack patiently waiting (unless you've ended your terminal session):

$ pwd ~ / one
$ pushd / tmp
$ cd { / etc, / var, / usr } ; sleep 2001
[...]
$ popd
$ pwd
~ / one **Pushd and popd in the real world**

The pushd and popd commands are surprisingly useful. Once you learn them, you'll find excuses to put them to good use, and you'll get familiar with the concept of the directory stack. Getting comfortable with pushd was what helped me understand git stash , which is entirely unrelated to pushd but similar in conceptual intangibility.

Using pushd and popd in shell scripts can be tempting, but generally, it's probably best to avoid them. They aren't portable outside of Bash and Zsh, and they can be obtuse when you're re-reading a script (pushd +3 is less clear than cd $HOME/$DIR/$TMP or similar).

Aside from these warnings, if you're a regular Bash or Zsh user, then you can and should try pushd and popd . Bash prompt tips and tricks Here are a few hidden treasures you can use to customize your Bash prompt. Dave Neary (Red Hat) **Topics** Bash Linux Command line **About the author** Seth Kenlon - Seth Kenlon is an independent multimedia artist, free culture advocate, and UNIX geek. He has worked in the film and computing industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project, http://slackermedia.info More about me **Recommended reading**
Add videos as wallpaper on your Linux desktop
Use systemd timers instead of cronjobs
Why I stick with xterm
Customizing my Linux terminal with tmux and Git
Back up your phone's storage with this Linux utility
Read and write data from anywhere with redirection in the Linux terminal **7 Comments**

matt on 07 Aug 2019

> Thank you for the write up for pushd and popd. I gotta remember to use these when I'm jumping around directories a lot. I got a hung up on a pushd example because my development work using arrays differentiates between the index and the count. In my experience, a zero-based array of A, B, C; C has an index of 2 and also is the third element. C would not be considered the second element cause that would be confusing it's index and it's count.

Seth Kenlon on 07 Aug 2019

> Interesting point, Matt. The difference between count and index had not occurred to me, but I'll try to internalise it. It's a great distinction, so thanks for bringing it up!

Greg Pittman on 07 Aug 2019

> This looks like a recipe for confusing myself.

Seth Kenlon on 07 Aug 2019

> It can be, but start out simple: use pushd to change to one directory, and then use popd to go back to the original. Sort of a single-use bookmark system.
>
> Then, once you're comfortable with pushd and popd, branch out and delve into the stack.
>
> A tcsh shell I used at an old job didn't have pushd and popd, so I used to have functions in my .cshrc to mimic just the back-and-forth use.

Jake on 07 Aug 2019

> "dirs" can be also used to view the stack. "dirs -v" helpfully numbers each directory with its index.

Seth Kenlon on 07 Aug 2019

> Thanks for that tip, Jake. I arguably should have included that in the article, but I wanted to try to stay focused on just the two {push,pop}d commands. Didn't occur to me to casually mention one use of dirs as you have here, so I've added it for posterity.
>
> There's so much in the Bash man and info pages to talk about!

other_Stu on 11 Aug 2019

> I use "pushd ." (dot for current directory) quite often. Like a working directory bookmark when you are several subdirectories deep somewhere, and need to cd to couple of other places to do some work or check something.
> And you can use the cd command with your DIRSTACK as well, thanks to tilde expansion.
> cd ~+3 will take you to the same directory as pushd +3 would.

[Jul 12, 2020] An introduction to parameter expansion in Bash - Opensource.com

Jul 12, 2020 | opensource.com

An introduction to parameter expansion in Bash Get started with this quick how-to guide on expansion modifiers that transform Bash variables and other parameters into powerful tools beyond simple value stores. 13 Jun 2017 James Pannacciulli Feed 366 up 4 comments Image by : Opensource.com x **Subscribe now**

Get the highlights in your inbox every week.

https://opensource.com/eloqua-embedded-email-capture-block.html?offer_id=70160000000QzXNAA0

In Bash, entities that store values are known as parameters. Their values can be strings or arrays with regular syntax, or they can be integers or associative arrays when special attributes are set with the **declare** built-in. There are three types of parameters: positional parameters, special parameters, and variables.

More Linux resources

- Linux commands cheat sheet
- Advanced Linux commands cheat sheet
- Free online course: RHEL Technical Overview
- Linux networking cheat sheet
- SELinux cheat sheet
- Linux common commands cheat sheet
- What are Linux containers?
- Our latest Linux articles

For the sake of brevity, this article will focus on a few classes of expansion methods available for string variables, though these methods apply equally to other types of parameters.

Variable assignment and unadulterated expansion

When assigning a variable, its name must be comprised solely of alphanumeric and underscore characters, and it may not begin with a numeral. There may be no spaces around the equal sign; the name must immediately precede it and the value immediately follow:

```
$ variable_1="my content"
```

Storing a value in a variable is only useful if we recall that value later; in Bash, substituting a parameter reference with its value is called expansion. To expand a parameter, simply precede the name with the **$** character, optionally enclosing the name in braces:

```
$ echo $variable_1 ${variable_1} my content my content
```

Crucially, as shown in the above example, expansion occurs before the command is called, so the command never sees the variable name, only the text passed to it as an argument that resulted from the expansion. Furthermore, parameter expansion occurs *before* word splitting; if the result of expansion contains spaces, the expansion should be quoted to preserve parameter integrity, if desired:

```
$ printf "%s\n" ${variable_1} my content $ printf "%s\n" "${variable_1}" my content
```

Parameter expansion modifiers

Parameter expansion goes well beyond simple interpolation, however. Inside the braces of a parameter expansion, certain operators, along with their arguments, may be placed after the name, before the closing brace. These operators may invoke conditional, subset, substring, substitution, indirection, prefix listing, element counting, and case modification expansion methods, modifying the result of the expansion. With the exception of the reassignment operators (**=** and **:=**), these operators only affect the expansion of the parameter without modifying the parameter's value for subsequent expansions.

About conditional, substring, and substitution parameter expansion operators Conditional parameter expansion

Conditional parameter expansion allows branching on whether the parameter is unset, empty, or has content. Based on these conditions, the parameter can be expanded to its value, a default value, or an alternate value; throw a customizable error; or reassign the parameter to a default value. The following table shows the conditional parameter expansions -- each row shows a parameter expansion using an operator to potentially modify the expansion, with the columns showing the result of that expansion given the parameter's status as indicated in the column headers. Operators with the ':' prefix treat parameters with empty values as if they were unset.

parameter expansion	*unset var*	*var=""*	*var="gnu"*
${var-default}	default	--	gnu
${var:-default}	default	default	gnu
${var+alternate}	--	alternate	alternate
${var:+alternate}	--	--	alternate
${var?error}	error	--	gnu
${var:?error}	error	error	gnu

The **=** and **:=** operators in the table function identically to **-** and **:-** , respectively, except that the **=** variants rebind the variable to the result of the expansion.

As an example, let's try opening a user's editor on a file specified by the **OUT_FILE** variable. If either the **EDITOR** environment variable or our **OUT_FILE** variable is not specified, we will have a problem. Using a conditional expansion, we can ensure that when the **EDITOR** variable is expanded, we get the specified value or at least a sane default:

```
$ echo ${EDITOR} /usr/bin/vi $ echo ${EDITOR:-$(which nano)} /usr/bin/vi $ unset EDITOR $ echo ${EDITOR:-$(which nano)} /usr/bin/nano
```

Building on the above, we can run the editor command and abort with a helpful error at runtime if there's no filename specified:

```
$ ${EDITOR:-$(which nano)} ${OUT_FILE:?Missing filename} bash: OUT_FILE: Missing filename
```

Substring parameter expansion

Parameters can be expanded to just part of their contents, either by offset or by removing content matching a pattern. When specifying a substring offset, a length may optionally be specified. If running Bash version 4.2 or greater, negative numbers may be used as offsets from the end of the string. Note the parentheses used around the negative offset, which ensure that Bash does not parse the expansion as having the conditional default expansion operator from above:

```
$ location="CA 90095" $ echo "Zip Code: ${location:3}" Zip Code: 90095 $ echo "Zip Code: ${location:(-5)}" Zip Code: 90095 $ echo "State: ${locatior
```

Another way to take a substring is to remove characters from the string matching a pattern, either from the left edge with the **#** and **##** operators or from the right edge with the **%** and **%** operators. A useful mnemonic is that **#** appears left of a comment and **%** appears right of a number. When the operator is doubled, it matches greedily, as opposed to the single version, which removes the most minimal set of characters matching the pattern.

var="open source"	
parameter expansion	*offset of 5 length of 4*
${var:offset}	source
${var:offset:length}	sour
pattern of *o?	
${var#pattern}	en source
${var##pattern}	rce
pattern of ?e*	
${var%pattern}	open sour
${var%pattern}	o

The pattern-matching used is the same as with filename globbing: * matches zero or more of any character, ? matches exactly one of any character, [...] brackets introduce a character class match against a single character, supporting negation (^), as well as the posix character classes, e.g. [[:alnum:]] . By excising characters from our string in this manner, we can take a substring without first knowing the offset of the data we need:

```
$ echo $PATH /usr/local/bin:/usr/bin:/bin $ echo "Lowest priority in PATH: ${PATH##*:}" Lowest priority in PATH: /bin $ echo "Everything except lowe
```

Substitution in parameter expansion

The same types of patterns are used for substitution in parameter expansion. Substitution is introduced with the / or // operators, followed by two arguments separated by another / representing the pattern and the string to substitute. The pattern matching is always greedy, so the doubled version of the operator, in this case, causes all matches of the pattern to be replaced in the variable's expansion, while the singleton version replaces only the leftmost.

var="free and open"	
parameter expansion	*pattern of [[:space:]] string of _*
${var/pattern/string}	free_and open
${var//pattern/string}	free_and_open

The wealth of parameter expansion modifiers transforms Bash variables and other parameters into powerful tools beyond simple value stores. At the very least, it is important to understand how parameter expansion works when reading Bash scripts, but I suspect that not unlike myself, many of you will enjoy the conciseness and expressiveness that these expansion modifiers bring to your scripts as well as your interactive sessions. **Topics** Linux **About the author** James Pannacciulli - James Pannacciulli is an advocate for software freedom & user autonomy with an MA in Linguistics. Employed as a Systems Engineer in Los Angeles, in his free time he occasionally gives talks on bash usage at various conferences. James likes his beers sour and his nettles stinging. More from James may be found on his home page . He has presented at conferences including SCALE ,...

[Jul 12, 2020] A sysadmin's guide to Bash by Maxim Burgerhout

Jul 12, 2020 | opensource.com

Use aliases

... ...

Make your root prompt stand out

... ...

Control your history

You probably know that when you press the Up arrow key in Bash, you can see and reuse all (well, many) of your previous commands. That is because those commands have been saved to a file called .bash_history in your home directory. That history file comes with a bunch of settings and commands that can be very useful.

First, you can view your entire recent command history by typing history , or you can limit it to your last 30 commands by typing history 30 . But that's pretty vanilla. You have more control over what Bash saves and how it saves it.

For example, if you add the following to your .bashrc, any commands that start with a space will not be saved to the history list:

```
HISTCONTROL=ignorespace
```

This can be useful if you need to pass a password to a command in plaintext. (Yes, that is horrible, but it still happens.)

If you don't want a frequently executed command to show up in your history, use:

```
HISTCONTROL=ignorespace:erasedups
```

With this, every time you use a command, all its previous occurrences are removed from the history file, and only the last invocation is saved to your history list.

A history setting I particularly like is the HISTTIMEFORMAT setting. This will prepend all entries in your history file with a timestamp. For example, I use:

```
HISTTIMEFORMAT="%F %T  "
```

When I type history 5 , I get nice, complete information, like this:

```
1009 2018 -06- 11 22 : 34 : 38 cat / etc / hosts
1010 2018 -06- 11 22 : 34 : 40 echo $foo
1011 2018 -06- 11 22 : 34 : 42 echo $bar
1012 2018 -06- 11 22 : 34 : 44 ssh myhost
1013 2018 -06- 11 22 : 34 : 55 vim .bashrc
```

That makes it a lot easier to browse my command history and find the one I used two days ago to set up an SSH tunnel to my home lab (which I forget again, and again, and again).

Best Bash practices

I'll wrap this up with my top 11 list of the best (or good, at least; I don't claim omniscience) practices when writing Bash scripts.

1. Bash scripts can become complicated and comments are cheap. If you wonder whether to add a comment, add a comment. If you return after the weekend and have to spend time figuring out what you were trying to do last Friday, you forgot to add a comment.

1. Wrap all your variable names in curly braces, like ${myvariable} . Making this a habit makes things like ${variable}_suffix possible and improves consistency throughout your scripts.

1. Do not use backticks when evaluating an expression; use the $() syntax instead. So use:

   ```
   for  file in $(ls); do
   ```

   not

   ```
   for  file in `ls`; do
   ```

   The former option is nestable, more easily readable, and keeps the general sysadmin population happy. Do not use backticks.

1. Consistency is good. Pick one style of doing things and stick with it throughout your script. Obviously, I would prefer if people picked the $() syntax over backticks and wrapped their variables in curly braces. I would prefer it if people used two or four spaces -- not tabs -- to indent, but even if you choose to do it

wrong, do it wrong consistently.

1. Use the proper shebang for a Bash script. As I'm writing Bash scripts with the intention of only executing them with Bash, I most often use #!/usr/bin/bash as my shebang. Do not use #!/bin/sh or #!/usr/bin/sh . Your script will execute, but it'll run in compatibility mode -- potentially with lots of unintended side effects. (Unless, of course, compatibility mode is what you want.)

1. When comparing strings, it's a good idea to quote your variables in if-statements, because if your variable is empty, Bash will throw an error for lines like these: if [${myvar} == "foo"] ; then
echo "bar"
fi And will evaluate to false for a line like this: if [" ${myvar} " == "foo"] ; then
echo "bar"
fi Also, if you are unsure about the contents of a variable (e.g., when you are parsing user input), quote your variables to prevent interpretation of some special characters and make sure the variable is considered a single word, even if it contains whitespace.

1. This is a matter of taste, I guess, but I prefer using the double equals sign (==) even when comparing strings in Bash. It's a matter of consistency, and even though -- for string comparisons only -- a single equals sign will work, my mind immediately goes "single equals is an assignment operator!"

1. Use proper exit codes. Make sure that if your script fails to do something, you present the user with a written failure message (preferably with a way to fix the problem) and send a non-zero exit code: # we have failed
echo "Process has failed to complete, you need to manually restart the whatchamacallit"
exit 1 This makes it easier to programmatically call your script from yet another script and verify its successful completion.

1. Use Bash's built-in mechanisms to provide sane defaults for your variables or throw errors if variables you expect to be defined are not defined: # this sets the value of $myvar to redhat, and prints 'redhat'
echo ${myvar:=redhat} # this throws an error reading 'The variable myvar is undefined, dear reader' if $myvar is undefined
${myvar:?The variable myvar is undefined, dear reader}

1. Especially if you are writing a large script, and especially if you work on that large script with others, consider using the local keyword when defining variables inside functions. The local keyword will create a local variable, that is one that's visible only within that function. This limits the possibility of clashing variables.

1. Every sysadmin must do it sometimes: debug something on a console, either a real one in a data center or a virtual one through a virtualization platform. If you have to debug a script that way, you will thank yourself for remembering this: Do not make the lines in your scripts too long!

   On many systems, the default width of a console is still 80 characters. If you need to debug a script on a console and that script has very long lines, you'll be a sad panda. Besides, a script with shorter lines -- the default is still 80 characters -- is a lot easier to read and understand in a normal editor, too!

I truly love Bash. I can spend hours writing about it or exchanging nice tricks with fellow enthusiasts. Make sure you drop your favorites in the comments!

[Jul 12, 2020] My favorite Bash hacks

Jan 09, 2020 | opensource.com

Get the highlights in your inbox every week.

When you work with computers all day, it's fantastic to find repeatable commands and tag them for easy use later on. They all sit there, tucked away in ~/.bashrc (or ~/.zshrc for Zsh users), waiting to help improve your day!

In this article, I share some of my favorite of these helper commands for things I forget a lot, in hopes that they will save you, too, some heartache over time.

Say when it's over

When I'm using longer-running commands, I often multitask and then have to go back and check if the action has completed. But not anymore, with this helpful invocation of say (this is on MacOS; change for your local equivalent):

```
function looooooooong {
START=$(date +%s.%N)
$*
EXIT_CODE=$?
END=$(date +%s.%N)
DIFF=$(echo "$END - $START" | bc)
RES=$(python -c "diff = $DIFF; min = int(diff / 60); print('%s min' % min)")
result="$1 completed in $RES, exit code $EXIT_CODE."
echo -e "\n ⏰ $result"
( say -r 250 $result 2>&1 > /dev/null & )
}
```

This command marks the start and end time of a command, calculates the minutes it takes, and speaks the command invoked, the time taken, and the exit code. I find this super helpful when a simple console bell just won't do.

... ...

There are many Docker commands, but there are even more docker compose commands. I used to forget the --rm flags, but not anymore with these useful aliases:

```
alias dc = "docker-compose"
alias dcr = "docker-compose run --rm"
alias dcb = "docker-compose run --rm --build"
```

gcurl helper for Google Cloud

This one is relatively new to me, but it's heavily documented . gcurl is an alias to ensure you get all the correct flags when using local curl commands with authentication headers when working with Google Cloud APIs.

Git and ~/.gitignore

I work a lot in Git, so I have a special section dedicated to Git helpers.

One of my most useful helpers is one I use to clone GitHub repos. Instead of having to run:

```
git clone git@github.com:org/repo /Users/glasnt/git/org/repo
```

I set up a clone function:

```
clone(){
echo Cloning $1 to ~/git/$1
cd ~/git
git clone git@github.com:$1 $1
cd $1
}
```

... ...

[Jul 09, 2020] My Favourite Secret Weapon strace

Jul 09, 2020 | zwischenzugs.com

Why strace ?

I'm often asked in my technical troubleshooting job to solve problems that development teams can't solve. Usually these do not involve knowledge of API calls or syntax, rather some kind of insight into what the right tool to use is, and why and how to use it. Probably because they're not taught in college, developers are often unaware that these tools exist, which is a shame, as playing with them can give a much deeper understanding of what's going on and ultimately lead to better code.

My favourite secret weapon in this path to understanding is strace.

strace (or its Solaris equivalents, trussdtruss is a tool that tells you which operating system (OS) calls your program is making.

An OS call (or just "system call") is your program asking the OS to provide some service for it. Since this covers a lot of the things that cause problems not directly to do with the domain of your application development (I/O, finding files, permissions etc) its use has a very high hit rate in resolving problems out of developers' normal problem space.

Usage Patterns

strace is useful in all sorts of contexts. Here's a couple of examples garnered from my experience.

My Netcat Server Won't Start!

Imagine you're trying to start an executable, but it's failing silently (no log file, no output at all). You don't have the source, and even if you did, the source code is neither readily available, nor ready to compile, nor readily comprehensible.

Simply running through strace will likely give you clues as to what's gone on.

```
$  nc -l localhost 80
nc: Permission denied
```

Let's say someone's trying to run this and doesn't understand why it's not working (let's assume manuals are unavailable).

Simply put strace at the front of your command. Note that the following output has been heavily edited for space reasons (deep breath):

```
$ strace nc -l localhost 80
execve("/bin/nc", ["nc", "-l", "localhost", "80"], [/* 54 vars */]) = 0
brk(0)                                  = 0x1e7a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f751c9c0000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/usr/local/lib/tls/x86_64/libglib-2.0.so.0", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/usr/local/lib/tls/x86_64", 0x7fff5686c240) = -1 ENOENT (No such file or directory)
[...]
open("libglib-2.0.so.0", O_RDONLY)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=179820, ...}) = 0
mmap(NULL, 179820, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f751c994000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libglib-2.0.so.0", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\3>\1\320k\1"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=975080, ...}) = 0
mmap(NULL, 3072520, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f751c4b3000
mprotect(0x7f751c5a0000, 2093056, PROT_NONE) = 0
mmap(0x7f751c79f000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xec000) = 0x7f751c79f000
mmap(0x7f751c7a1000, 520, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f751c7a1000
close(3)                                = 0
open("/usr/local/lib/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
[...]
mmap(NULL, 179820, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f751c994000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libnss_files.so.2", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\3>\1\20\""..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=51728, ...}) = 0
mmap(NULL, 2148104, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f751b8b0000
mprotect(0x7f751b8bc000, 2093056, PROT_NONE) = 0
mmap(0x7f751babb000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xb000) = 0x7f751babb000
close(3)                                = 0
mprotect(0x7f751babb000, 4096, PROT_READ) = 0
munmap(0x7f751c994000, 179820)          = 0
open("/etc/hosts", O_RDONLY|O_CLOEXEC)  = 3
fcntl(3, F_GETFD)                       = 0x1 (flags FD_CLOEXEC)
fstat(3, {st_mode=S_IFREG|0644, st_size=315, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f751c9bf000
read(3, "127.0.0.1\tlocalhost\n127.0.1.1\tal"..., 4096) = 315
read(3, "", 4096)                       = 0
close(3)                                = 0
munmap(0x7f751c9bf000, 4096)            = 0
open("/etc/gai.conf", O_RDONLY)         = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3343, ...}) = 0
fstat(3, {st_mode=S_IFREG|0644, st_size=3343, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f751c9bf000
read(3, "# Configuration for getaddrinfo("..., 4096) = 3343
read(3, "", 4096)                       = 0
close(3)                                = 0
```

```
munmap(0x7f751c9bf000, 4096)            = 0
futex(0x7f751c4af460, FUTEX_WAKE_PRIVATE, 2147483647) = 0
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
getsockname(3, {sa_family=AF_INET, sin_port=htons(58567), sin_addr=inet_addr("127.0.0.1")}, [16]) = 0
close(3)                                = 0
socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET6, sin6_port=htons(80), inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, 28) = 0
getsockname(3, {sa_family=AF_INET6, sin6_port=htons(42803), inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, [28]) =
close(3)                                = 0
socket(PF_INET6, SOCK_STREAM, IPPROTO_TCP) = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET6, sin6_port=htons(80), inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, 28) = -1 EACCES (
close(3)                                = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EACCES (Permission denied)
close(3)                                = 0
write(2, "nc: ", 4nc: )                 = 4
write(2, "Permission denied\n", 18Permission denied
)     = 18
exit_group(1)                           = ?
```

To most people that see this flying up their terminal this initially looks like gobbledygook, but it's really quite easy to parse when a few things are explained.

For each line:

- the first entry on the left is the system call being performed
- the bit in the parentheses are the arguments to the system call
- the right side of the equals sign is the return value of the system call

```
open("/etc/gai.conf", O_RDONLY)         = 3
```

Therefore for this particular line, the system call is open , the arguments are the string /etc/gai.conf and the constant O_RDONLY , and the return value was 3 .

How to make sense of this?

Some of these system calls can be guessed or enough can be inferred from context. Most readers will figure out that the above line is the attempt to open a file with read-only permission.

In the case of the above failure, we can see that before the program calls exit_group, there is a couple of calls to bind that return "Permission denied":

```
bind(3, {sa_family=AF_INET6, sin6_port=htons(80), inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, 28) = -1 EACCES (
close(3)                                = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EACCES (Permission denied)
close(3)                                = 0
write(2, "nc: ", 4nc: )                 = 4
write(2, "Permission denied\n", 18Permission denied
)     = 18
exit_group(1)                           = ?
```

We might therefore want to understand what "bind" is and why it might be failing.

You need to get a copy of the system call's documentation. On ubuntu and related distributions of linux, the documentation is in the manpages-dev package, and can be invoked by eg man 2 bind (I just used strace to determine which file man 2 bind opened and then did a dpkg -S to determine from which package it came!). You can also look up online if you have access, but if you can auto-install via a package manager you're more likely to get docs that match your installation.

Right there in my man 2 bind page it says:

```
ERRORS
EACCES The address is protected, and the user is not the superuser.
```

So there is the answer – we're trying to bind to a port that can only be bound to if you are the super-user.

My Library Is Not Loading!

Imagine a situation where developer A's perl script is working fine, but not on developer B's identical one is not (again, the output has been edited).
In this case, we strace the output on developer B's computer to see how it's working:

```
$ strace perl a.pl
execve("/usr/bin/perl", ["perl", "a.pl"], [/* 57 vars */]) = 0
brk(0)                                  = 0xa8f000
[...]fcntl(3, F_SETFD, FD_CLOEXEC)          = 0
fstat(3, {st_mode=S_IFREG|0664, st_size=14, ...}) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
brk(0xad1000)                           = 0xad1000
read(3, "use blahlib;\n\n", 4096)       = 14
stat("/space/myperllib/blahlib.pmc", 0x7fffbaf7f3d0) = -1 ENOENT (No such file or directory)
stat("/space/myperllib/blahlib.pm", {st_mode=S_IFREG|0644, st_size=7692, ...}) = 0
open("/space/myperllib/blahlib.pm", O_RDONLY) = 4
ioctl(4, SNDCTL_TMR_TIMEBASE or TCGETS, 0x7fffbaf7f090) = -1 ENOTTY (Inappropriate ioctl for device)
[...]mmap(0x7f4c45ea8000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 5, 0x4000) = 0x7f4c45ea8000
close(5)                                = 0
mprotect(0x7f4c45ea8000, 4096, PROT_READ) = 0
brk(0xb55000)                           = 0xb55000
read(4, "swrite($_[0], $_[1], $_[2], $_[3"..., 4096) = 3596
brk(0xb77000)                           = 0xb77000
read(4, "", 4096)                       = 0
close(4)                                = 0
read(3, "", 4096)                       = 0
```

```
close(3)                = 0
exit_group(0)           = ?
```

We observe that the file is found in what looks like an unusual place.

```
open("/space/myperllib/blahlib.pm", O_RDONLY) = 4
```

Inspecting the environment, we see that:

```
$ env | grep myperl
PERL5LIB=/space/myperllib
```

So the solution is to set the same env variable before running:

```
export PERL5LIB=/space/myperllib
```

Get to know the internals bit by bit

If you do this a lot, or idly run strace on various commands and peruse the output, you can learn all sorts of things about the internals of your OS. If you're like me, this is a great way to learn how things work. For example, just now I've had a look at the file /etc/gai.conf , which I'd never come across before writing this.

Once your interest has been piqued, I recommend getting a copy of "Advanced Programming in the Unix Environment" by Stevens & Rago, and reading it cover to cover. Not all of it will go in, but as you use strace more and more, and (hopefully) browse C code more and more understanding will grow.

Gotchas

If you're running a program that calls other programs, it's important to run with the -f flag, which "follows" child processes and straces them. -ff creates a separate file with the pid suffixed to the name.

If you're on solaris, this program doesn't exist – you need to use truss instead.

Many production environments will not have this program installed for security reasons. strace doesn't have many library dependencies (on my machine it has the same dependencies as 'echo'), so if you have permission, (or are feeling sneaky) you can just copy the executable up.

Other useful tidbits

You can attach to running processes (can be handy if your program appears to hang or the issue is not readily reproducible) with -p .

If you're looking at performance issues, then the time flags (-t , -tt , -ttt , and -T) can help significantly.

vasudevram February 11, 2018 at 5:29 pm

Interesting post. One point: The errors start earlier than what you said.There is a call to access() near the top of the strace output, which fails:

access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)

vasudevram February 11, 2018 at 5:29 pm

I guess that could trigger the other errors.

Benji Wiebe February 11, 2018 at 7:30 pm

A failed access or open system call is not usually an error in the context of launching a program. Generally it is merely checking if a config file exists.

vasudevram February 11, 2018 at 8:24 pm

>A failed access or open system call is not usually an error in the context of launching a program.

Yes, good point, that could be so, if the programmer meant to ignore the error, and if it was not an issue to do so.

>Generally it is merely checking if a config file exists.

The file name being access'ed is "/etc/ld.so.nohwcap" – not sure if it is a config file or not.

[Jul 07, 2020] The Missing Readline Primer by Ian Miell

Highly recommended!

This is from the book Learn Bash the Hard Way, available for $6.99.

Jul 07, 2020 | zwischenzugs.com

The Missing Readline Primer zwischenzugs Uncategorized April 23, 2019 7 Minutes

Readline is one of those technologies that is so commonly used many users don't realise it's there.

I went looking for a good primer on it so I could understand it better, but failed to find one. This is an attempt to write a primer that may help users get to grips with it, based on what I've managed to glean as I've tried to research and experiment with it over the years.

Bash Without Readline

First you're going to see what bash looks like without readline.

In your 'normal' bash shell, hit the TAB key twice. You should see something like this:

```
Display all 2335 possibilities? (y or n)
```

That's because bash normally has an 'autocomplete' function that allows you to see what commands are available to you if you tap tab twice.

Hit n to get out of that autocomplete.

Another useful function that's commonly used is that if you hit the up arrow key a few times, then the previously-run commands should be brought back to the command line.

Now type:

```
$ bash --noediting
```

The --noediting flag starts up bash *without* the readline library enabled.

If you hit TAB twice now you will see something different: the shell no longer 'sees' your tab and just sends a tab direct to the screen, moving your cursor along. Autocomplete has gone.

Autocomplete is just one of the things that the readline library gives you in the terminal. You might want to try hitting the up or down arrows as you did above to see that that no longer works as well.

Hit return to get a fresh command line, and exit your non-readline-enabled bash shell:

```
$ exit
```

Other Shortcuts

There are a great many shortcuts like autocomplete available to you if readline is enabled. I'll quickly outline four of the most commonly-used of these before explaining how you can find out more.

```
$ echo 'some command'
```

There should not be many surprises there. Now if you hit the 'up' arrow, you will see you can get the last command back on your line. If you like, you can re-run the command, but there are other things you can do with readline before you hit return.

If you hold down the ctrl key and then hit a at the same time your cursor will return to the start of the line. Another way of representing this 'multi-key' way of inputting is to write it like this: \C-a . This is one conventional way to represent this kind of input. The \C represents the control key, and the -a represents that the a key is depressed at the same time.

Now if you hit \C-e (ctrl and e) then your cursor has moved to the end of the line. I use these two dozens of times a day.

Another frequently useful one is \C-l , which clears the screen, but leaves your command line intact.

The last one I'll show you allows you to search your history to find matching commands while you type. Hit \C-r , and then type ec . You should see the echo command you just ran like this:

```
(reverse-i-search)`ec': echo echo
```

Then do it again, but keep hitting \C-r over and over. You should see all the commands that have `ec` in them that you've input before (if you've only got one echo command in your history then you will only see one). As you see them you are placed at that point in your history and you can move up and down from there or just hit return to re-run if you want.

There are many more shortcuts that you can use that readline gives you. Next I'll show you how to view these. **Using `bind` to Show Readline Shortcuts**

If you type:

```
$ bind -p
```

You will see a list of bindings that readline is capable of. There's a lot of them!

Have a read through if you're interested, but don't worry about understanding them all yet.

If you type:

```
$ bind -p | grep C-a
```

you'll pick out the 'beginning-of-line' binding you used before, and see the \C-a notation I showed you before.

As an exercise at this point, you might want to look for the \C-e and \C-r bindings we used previously.

If you want to look through the entirety of the bind -p output, then you will want to know that \M refers to the Meta key (which you might also know as the Alt key), and \e refers to the Esc key on your keyboard. The 'escape' key bindings are different in that you don't hit it and another key at the same time, rather you hit it, and then hit another key afterwards. So, for example, typing the Esc key, and then the ? key also tries to auto-complete the command you are typing. This is documented as:

```
"\e?": possible-completions
```

in the bind -p output.

Readline and Terminal Options

If you've looked over the possibilities that readline offers you, you might have seen the \C-r binding we looked at earlier:

```
"\C-r": reverse-search-history
```

You might also have seen that there is another binding that allows you to search forward through your history too:

```
"\C-s": forward-search-history
```

What often happens to me is that I hit \C-r over and over again, and then go too fast through the history and fly past the command I was looking for. In these cases I might try to hit \C-s to search forward and get to the one I missed.

Watch out though! Hitting \C-s to search forward through the history might well not work for you.

Why is this, if the binding is there and readline is switched on?

It's because something picked up the \C-s *before* it got to the readline library: the terminal settings.

The terminal program you are running in may have standard settings that do other things on hitting some of these shortcuts before readline gets to see it.

If you type:

```
$ stty -e
```

you should get output similar to this:

```
speed 9600 baud; 47 rows; 202 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -iutf8 -ignbrk brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts -dsrflow -dtrflow -mdmbuf
discard dsusp   eof     eol     eol2    erase   intr    kill    lnext
```

```
^O     ^Y     ^D     <undef> <undef> ^?     ^C     ^U     ^V
min    quit   reprint start  status  stop   susp   time   werase
1      ^\     ^R     ^Q     ^T     ^S     ^Z     0      ^W
```

You can see on the last four lines (discard dsusp [...]) there is a table of key bindings that your terminal will pick up before readline sees them. The ^ character (known as the 'caret') here represents the ctrl key that we previously represented with a \C .

If you think this is confusing I won't disagree. Unfortunately in the history of Unix and Linux documenters did not stick to one way of describing these key combinations.

If you encounter a problem where the terminal options seem to catch a shortcut key binding before it gets to readline, then you can use the stty program to unset that binding. In this case, we want to unset the 'stop' binding.

If you are in the same situation, type:

```
$ stty stop undef
```

Now, if you re-run stty -e , the last two lines might look like this:

```
[...]
min    quit   reprint start  status  stop   susp   time   werase
1      ^\     ^R     ^Q     ^T     <undef> ^Z     0      ^W
```

where the stop entry now has <undef> underneath it.

Strangely, for me C-r is also bound to 'reprint' above (^R).

But (on my terminals at least) that gets to readline without issue as I search up the history. Why this is the case I haven't been able to figure out. I suspect that reprint is ignored by modern terminals that don't need to 'reprint' the current line.

While we are looking at this table:

```
discard dsusp  eof    eol    eol2   erase  intr   kill   lnext
^O     ^Y     ^D     <undef> <undef> ^?     ^C     ^U     ^V
min    quit   reprint start  status  stop   susp   time   werase
1      ^\     ^R     ^Q     ^T     <undef> ^Z     0      ^W
```

it's worth noting a few other key bindings that are used regularly.

First, one you may well already be familiar with is \C-c , which interrupts a program, terminating it:

```
$ sleep 99
[[Hit \C-c]]
^C
$
```

Similarly, \C-z suspends a program, allowing you to 'foreground' it again and continue with the fg builtin.

```
$ sleep 10
[[ Hit \C-z]]
^Z
[1]+  Stopped                 sleep 10
$ fg
sleep 10
```

\C-d sends an 'end of file' character. It's often used to indicate to a program that input is over. If you type it on a bash shell, the bash shell you are in will close.

Finally, \C-w deletes the word before the cursor

These are the most commonly-used shortcuts that are picked up by the terminal before they get to the readline library.

Daz April 29, 2019 at 11:15 pm

Hi Ian,

What OS are you running because stty -e gives the following on Centos 6.x and Ubuntu 18.04.2

```
stty -e
stty: invalid argument '-e'
Try 'stty –help' for more information.
```

Reply

Leon May 14, 2019 at 5:12 am

`stty -a` works for me (Ubuntu 14)

yachris May 16, 2019 at 4:40 pm

You might want to check out the 'rlwrap' program. It allows you to have readline behavior on programs that don't natively support readline, but which have a 'type in a command' type interface. For instance, we use Oracle here (alas :-)) and the 'sqlplus' program, that lets you type SQL commands to an Oracle instance does not have anything like readline built into it, so you can't go back to edit previous commands. But running 'rlwrap sqlplus' gives me readline behavior in sqlplus! It's fantastic to have.

AriSweedler May 17, 2019 at 4:50 am

I was told to use this in a class, and I didn't understand what I did. One rabbit hole later, I was shocked and amazed at how advanced the readline library is. One thing I'd like to add is that you can write a '~/.inputrc' file and have those readline commands sourced at startup!

I do not know exactly when or how the inputrc is read.

Most of what I learned about inputrc stuff is from https://www.topbug.net/blog/2017/07/31/inputrc-for-humans/ .

Here is my inputrc, if anyone wants: https://github.com/AriSweedler/dotfiles/blob/master/.inputrc .

[Jul 07, 2020] More stupid Bash tricks- Variables, find, file descriptors, and remote operations - Enable Sysadmin by Valentin Bajrami

The first part is at Stupid Bash tricks- History, reusing arguments, files and directories, functions, and more - Enable Sysadmin

Jul 02, 2020 | www.redhat.com

These tips and tricks will make your Linux command line experience easier and more efficient.

Image

Photo by Jonathan Meyer from Pexels

More Linux resources

- Download Now: Linux Commands Cheat Sheet
- Advanced Linux Commands Cheat Sheet for Developers
- Download Red Hat Enterprise Linux Server 8 Trial
- Linux System Administration Skills Assessment

This blog post is the second of two covering some practical tips and tricks to get the most out of the Bash shell. In part one , I covered history, last argument, working with files and directories, reading files, and Bash functions. In this segment, I cover shell variables, find, file descriptors, and remote operations.

Use shell variables

The Bash variables are set by the shell when invoked. Why would I use hostname when I can use $HOSTNAME, or why would I use whoami when I can use $USER? Bash variables are very fast and do not require external applications.

These are a few frequently-used variables:

```
$PATH
$HOME
$USER
$HOSTNAME
$PS1
..
$PS4
```

Use the echo command to expand variables. For example, the $PATH shell variable can be expanded by running:

```
$> echo $PATH
```

[Download now: A sysadmin's guide to Bash scripting .]

Use the find command

The find command is probably one of the most used tools within the Linux operating system. It is extremely useful in interactive shells. It is also used in scripts. With find I can list files older or newer than a specific date, delete them based on that date, change permissions of files or directories, and so on.

Let's get more familiar with this command.

To list files older than 30 days, I simply run:

```
$> find /tmp -type f -mtime +30
```

To delete files older than 30 days, run:

```
$> find /tmp -type f -mtime +30 -exec rm -rf {} \;
```

or

```
$> find /tmp -type f -mtime +30 -exec rm -rf {} +
```

While the above commands will delete files older than 30 days, as written, they fork the rm command each time they find a file. This search can be written more efficiently by using xargs :

```
$> find /tmp -name '*.tmp' -exec printf '%s\0' {} \; | xargs -0 rm
```

I can use find to list sha256sum files only by running:

```
$> find . -type f -exec sha256sum {} +
```

And now to search for and get rid of duplicate .jpg files:

```
$> find . -type f -name '*.jpg' -exec sha256sum {} + | sort -uk1,1
```

Reference file descriptors

In the Bash shell, file descriptors (FDs) are important in managing the input and output of commands. Many people have issues understanding file descriptors correctly. Each process has three default file descriptors, namely:

Code	Meaning	Location	Description
0	Standard input	/dev/stdin	Keyboard, file, or some stream
1	Standard output	/dev/stdout	Monitor, terminal, display
2	Standard error	/dev/stderr	Non-zero exit codes are usually >FD2, display

Now that you know what the default FDs do, let's see them in action. I start by creating a directory named foo , which contains file1 .

```
$> ls foo/ bar/
ls: cannot access 'bar/': No such file or directory
foo/:
file1
```

The output *No such file or directory* goes to Standard Error (stderr) and is also displayed on the screen. I will run the same command, but this time use 2> to omit stderr:

```
$> ls foo/ bar/ 2>/dev/null
foo/:
file1
```

It is possible to send the output of foo to Standard Output (stdout) and to a file simultaneously, and ignore stderr. For example:

```
$> { ls foo bar | tee -a ls_out_file ;} 2>/dev/null
foo:
```

```
file1
```

Then:

```
$> cat ls_out_file
foo:
file1
```

The following command sends stdout to a file and stderr to /dev/null so that the error won't display on the screen:

```
$> ls foo/ bar/ >to_stdout 2>/dev/null
$> cat to_stdout
foo/:
file1
```

The following command sends stdout and stderr to the same file:

```
$> ls foo/ bar/ >mixed_output 2>&1
$> cat mixed_output
ls: cannot access 'bar/': No such file or directory
foo/:
file1
```

This is what happened in the last example, where stdout and stderr were redirected to the same file:

```
ls foo/ bar/ >mixed_output 2>&1
      |        |
      |        Redirect stderr to where stdout is sent
      |
      stdout is sent to mixed_output
```

Another short trick (> Bash 4.4) to send both stdout and stderr to the same file uses the ampersand sign. For example:

```
$> ls foo/ bar/ &>mixed_output
```

Here is a more complex redirection:

```
exec 3>&1 >write_to_file; echo "Hello World"; exec 1>&3 3>&-
```

This is what occurs:

- exec 3>&1 Copy stdout to file descriptor 3
- > write_to_file Make FD 1 to write to the file
- echo "Hello World" Go to file because FD 1 now points to the file
- exec 1>&3 Copy FD 3 back to 1 (swap)
- Three>&- Close file descriptor three (we don't need it anymore)

Often it is handy to group commands, and then send the Standard Output to a single file. For example:

```
$> { ls non_existing_dir; non_existing_command; echo "Hello world"; } 2> to_stderr
Hello world
```

As you can see, only "Hello world" is printed on the screen, but the output of the failed commands is written to the to_stderr file.

Execute remote operations

I use Telnet, netcat, Nmap, and other tools to test whether a remote service is up and whether I can connect to it. These tools are handy, but they aren't installed by default on all systems.

Fortunately, there is a simple way to test a connection without using external tools. To see if a remote server is running a web, database, SSH, or any other service, run:

```
$> timeout 3 bash -c '</dev/tcp/remote_server/remote_port' || echo "Failed to connect"
```

For example, to see if serverA is running the MariaDB service:

```
$> timeout 3 bash -c '</dev/tcp/serverA/3306' || echo "Failed to connect"
```

If the connection fails, the *Failed to connect* message is displayed on your screen.

Assume serverA is behind a firewall/NAT. I want to see if the firewall is configured to allow a database connection to serverA , but I haven't installed a database server yet. To emulate a database port (or any other port), I can use the following:

```
[serverA ~]# nc -l 3306
```

On clientA , run:

```
[clientA ~]# timeout 3 bash -c '</dev/tcp/serverA/3306' || echo "Failed"
```

While I am discussing remote connections, what about running commands on a remote server over SSH? I can use the following command:

```
$> ssh remotehost <<EOF  # Press the Enter key here
> ls /etc
EOF
```

This command runs ls /etc on the remote host.

I can also execute a local script on the remote host without having to copy the script over to the remote server. One way is to enter:

```
$> ssh remote_host 'bash -s' < local_script
```

Another example is to pass environment variables locally to the remote server and terminate the session after execution.

```
$> exec ssh remote_host ARG1=FOO ARG2=BAR 'bash -s' <<'EOF'
> printf %s\\n "$ARG1" "$ARG2"
> EOF
```

```
Password:
FOO
BAR
Connection to remote_host closed.
```

There are many other complex actions I can perform on the remote host.

Wrap up

There is certainly more to Bash than I was able to cover in this two-part blog post. I am sharing what I know and what I deal with daily. The idea is to familiarize you with a few techniques that could make your work less error-prone and more fun.

[Want to test your sysadmin skills? Take a skills assessment today.] **Valentin Bajrami**

Valentin is a system engineer with more than six years of experience in networking, storage, high-performing clusters, and automation. He is involved in different open source projects like bash, Fedora, Ceph, FreeBSD and is a member of Red Hat Accelerators. More about me

[Jul 05, 2020] Learn Bash the Hard Way by Ian Miell [Leanpub PDF-iPad-Kindle]

Highly recommended!

Jul 05, 2020 | leanpub.com

skeptic *5.0 out of 5 stars* Reviewed in the United States on July 2, 2020

A short (160 pages) book that covers some difficult aspects of bash needed to customize bash env.

Whether we want it or not, bash is the shell you face in Linux, and unfortunately, it is often misunderstood and misused. Issues related to creating your bash environment are not well addressed in existing books. This book fills the gap.

Few authors understand that bash is a complex, non-orthogonal language operating in a complex Linux environment. To make things worse, bash is an evolution of Unix shell and is a rather old language with warts and all. Using it properly as a programming language requires a serious study, not just an introduction to the basic concepts. Even issues related to customization of dotfiles are far from trivial, and you need to know quite a bit to do it properly.

At the same time, proper customization of bash environment does increase your productivity (or at least lessens the frustration of using Linux on the command line ;-)

The author covered the most important concepts related to this task, such as bash history, functions, variables, environment inheritance, etc. It is really sad to watch like the majorly of Linux users do not use these opportunities and forever remain on the "level zero" using default dotfiles with bare minimum customization.

This book contains some valuable tips even for a seasoned sysadmin (for example, the use of !& in pipes), and as such, is worth at least double of suggested price. It allows you intelligently customize your bash environment after reading just 160 pages and doing the suggested exercises.

Contents:

- **Foreword**
- **Learn Bash the Hard Way**
  - Introduction
  - Structure
- **Part I - Core Bash**
  - What is Bash?
  - Unpicking the Shell: Globbing and Quoting
  - Variables in Bash
  - Functions in Bash
  - Pipes and redirects
  - Scripts and Startups
- **Part II - Scripting Bash**
  - Command Substitution and Evaluation
  - Exit Codes
  - Tests
  - Loops
  - The set Builtin
  - Process Substitution
  - Subshells
  - Internal Field Separator
- **Part III - Bash Features**
  - Readline
  - Terminal Codes and Non-Standard Characters
  - The Prompt
  - Here Documents
  - History
  - Bash in Practice
- **Part IV - Advanced Bash**
  - Job Control
  - Traps and Signals
  - Advanced Variables
  - String Manipulation
  - Debugging Bash Scripts
  - Autocomplete
  - Example Bash Script
  - Finished!

[Jul 04, 2020] Eleven bash Tips You Might Want to Know by Ian Miell

Highly recommended!

Notable quotes:

"... Material here based on material from my book Learn Bash the Hard Way . Free preview available here . ..."

"... natively in bash ..."

Jul 04, 2020 | zwischenzugs.com

Here are some tips that might help you be more productive with bash.

1) ^x^y^

A gem I use all the time.

Ever typed anything like this?

```
$ grp somestring somefile
-bash: grp: command not found
```

Sigh. Hit 'up', 'left' until at the 'p' and type 'e' and return.

Or do this:

```
$ ^rp^rep^
grep 'somestring' somefile
$
```

One subtlety you may want to note though is:

```
$ grp rp somefile
$ ^rp^rep^
$ grep rp somefile
```

If you wanted rep to be searched for, then you'll need to dig into the man page and use a more powerful history command:

```
$ grp rp somefile
$ !!:gs/rp/rep
grep rep somefile
$
```

... ...

Material here based on material from my book
Learn Bash the Hard Way .
Free preview available here .

---

3) shopt vs set

This one bothered me for a while.

What's the difference between set and shopt ?

set s we saw before , but shopt s look very similar. Just inputting shopt shows a bunch of options:

```
$ shopt
cdable_vars     off
cdspell         on
checkhash       off
checkwinsize    on
cmdhist         on
compat31        off
dotglob         off
```

I found a set of answers here . Essentially, it looks like it's a consequence of bash (and other shells) being built on sh, and adding shopt as another way to set extra shell options. But I'm still unsure if you know the answer, let me know.

4) Here Docs and Here Strings

'Here docs' are files created inline in the shell.

The 'trick' is simple. Define a closing word, and the lines between that word and when it appears alone on a line become a file.

Type this:

```
$ cat > afile << SOMEENDSTRING
> here is a doc
> it has three lines
> SOMEENDSTRING alone on a line will save the doc
> SOMEENDSTRING
$ cat afile
here is a doc
it has three lines
SOMEENDSTRING alone on a line will save the doc
```

Notice that:

- the string could be included in the file if it was not 'alone' on the line
- the string SOMEENDSTRING is more normally END , but that is just convention

Lesser known is the 'here string':

```
$ cat > asd <<< 'This file has one line'
```

5) String Variable Manipulation

You may have written code like this before, where you use tools like sed to manipulate strings:

```
$ VAR='HEADERMy voice is my passwordFOOTER'
$ PASS="$(echo $VAR | sed 's/^HEADER(.*)FOOTER/1/')"
$ echo $PASS
```

But you may not be aware that this is possible ***natively in bash*** .

This means that you can dispense with lots of sed and awk shenanigans.

One way to rewrite the above is:

```
$ VAR='HEADERMy voice is my passwordFOOTER'
$ PASS="${VAR#HEADER}"
$ PASS="${PASS%FOOTER}"
$ echo $PASS
```

- The # means 'match and remove the following pattern from the start of the string'
- The % means 'match and remove the following pattern from the end of the string

The second method is twice as fast as the first on my machine. And (to my surprise), it was roughly the same speed as a similar python script .

If you want to use glob patterns that are greedy (see globbing here) then you double up:

```
$ VAR='HEADERMy voice is my passwordFOOTER'
$ echo ${VAR##HEADER*}
$ echo ${VAR%%*FOOTER}
```

6) Variable Defaults

These are very handy when you're knocking up scripts quickly.

If you have a variable that's not set, you can 'default' them by using this. Create a file called default.sh with these contents

```
#!/bin/bash
FIRST_ARG="${1:-no_first_arg}"
SECOND_ARG="${2:-no_second_arg}"
THIRD_ARG="${3:-no_third_arg}"
echo ${FIRST_ARG}
echo ${SECOND_ARG}
echo ${THIRD_ARG}
```

Now run chmod +x default.sh and run the script with ./default.sh first second .

Observer how the third argument's default has been assigned, but not the first two.

You can also assign directly with ${VAR: = defaultval} (equals sign, not dash) but note that this won't work with positional variables in scripts or functions. Try changing the above script to see how it fails.

7) Traps

The trap built-in can be used to 'catch' when a signal is sent to your script.

Here's an example I use in my own cheapci script:

```
function cleanup() {
   rm -rf "${BUILD_DIR}"
   rm -f "${LOCK_FILE}"
   # get rid of /tmp detritus, leaving anything accessed 2 days ago+
   find "${BUILD_DIR_BASE}"/* -type d -atime +1 | rm -rf
   echo "cleanup done"
}
trap cleanup TERM INT QUIT
```

Any attempt to CTRL-C , CTRL- or terminate the program using the TERM signal will result in cleanup being called first.

Be aware:

- Trap logic can get very tricky (eg handling signal race conditions)
- The KILL signal can't be trapped in this way

But mostly I've used this for 'cleanups' like the above, which serve their purpose.

8) Shell Variables

It's well worth getting to know the standard shell variables available to you . Here are some of my favourites:

RANDOM

Don't rely on this for your cryptography stack, but you can generate random numbers eg to create temporary files in scripts:

```
$ echo ${RANDOM}
16313
$ # Not enough digits?
$ echo ${RANDOM}${RANDOM}
113610703
$ NEWFILE=/tmp/newfile_${RANDOM}
$ touch $NEWFILE
```

REPLY

No need to give a variable name for read

```
$ read
my input
$ echo ${REPLY}
```

LINENO and SECONDS

Handy for debugging

```
$ echo ${LINENO}
115
```

```
$ echo ${SECONDS}; sleep 1; echo ${SECONDS}; echo $LINENO
174380
174381
116
```

Note that there are two 'lines' above, even though you used ; to separate the commands.

TMOUT

You can timeout reads, which can be really handy in some scripts

```
#!/bin/bash
TMOUT=5
echo You have 5 seconds to respond...
read
echo ${REPLY:-noreply}
```

... ...

10) Associative Arrays

Talking of moving to other languages, a rule of thumb I use is that if I need arrays then I drop bash to go to python (I even created a Docker container for a tool to help with this here).

What I didn't know until I read up on it was that you can have associative arrays in bash.

Type this out for a demo:

```
$ declare -A MYAA=([one]=1 [two]=2 [three]=3)
$ MYAA[one]="1"
$ MYAA[two]="2"
$ echo $MYAA
$ echo ${MYAA[one]}
$ MYAA[one]="1"
$ WANT=two
$ echo ${MYAA[$WANT]}
```

Note that this is only available in bashes 4.x+.

... ...

[Jul 02, 2020] 7 Bash history shortcuts you will actually use by Ian Miell

Highly recommended!

Notable quotes:

"... The "last argument" one: !$ ..."

"... The " n th argument" one: !:2 ..."

"... The "all the arguments": !* ..."

"... The "last but n " : !-2:$ ..."

"... The "get me the folder" one: !$:h ..."

"... I use "!*" for "all arguments". It doesn't have the flexibility of your approach but it's faster for my most common need. ..."

"... Provided that your shell is readline-enabled, I find it much easier to use the arrow keys and modifiers to navigate through history than type !:1 (or having to remeber what it means). ..."

Oct 02, 2019 | opensource.com

7 Bash history shortcuts you will actually use Save time on the command line with these essential Bash shortcuts. 02 Oct 2019 Ian 205 up 12 comments
Image by : Opensource.com x **Subscribe now**

Most guides to Bash history shortcuts exhaustively list every single one available. The problem with that is I would use a shortcut once, then glaze over as I tried out all the possibilities. Then I'd move onto my working day and completely forget them, retaining only the well-known !! trick I learned when I first started using Bash.

So most of them were never committed to memory.

More on Bash

- Bash cheat sheet
- An introduction to programming with Bash
- A sysadmin's guide to Bash scripting
- Latest Bash articles

This article outlines the shortcuts I *actually use* every day. It is based on some of the contents of my book, *Learn Bash the hard way* ; (you can read a preview of it to learn more).

When people see me use these shortcuts, they often ask me, "What did you do there!?" There's minimal effort or intelligence required, but to really learn them, I recommend using one each day for a week, then moving to the next one. It's worth taking your time to get them under your fingers, as the time you save will be significant in the long run.

1. ***The "last argument" one: !$***

If you only take one shortcut from this article, make it this one. It substitutes in the last argument of the last command into your line.

Consider this scenario:

$ mv / path / to / wrongfile / some / other / place
mv: cannot stat '/path/to/wrongfile' : No such file or directory

Ach, I put the wrongfile filename in my command. I should have put rightfile instead.

You might decide to retype the last command and replace wrongfile with rightfile completely. Instead, you can type:

$ mv / path / to / rightfile ! $
mv / path / to / rightfile / some / other / place

and the command will work.

There are other ways to achieve the same thing in Bash with shortcuts, but this trick of reusing the last argument of the last command is one I use the most.

2. *The " n th argument" one: !:2*

Ever done anything like this?

$ tar -cvf afolder afolder.tar
tar: failed to open

Like many others, I get the arguments to tar (and ln) wrong more often than I would like to admit.

tar_2x.png

When you mix up arguments like that, you can run:

$! : 0 ! : 1 ! : 3 ! : 2
tar -cvf afolder.tar afolder

and your reputation will be saved.

The last command's items are zero-indexed and can be substituted in with the number after the !: .

Obviously, you can also use this to reuse specific arguments from the last command rather than all of them.

3. *The "all the arguments": !**

Imagine I run a command like:

$ grep '(ping|pong)' afile

The arguments are correct; however, I want to match ping or pong in a file, but I used grep rather than egrep .

I start typing egrep , but I don't want to retype the other arguments. So I can use the !:1$ shortcut to ask for all the arguments to the previous command from the second one (remember they're zero-indexed) to the last one (represented by the **$** sign).

$ egrep ! : 1 -$
egrep '(ping|pong)' afile
ping

You don't need to pick 1-$; you can pick a subset like 1-2 or 3-9 (if you had that many arguments in the previous command).

4. *The "last but n " : !-2:$*

The shortcuts above are great when I know immediately how to correct my last command, but often I run commands *after* the original one, which means that the last command is no longer the one I want to reference.

For example, using the mv example from before, if I follow up my mistake with an ls check of the folder's contents:

$ mv / path / to / wrongfile / some / other / place
mv: cannot stat '/path/to/wrongfile' : No such file or directory
$ ls / path / to /
rightfile

I can no longer use the !$ shortcut.

In these cases, I can insert a - *n* : (where *n* is the number of commands to go back in the history) after the ! to grab the last argument from an older command:

$ mv / path / to / rightfile ! - 2 :$
mv / path / to / rightfile / some / other / place

Again, once you learn it, you may be surprised at how often you need it.

5. *The "get me the folder" one: !$:h*

This one looks less promising on the face of it, but I use it dozens of times daily.

Imagine I run a command like this:

$ tar -cvf system.tar / etc / system
tar: / etc / system: Cannot stat: No such file or directory
tar: Error exit delayed from previous errors.

The first thing I might want to do is go to the /etc folder to see what's in there and work out what I've done wrong.

I can do this at a stroke with:

$ cd ! $:h
cd / etc

This one says: "Get the last argument to the last command (/etc/system) and take off its last filename component, leaving only the /etc ."

6. The "the current line" one: !#:1

For years, I occasionally wondered if I could reference an argument on the current line before finally looking it up and learning it. I wish I'd done so a long time ago. I most commonly use it to make backup files:

$ cp / path / to / some / file ! #:1.bak
cp / path / to / some / file / path / to / some / file.bak

but once under the fingers, it can be a very quick alternative to

7. The "search and replace" one: !!:gs

This one searches across the referenced command and replaces what's in the first two / characters with what's in the second two.

Say I want to tell the world that my s key does not work and outputs f instead:

$ echo my f key doef not work
my f key doef not work

Then I realize that I was just hitting the f key by accident. To replace all the f s with s es, I can type:

$!! :gs / f / s /
echo my s key does not work
my s key does not work

It doesn't work only on single characters; I can replace words or sentences, too:

$!! :gs / does / did /
echo my s key did not work
my s key did not work **Test them out**

Just to show you how these shortcuts can be combined, can you work out what these toenail clippings will output?

$ ping ! #:0:gs/i/o
$ vi / tmp /! : 0 .txt
$ ls ! $:h
$ cd ! - 2 :h
$ touch ! $! - 3 :$!! ! $.txt
$ cat ! : 1 -$ **Conclusion**

Bash can be an elegant source of shortcuts for the day-to-day command-line user. While there are thousands of tips and tricks to learn, these are my favorites that I frequently put to use.

If you want to dive even deeper into all that Bash can teach you, pick up my book, *Learn Bash the hard way* or check out my online course, Master the Bash shell .

This article was originally posted on Ian's blog, Zwischenzugs.com , and is reused with permission.

Orr, August 25, 2019 at 10:39 pm

BTW – you inspired me to try and understand how to repeat the nth command entered on command line. For example I type 'ls' and then accidentally type 'clear'. !! will retype clear again but I wanted to retype ls instead using a shortcut.
Bash doesn't accept ':' so !:2 didn't work. !-2 did however, thank you!

Dima August 26, 2019 at 7:40 am

Nice article! Just another one cool and often used command: i.e.: !vi opens the last vi command with their arguments.

cbarrick on 03 Oct 2019

Your "current line" example is too contrived. Your example is copying to a backup like this:

```
$ cp /path/to/some/file !#:1.bak
```

But a better way to write that is with filename generation:

```
$ cp /path/to/some/file{,.bak}
```

That's not a history expansion though... I'm not sure I can come up with a good reason to use `!#:1`.

Darryl Martin August 26, 2019 at 4:41 pm

I seldom get anything out of these "bash commands you didn't know" articles, but you've got some great tips here. I'm writing several down and sticking them on my terminal for reference.

A couple additions I'm sure you know.

1. ***I use "!*" for "all arguments". It doesn't have the flexibility of your approach but it's faster for my most common need.***
2. I recently started using Alt-. as a substitute for "!$" to get the last argument. It expands the argument on the line, allowing me to modify it if necessary.

Ricardo J. Barberis on 06 Oct 2019

The problem with bash's history shorcuts for me is... that I never had the need to learn them.

Provided that your shell is readline-enabled, I find it much easier to use the arrow keys and modifiers to navigate through history than type !:1 (or having to remeber what it means).

Examples:

Ctrl+R for a Reverse search

Ctrl+A to move to the begnining of the line (Home key also)

Ctrl+E to move to the End of the line (End key also)

Ctrl+K to Kill (delete) text from the cursor to the end of the line

Ctrl+U to kill text from the cursor to the beginning of the line

Alt+F to move Forward one word (Ctrl+Right arrow also)

Alt+B to move Backward one word (Ctrl+Left arrow also)

etc.

YMMV of course.

[Jul 02, 2020] Some Relatively Obscure Bash Tips zwischenzugs

Jul 02, 2020 | zwischenzugs.com

2) |&

You may already be familiar with 2>&1 , which redirects standard error to standard output, but until I stumbled on it in the manual, I had no idea that you can pipe both standard output and standard error into the next stage of the pipeline like this:

```
if doesnotexist |& grep 'command not found' >/dev/null
then
  echo oops
fi
```

3) $''

This construct allows you to specify specific bytes in scripts without fear of triggering some kind of encoding problem. Here's a command that will grep through files looking for UK currency ('£') signs in hexadecimal recursively:

```
grep -r $'\xc2\xa3' *
```

You can also use octal:

```
grep -r $'\302\243' *
```

4) HISTIGNORE

If you are concerned about security, and ever type in commands that might have sensitive data in them, then this one may be of use.

This environment variable does ***not*** put the commands specified in your history file if you type them in. The commands are separated by colons:

```
HISTIGNORE="ls *:man *:history:clear:AWS_KEY*"
```

You have to specify the ***whole*** line, so a glob character may be needed if you want to exclude commands ***and*** their arguments or flags.

5) fc

If readline key bindings aren't under your fingers, then this one may come in handy.

It calls up the last command you ran, and places it into your preferred editor (specified by the EDITOR variable). Once edited, it re-runs the command.

6) ((i++))

If you can't be bothered with faffing around with variables in bash with the $[] construct, you can use the C-style compound command.

So, instead of:

```
A=1
A=$[$A+1]
echo $A
```

you can do:

```
A=1
((A++))
echo $A
```

which, especially with more complex calculations, might be easier on the eye.

7) caller

Another builtin bash command, caller gives context about the context of your shell's

SHLVL is a related shell variable which gives the level of depth of the calling stack.

This can be used to create stack traces for more complex bash scripts.

Here's a die function, adapted from the bash hackers' wiki that gives a stack trace up through the calling frames:

```
#!/bin/bash
die() {
 local frame=0
 ((FRAMELEVEL=SHLVL - frame))
 echo -n "${FRAMELEVEL}: "
 while caller $frame; do
  ((frame++));
  ((FRAMELEVEL=SHLVL - frame))
  if [[ ${FRAMELEVEL} -gt -1 ]]
  then
   echo -n "${FRAMELEVEL}: "
  fi
 done
 echo "$*"
 exit 1
}
```

which outputs:

```
3: 17 f1 ./caller.sh
2: 18 f2 ./caller.sh
1: 19 f3 ./caller.sh
0: 20 main ./caller.sh
*** an error occured ***
```

8) /dev/tcp/host/port

This one can be particularly handy if you find yourself on a container running within a Kubernetes cluster service mesh without any network tools (a frustratingly common experience).

Bash provides you with some virtual files which, when referenced, can create socket connections to other servers.

This snippet, for example, makes a web request to a site and returns the output.

```
exec 9<>/dev/tcp/brvtsdflnxhkzcmw.neverssl.com/80
echo -e "GET /online HTTP/1.1\r\nHost: brvtsdflnxhkzcmw.neverssl.com\r\n\r\n" >&9
cat <&9
```

The first line opens up file descriptor 9 to the host brvtsdflnxhkzcmw.neverssl.com on port 80 for reading and writing. Line two sends the raw HTTP request to that socket connection's file descriptor. The final line retrieves the response.

Obviously, this doesn't handle SSL for you, so its use is limited now that pretty much everyone is running on https, but when running from application containers within a service mesh can still prove invaluable, as requests there are initiated using HTTP.

9) Co-processes

Since version 4 of bash it has offered the capability to run named coprocesses.

It seems to be particularly well-suited to managing the inputs and outputs to other processes in a fine-grained way. Here's an annotated and trivial example:

```
coproc testproc (
  i=1
  while true
  do
    echo "iteration:${i}"
    ((i++))
    read -r aline
    echo "${aline}"
  done
)
```

This sets up the coprocess as a subshell with the name testproc .

Within the subshell, there's a never-ending while loop that counts its own iterations with the i variable. It outputs two lines: the iteration number, and a line read in from standard input.

After creating the coprocess, bash sets up an array with that name with the file descriptor numbers for the standard input and standard output. So this:

```
echo "${testproc[@]}"
```

in my terminal outputs:

```
63 60
```

Bash also sets up a variable with the process identifier for the coprocess, which you can see by echoing it:

```
echo "${testproc_PID}"
```

You can now input data to the standard input of this coprocess at will like this:

```
echo input1 >&"${testproc[1]}"
```

In this case, the command resolves to: echo input1 >&60 , and the >&[INTEGER] construct ensures the redirection goes to the coprocess's standard input.

Now you can read the output of the coprocess's two lines in a similar way, like this:

```
read -r output1a <&"${testproc[0]}"
read -r output1b <&"${testproc[0]}"
```

You might use this to create an expect -like script if you were so inclined, but it could be generally useful if you want to manage inputs and outputs. Named pipes are another way to achieve a similar result.

Here's a complete listing for those who want to cut and paste:

```
!/bin/bash
coproc testproc (
  i=1
  while true
  do
    echo "iteration:${i}"
    ((i++))
    read -r aline
    echo "${aline}"
  done
)
echo "${testproc[@]}"
echo "${testproc_PID}"
echo input1 >&"${testproc[1]}"
read -r output1a <&"${testproc[0]}"
read -r output1b <&"${testproc[0]}"
echo "${output1a}"
echo "${output1b}"
echo input2 >&"${testproc[1]}"
read -r output2a <&"${testproc[0]}"
read -r output2b <&"${testproc[0]}"
echo "${output2a}"
echo "${output2b}"
```

[Jul 02, 2020] Associative arrays in Bash by Seth Kenlon

Apr 02, 2020 | opensource.com

Originally from: Get started with Bash scripting for sysadmins - Opensource.com

Most shells offer the ability to create, manipulate, and query indexed arrays. In plain English, an indexed array is a list of things prefixed with a number. This list of things, along with their assigned number, is conveniently wrapped up in a single variable, which makes it easy to "carry" it around in your code.

Bash, however, includes the ability to create associative arrays and treats these arrays the same as any other array. An associative array lets you create lists of key and value pairs, instead of just numbered values.

The nice thing about associative arrays is that keys can be arbitrary:

```
$ declare -A userdata
$ userdata [ name ] =seth
$ userdata [ pass ] =8eab07eb620533b083f241ec4e6b9724
$ userdata [ login ] = ` date --utc + % s `
```

Query any key:

```
$ echo " ${userdata[name]} "
seth
$ echo " ${userdata[login]} "
1583362192
```

Most of the usual array operations you'd expect from an array are available.

Resources

- How to program with Bash: Syntax and tools
- How to program with Bash: Logical operators and shell expansions
- How to program with Bash: Loops

[Jul 01, 2020] Stupid Bash tricks- History, reusing arguments, files and directories, functions, and more by Valentin Bajrami

A moderately interesting example here is the example of changing sudo systemctl start into sudo systemctl stop via !!:s/status/start/

But it probably can be optimized so that you do not need to type start (it can be deleted as the last word). So you can try !0 stop instead

Jul 01, 2020 | www.redhat.com

See also Bash bang commands- A must-know trick for the Linux command line - Enable Sysadmin

Let's say I run the following command:

```
$> sudo systemctl status sshd
```

Bash tells me the sshd service is not running, so the next thing I want to do is start the service. I had checked its status with my previous command. That command was saved in history , so I can reference it. I simply run:

```
$> !!:s/status/start/
sudo systemctl start sshd
```

The above expression has the following content:

- !! - repeat the last command from history
- :s/status/start/ - substitute status with start

The result is that the sshd service is started.

Next, I increase the default HISTSIZE value from 500 to 5000 by using the following command:

```
$> echo "HISTSIZE=5000" >> ~/.bashrc && source ~/.bashrc
```

What if I want to display the last three commands in my history? I enter:

```
$> history 3
 1002  ls
 1003  tail audit.log
 1004  history 3
```

I run tail on audit.log by referring to the history line number. In this case, I use line 1003:

```
$> !1003
tail audit.log
```

Reference the last argument of the previous command

When I want to list directory contents for different directories, I may change between directories quite often. There is a nice trick you can use to refer to the last argument of the previous command. For example:

```
$> pwd
/home/username/

$> ls some/very/long/path/to/some/directory
foo-file bar-file baz-file
```

In the above example, /some/very/long/path/to/some/directory is the last argument of the previous command.

If I want to cd (change directory) to that location, I enter something like this:

```
$> cd $_

$> pwd
/home/username/some/very/long/path/to/some/directory
```

Now simply use a dash character to go back to where I was:

```
$> cd -
$> pwd
/home/username/
```

[Mar 05, 2020] How to tell if you're using a bash builtin in Linux

Mar 05, 2020 | www.networkworld.com

One quick way to determine whether the command you are using is a bash built-in or not is to use the command "command". Yes, the command is called "command". Try it with a -V (capital V) option like this:

```
$ command -V command
command is a shell builtin
$ command -V echo
echo is a shell builtin
$ command -V date
date is hashed (/bin/date)
```

When you see a "command is hashed" message like the one above, that means that the command has been put into a hash table for quicker lookup.

... ... How to tell what shell you're currently using

If you switch shells you can't depend on $SHELL to tell you what shell you're currently using because $SHELL is just an environment variable that is set when you log in and doesn't necessarily reflect your current shell. Try ps -p $$ instead as shown in these examples:

```
$ ps -p $$
  PID TTY          TIME CMD
18340 pts/0    00:00:00 bash    <==
$ /bin/dash
$ ps -p $$
  PID TTY          TIME CMD
19517 pts/0    00:00:00 dash    <==
```

Built-ins are extremely useful and give each shell a lot of its character. If you use some particular shell all of the time, it's easy to lose track of which commands are part of your shell and which are not.

Differentiating a shell built-in from a Linux executable requires only a little extra effort.

[Mar 05, 2020] Bash IDE - Visual Studio Marketplace

Notable quotes:

"... all your shell scripts ..."

Mar 05, 2020 | marketplace.visualstudio.com

Bash IDE

Visual Studio Code extension utilizing the bash language server , that is based on Tree Sitter and its grammar for Bash and supports explainshell integration.

Features

- [x] Jump to declaration
- [x] Find references
- [x] Code Outline & Show Symbols
- [x] Highlight occurrences
- [x] Code completion
- [x] Simple diagnostics reporting
- [x] Documentation for flags on hover
- [ ] Rename symbol

Configuration

To get documentation for flags on hover (thanks to explainshell), run the explainshell Docker container :

```
docker run --rm --name bash-explainshell -p 5000:5000 chrismwendt/codeintel-bash-with-explainshell
```

And add this to your VS Code settings:

```
"bashIde.explainshellEndpoint": "http://localhost:5000",
```

For security reasons, it defaults to "" , which disables explainshell integration. When set, this extension will send requests to the endpoint and displays documentation for flags.

Once https://github.com/idank/explainshell/pull/125 is merged, it would be possible to set this to "https://explainshell.com" , however doing this is not recommended as it will leak ***all your shell scripts*** to a third party -- do this at your own risk, or better always use a locally running Docker image.

[Nov 08, 2019] Bash aliases you can't live without by Seth Kenlon

Jul 31, 2019 | opensource.com

Tired of typing the same long commands over and over? Do you feel inefficient working on the command line? Bash aliases can make a world of difference. 28 comments

A Bash alias is a method of supplementing or overriding Bash commands with new ones. Bash aliases make it easy for users to customize their experience in a POSIX terminal. They are often defined in $HOME/.bashrc or $HOME/bash_aliases (which must be loaded by $HOME/.bashrc).

Most distributions add at least some popular aliases in the default .bashrc file of any new user account. These are simple ones to demonstrate the syntax of a Bash alias:

```
alias ls = 'ls -F'
alias ll = 'ls -lh'
```

Not all distributions ship with pre-populated aliases, though. If you add aliases manually, then you must load them into your current Bash session:

```
$ source ~/.bashrc
```

Otherwise, you can close your terminal and re-open it so that it reloads its configuration file.

With those aliases defined in your Bash initialization script, you can then type ll and get the results of ls -l , and when you type ls you get, instead of the output of plain old ls .

Those aliases are great to have, but they just scratch the surface of what's possible. Here are the top 10 Bash aliases that, once you try them, you won't be able to live without.

Set up first

Before beginning, create a file called ~/.bash_aliases :

```
$ touch ~/.bash_aliases
```

Then, make sure that this code appears in your ~/.bashrc file:

```
if [ -e $HOME / .bash_aliases ] ; then
source $HOME / .bash_aliases
fi
```

If you want to try any of the aliases in this article for yourself, enter them into your .bash_aliases file, and then load them into your Bash session with the source ~/.bashrc command.

Sort by file size

If you started your computing life with GUI file managers like Nautilus in GNOME, the Finder in MacOS, or Explorer in Windows, then you're probably used to sorting a list of files by their size. You can do that in a terminal as well, but it's not exactly succinct.

Add this alias to your configuration on a GNU system:

```
alias lt = 'ls --human-readable --size -1 -S --classify'
```

This alias replaces lt with an ls command that displays the size of each item, and then sorts it by size, in a single column, with a notation to indicate the kind of file. Load your new alias, and then try it out:

```
$ source ~ / .bashrc
$ lt
total 344K
140K configure *
44K aclocal.m4
36K LICENSE
32K config.status *
24K Makefile
24K Makefile.in
12K config.log
8.0K README.md
4.0K info.slackermedia.Git-portal.json
4.0K git-portal.spec
4.0K flatpak.path.patch
4.0K Makefile.am *
4.0K dot-gitlab.ci.yml
4.0K configure.ac *
0 autom4te.cache /
0 share /
0 bin /
0 install-sh @
0 compile @
0 missing @
0 COPYING @
```

On MacOS or BSD, the ls command doesn't have the same options, so this alias works instead:

```
alias lt = 'du -sh * | sort -h'
```

The results of this version are a little different:

```
$ du -sh * | sort -h
0 compile
0 COPYING
0 install-sh
0 missing
4.0K configure.ac
4.0K dot-gitlab.ci.yml
4.0K flatpak.path.patch
4.0K git-portal.spec
4.0K info.slackermedia.Git-portal.json
4.0K Makefile.am
8.0K README.md
12K config.log
16K bin
24K Makefile
24K Makefile.in
32K config.status
36K LICENSE
44K aclocal.m4
60K share
140K configure
476K autom4te.cache
```

In fact, even on Linux, that command is useful, because using ls lists directories and symlinks as being 0 in size, which may not be the information you actually want. It's your choice.

Thanks to Brad Alexander for this alias idea.

View only mounted drives

The mount command used to be so simple. With just one command, you could get a list of all the mounted filesystems on your computer, and it was frequently used for an overview of what drives were attached to a workstation. It used to be impressive to see more than three or four entries because most computers don't have many more USB ports than that, so the results were manageable.

Computers are a little more complicated now, and between LVM, physical drives, network storage, and virtual filesystems, the results of mount can be difficult to parse:

```
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
devtmpfs on /dev type devtmpfs (rw,nosuid,seclabel,size=8131024k,nr_inodes=2032756,mode=755)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
[...]
/dev/nvme0n1p2 on /boot type ext4 (rw,relatime,seclabel)
/dev/nvme0n1p1 on /boot/efi type vfat (rw,relatime,fmask=0077,dmask=0077,codepage=437,iocharset=ascii,shortname=winnt,errors=remount-ro)
[...]
gvfsd-fuse on /run/user/100977/gvfs type fuse.gvfsd-fuse (rw,nosuid,nodev,relatime,user_id=100977,group_id=100977)
/dev/sda1 on /run/media/seth/pocket type ext4 (rw,nosuid,nodev,relatime,seclabel,uhelper=udisks2)
/dev/sdc1 on /run/media/seth/trip type ext4 (rw,nosuid,nodev,relatime,seclabel,uhelper=udisks2)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,relatime)
```

To solve that problem, try an alias like this:

```
alias mnt = "mount | awk -F' ' '{ printf \" %s \t %s \n\" , \$ 1, \$ 3; }' | column -t | egrep ^/dev/ | sort"
```

This alias uses awk to parse the output of mount by column, reducing the output to what you probably looking for (what hard drives, and not file systems, are mounted):

```
$ mnt
/dev/mapper/fedora-root /
/dev/nvme0n1p1 /boot/efi
/dev/nvme0n1p2 /boot
/dev/sda1 /run/media/seth/pocket
/dev/sdc1 /run/media/seth/trip
```

On MacOS, the mount command doesn't provide terribly verbose output, so an alias may be overkill. However, if you prefer a succinct report, try this:

```
alias mnt = 'mount | grep -E ^/dev | column -t'
```

The results:

```
$ mnt
/dev/disk1s1 on / (apfs, local, journaled)
/dev/disk1s4 on /private/var/vm (apfs, local, noexec, journaled, noatime, nobrowse) **Find a command in your grep history**
```

Sometimes you figure out how to do something in the terminal, and promise yourself that you'll never forget what you've just learned. Then an hour goes by, and you've completely forgotten what you did.

Searching through your Bash history is something everyone has to do from time to time. If you know exactly what you're searching for, you can use Ctrl+R to do a reverse search through your history, but sometimes you can't remember the exact command you want to find.

Here's an alias to make that task a little easier:

```
alias gh = 'history|grep'
```

Here's an example of how to use it:

```
$ gh bash
482 cat ~/.bashrc | grep _alias
498 emacs ~/.bashrc
530 emacs ~/.bash_aliases
531 source ~/.bashrc **Sort by modification time**
```

It happens every Monday: You get to work, you sit down at your computer, you open a terminal, and you find you've forgotten what you were doing last Friday. What you need is an alias to list the most recently modified files.

You can use the ls command to create an alias to help you find where you left off:

```
alias left = 'ls -t -1'
```

The output is simple, although you can extend it with the -- long option if you prefer. The alias, as listed, displays this:

```
$ left
demo.jpeg
demo.xcf
design-proposal.md
rejects.txt
brainstorm.txt
query-letter.xml **Count files**
```

If you need to know how many files you have in a directory, the solution is one of the most classic examples of UNIX command construction: You list files with the ls command, control its output to be only one column with the -1 option, and then pipe that output to the wc (word count) command to count how many lines of single files there are.

It's a brilliant demonstration of how the UNIX philosophy allows users to build their own solutions using small system components. This command combination is also a lot to type if you happen to do it several times a day, and it doesn't exactly work for a directory of directories without using the -R option, which introduces new lines to the output and renders the exercise useless.

Instead, this alias makes the process easy:

```
alias count = 'find . -type f | wc -l'
```

This one counts files, ignoring directories, but *not* the contents of directories. If you have a project folder containing two directories, each of which contains two files, the alias returns four, because there are four files in the entire project.

```
$ ls
foo bar
$ count
4 **Create a Python virtual environment**
```

Do you code in Python?

Do you code in Python a lot?

If you do, then you know that creating a Python virtual environment requires, at the very least, 53 keystrokes.
That's 49 too many, but that's easily circumvented with two new aliases called ve and va :

```
alias ve = 'python3 -m venv ./venv'
alias va = 'source ./venv/bin/activate'
```

Running ve creates a new directory, called venv , containing the usual virtual environment filesystem for Python3. The va alias activates the environment in your current shell:

```
$ cd my-project
$ ve
$ va
(venv) $
```

Add a copy progress bar

Everybody pokes fun at progress bars because they're infamously inaccurate. And yet, deep down, we all seem to want them. The UNIX cp command has no progress bar, but it does have a -v option for verbosity, meaning that it echoes the name of each file being copied to your terminal. That's a pretty good hack, but it doesn't work so well when you're copying one big file and want some indication of how much of the file has yet to be transferred.

The pv command provides a progress bar during copy, but it's not common as a default application. On the other hand, the rsync command is included in the default installation of nearly every POSIX system available, and it's widely recognized as one of the smartest ways to copy files both remotely and locally.

Better yet, it has a built-in progress bar.

```
alias cpv = 'rsync -ah --info=progress2'
```

Using this alias is the same as using the cp command:

```
$ cpv bigfile.flac /run/media/seth/audio/
3.83M 6% 213.15MB/s 0:00:00 (xfr#4, to-chk=0/4)
```

An interesting side effect of using this command is that rsync copies both files and directories without the -r flag that cp would otherwise require.

Protect yourself from file removal accidents

You shouldn't use the rm command. The rm manual even says so:

> *Warning* : If you use 'rm' to remove a file, it is usually possible to recover the contents of that file. If you want more assurance that the contents are truly unrecoverable, consider using 'shred'.

If you want to remove a file, you should move the file to your Trash, just as you do when using a desktop.

POSIX makes this easy, because the Trash is an accessible, actual location in your filesystem. That location may change, depending on your platform: On a FreeDesktop , the Trash is located at ~/.local/share/Trash , while on MacOS it's ~/.Trash , but either way, it's just a directory into which you place files that you want out of sight until you're ready to erase them forever.

This simple alias provides a way to toss files into the Trash bin from your terminal:

```
alias tcn = 'mv --force -t ~/.local/share/Trash '
```

This alias uses a little-known mv flag that enables you to provide the file you want to move as the final argument, ignoring the usual requirement for that file to be listed first. Now you can use your new command to move files and folders to your system Trash:

```
$ ls
foo bar
$ tcn foo
$ ls
bar
```

Now the file is "gone," but only until you realize in a cold sweat that you still need it. At that point, you can rescue the file from your system Trash; be sure to tip the Bash and mv developers on the way out.

Note: If you need a more robust Trash command with better FreeDesktop compliance, see Trashy .

Simplify your Git workflow

Everyone has a unique workflow, but there are usually repetitive tasks no matter what. If you work with Git on a regular basis, then there's probably some sequence you find yourself repeating pretty frequently. Maybe you find yourself going back to the master branch and pulling the latest changes over and over again during the day, or maybe you find yourself creating tags and then pushing them to the remote, or maybe it's something else entirely.

No matter what Git incantation you've grown tired of typing, you may be able to alleviate some pain with a Bash alias. Largely thanks to its ability to pass arguments to hooks, Git has a rich set of introspective commands that save you from having to perform uncanny feats in Bash.

For instance, while you might struggle to locate, in Bash, a project's top-level directory (which, as far as Bash is concerned, is an entirely arbitrary designation, since the absolute top level to a computer is the root directory), Git knows its top level with a simple query. If you study up on Git hooks, you'll find yourself able to find out all kinds of information that Bash knows nothing about, but you can leverage that information with a Bash alias.

Here's an alias to find the top level of a Git project, no matter where in that project you are currently working, and then to change directory to it, change to the master branch, and perform a Git pull:

```
alias startgit = 'cd `git rev-parse --show-toplevel` && git checkout master && git pull'
```

This kind of alias is by no means a universally useful alias, but it demonstrates how a relatively simple alias can eliminate a lot of laborious navigation, commands, and waiting for prompts.

A simpler, and probably more universal, alias returns you to the Git project's top level. This alias is useful because when you're working on a project, that project more or less becomes your "temporary home" directory. It should be as simple to go "home" as it is to go to your actual home, and here's an alias to do it:

```
alias cg = 'cd `git rev-parse --show-toplevel`'
```

Now the command cg takes you to the top of your Git project, no matter how deep into its directory structure you have descended.

Change directories and view the contents at the same time

It was once (allegedly) proposed by a leading scientist that we could solve many of the planet's energy problems by harnessing the energy expended by geeks typing cd followed by ls .
It's a common pattern, because generally when you change directories, you have the impulse or the need to see what's around.

But "walking" your computer's directory tree doesn't have to be a start-and-stop process.

This one's cheating, because it's not an alias at all, but it's a great excuse to explore Bash functions. While aliases are great for quick substitutions, Bash allows you to add local functions in your .bashrc file (or a separate functions file that you load into .bashrc , just as you do your aliases file).

To keep things modular, create a new file called ~/.bash_functions and then have your .bashrc load it:

```
if [ -e $HOME / .bash_functions ] ; then
source $HOME / .bash_functions
fi
```

In the functions file, add this code:

```
function cl () {
DIR = "$*" ;
# if no DIR given, go home
if [ $# -lt 1 ] ; then
DIR = $HOME ;
fi ;
builtin cd " ${DIR} " && \
# use your preferred ls command
ls -F --color =auto
}
```

Load the function into your Bash session and then try it out:

```
$ source ~ / .bash_functions
$ cl Documents
foo bar baz
$ pwd
/ home / seth / Documents
$ cl ..
Desktop Documents Downloads
[ ... ]
$ pwd
/ home / seth
```

Functions are much more flexible than aliases, but with that flexibility comes the responsibility for you to ensure that your code makes sense and does what you expect. Aliases are meant to be simple, so keep them easy, but useful. For serious modifications to how Bash behaves, use functions or custom shell scripts saved to a location in your PATH .

For the record, there *are* some clever hacks to implement the cd and ls sequence as an alias, so if you're patient enough, then the sky is the limit even using humble aliases.

Start aliasing and functioning

Customizing your environment is what makes Linux fun, and increasing your efficiency is what makes Linux life-changing. Get started with simple aliases, graduate to functions, and post your must-have aliases in the comments!

ACG on 31 Jul 2019 Permalink

> One function I like a lot is a function that diffs a file and its backup.
> It goes something like
>
> ```
> #!/usr/bin/env bash
> file="${1:?File not given}"
>
> if [[ ! -e "$file" || ! -e "$file"~ ]]; then
> echo "File doesn't exist or has no backup" 1>&2
> exit 1
> fi
>
> diff --color=always "$file"{~,} | less -r
> ```
>
> I may have gotten the if wrong, but you get the idea. I'm typing this on my phone, away from home.
> Cheers

Seth Kenlon on 31 Jul 2019 Permalink

> That's pretty slick! I like it.
>
> My backup tool of choice (rdiff-backup) handles these sorts of comparisons pretty well, so I tend to be confident in my backup files. That said, there's always the edge case, and this kind of function is a great solution for those. Thanks!

Kevin Cole on 13 Aug 2019 Permalink

> A few of my "cannot-live-withouts" are regex based:
>
> Decomment removes full-line comments and blank lines. For example, when looking at a "stock" /etc/httpd/whatever.conf file that has a gazillion lines in it,
>
> ```
> alias decomment='egrep -v "^[[:space:]]*((#|;|//).*)?$" '
> ```
>
> will show you that only four lines in the file actually DO anything, and the gazillion minus four are comments. I use this ALL the time with config files, Python (and other languages) code, and god knows where else.
>
> Then there's unprintables and expletives which are both very similar:
>
> ```
> alias unprintable='grep --color="auto" -P -n "[\x00-\x1E]"'
> alias expletives='grep --color="auto" -P -n "[^\x00-\x7E]" '
> ```
>
> The first shows which lines (with line numbers) in a file contain control characters, and the second shows which lines in a file contain anything "above" a RUBOUT, er, excuse me, I mean above ASCII 127. (I feel old.) ;-) Handy when, for example, someone gives you a program that they edited or created with LibreOffice, and oops... half of the quoted strings have "real" curly opening and closing quote marks instead of ASCII 0x22 "straight" quote mark delimiters... But there's actually a few curlies you want to keep, so a "nuke 'em all in one swell foop" approach won't work.

Seth Kenlon on 14 Aug 2019 Permalink

> These are great!

Dan Jones on 13 Aug 2019 Permalink

> Your `cl` function could be simplified, since `cd` without arguments already goes to home.
>
> ```
> function cl() {
> cd "$@" && \
> ls -F --color=auto
> }
> ```

Seth Kenlon on 14 Aug 2019 Permalink

> Nice!

jkeener on 20 Aug 2019 Permalink

> The first alias in my .bash_aliases file is always:

alias realias='vim ~/.bash_aliases; source ~/.bash_aliases'

replace vim with your favorite editor or $VISUAL

bhuvana on 04 Oct 2019 Permalink

Thanks for this post! I have created a Github repo- https://github.com/bhuvana-guna/awesome-bash-shortcuts
with a motive to create an extended list of aliases/functions for various programs. As I am a newbie to terminal and linux, please do contribute to it with these and other super awesome utilities and help others easily access them.

[Nov 08, 2019] Winterize your Bash prompt in Linux

Nov 08, 2019 | opensource.com

Your Linux terminal probably supports Unicode, so why not take advantage of that and add a seasonal touch to your prompt? 11 Dec 2018 Jason Baker (Red Hat) Feed 84 up 3 comments Image credits : Jason Baker x **Subscribe now**

Get the highlights in your inbox every week.

https://opensource.com/eloqua-embedded-email-capture-block.html?offer_id=70160000000QzXNAA0

- Top 7 terminal emulators for Linux
- 10 command-line tools for data analysis in Linux
- Download Now: SSH cheat sheet
- Advanced Linux commands cheat sheet
- Linux command line tutorials

Hello once again for another installment of the Linux command-line toys advent calendar. If this is your first visit to the series, you might be asking yourself what a command-line toy even is? Really, we're keeping it pretty open-ended: It's anything that's a fun diversion at the terminal, and we're giving bonus points for anything holiday-themed.

Maybe you've seen some of these before, maybe you haven't. Either way, we hope you have fun.

Today's toy is super-simple: It's your Bash prompt. Your Bash prompt? Yep! We've got a few more weeks of the holiday season left to stare at it, and even more weeks of winter here in the northern hemisphere, so why not have some fun with it.

Your Bash prompt currently might be a simple dollar sign ($), or more likely, it's something a little longer. If you're not sure what makes up your Bash prompt right now, you can find it in an environment variable called $PS1. To see it, type:

```
echo $PS1
```

For me, this returns:

```
[\u@\h \W]\$
```

The \u , \h , and \W are special characters for username, hostname, and working directory. There are others you can use as well; for help building out your Bash prompt, you can use EzPrompt , an online generator of PS1 configurations that includes lots of options including date and time, Git status, and more.

You may have other variables that make up your Bash prompt set as well; $PS2 for me contains the closing brace of my command prompt. See this article for more information.

To change your prompt, simply set the environment variable in your terminal like this:

```
$ PS1 = '\u is cold: '
jehb is cold:
```

To set it permanently, add the same code to your /etc/bashrc using your favorite text editor.

So what does this have to do with winterization? Well, chances are on a modern machine, your terminal support Unicode, so you're not limited to the standard ASCII character set. You can use any emoji that's a part of the Unicode specification, including a snowflake ❄, a snowman ☃, or a pair of skis 🎿. You've got plenty of wintery options to choose from.

🎄 Christmas Tree
🧥 Coat
🦌 Deer
🧤 Gloves
🤶 Mrs. Claus
🎅 Santa Claus
🧣 Scarf
🎿 Skis
🏂 Snowboarder
❄ Snowflake
☃ Snowman
⛄ Snowman Without Snow
🎁 Wrapped Gift

Pick your favorite, and enjoy some winter cheer. Fun fact: modern filesystems also support Unicode characters in their filenames, meaning you can technically name your next program "❄❄❄❄❄.py" . That said, please don't.

Do you have a favorite command-line toy that you think I ought to include? The calendar for this series is mostly filled out but I've got a few spots left. Let me know in the comments below, and I'll check it out. If there's space, I'll try to include it. If not, but I get some good submissions, I'll do a round-up of honorable mentions at the end.

[Nov 08, 2019] How to change the default shell prompt

Jun 29, 2014 | access.redhat.com

- The shell prompt is controlled via the PS environment variables.

Raw

PS1 - The value of this parameter is expanded and used as the primary prompt string. The default value is \u@\h \W\\$.
PS2 - The value of this parameter is expanded as with PS1 and used as the secondary prompt string. The default is]

PS3 - The value of this parameter is used as the prompt for the select command
PS4 - The value of this parameter is expanded as with PS1 and the value is printed before each command bash displays during an execution trace. The

- PS1 is a primary prompt variable which holds \u@\h \W\\$ special bash characters. This is the default structure of the bash prompt and is displayed every time a user logs in using a terminal. These default values are set in the /etc/bashrc file.
- The special characters in the default prompt are as follows:

Raw

```
\u = username
\h = hostname
\W = current working directory
```

- This command will show the current value.

Raw

```
# echo $PS1
```

- This can be modified by changing the PS1 variable:

Raw

```
# PS1='[[prod]\u@\h \W]\$'
```

- The modified shell prompt will look like:

Raw

```
[[prod]root@hostname ~]#
```

- In order to make these settings permanent, edit the /etc/bashrc file:

Find this line:

Raw

```
[ "$PS1" = "\\s-\\v\\$ " ] && PS1="[\u@\h \W]\\$ "
```

And change it as needed:

Raw

```
[ "$PS1" = "\\s-\\v\\$ " ] && PS1="[[prod]\u@\h \W]\\$ "
```

- Product(s)
- Red Hat Enterprise Linux

- Component
- bash

- Category
- Customize or extend

- Tags
- rhel_4
- rhel_5
- rhel_6
- shells

This solution is part of Red Hat's fast-track publication program, providing a huge library of solutions that Red Hat engineers have created while supporting our customers. To give you the knowledge you need the instant it becomes available, these articles may be presented in a raw and unedited form. **2 Comments**
Sort By Oldest Log in to comment MW Community Member 48 points

6 October 2016 1:53 PM Mike Willis

This solution has simply "Red Hat Enterprise Linux" in the Environment section implying it applies to all versions of Red Hat Enterprise Linux.

Editing /etc/bashrc is against the advice of the comments in /etc/bashrc on Red Hat Enterprise Linux 7 which say

Raw

```
# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.
```

On RHEL 7 instead of the solution suggested above create a /etc/profile.d/custom.sh which contains

Raw

```
PS1="[[prod]\u@\h \W]\$ "
```

27 March 2019 12:44 PM Mike Chanslor

Hello Red Hat community! I also found this useful: Raw

```
Special prompt variable characters:
 \d   The date, in "Weekday Month Date" format (e.g., "Tue May 26").

 \h   The hostname, up to the first . (e.g. deckard)
 \H   The hostname. (e.g. deckard.SS64.com)

 \j   The number of jobs currently managed by the shell.

 \l   The basename of the shell's terminal device name.

 \s   The name of the shell, the basename of $0 (the portion following
      the final slash).
```

\t The time, in 24-hour HH:MM:SS format.
\T The time, in 12-hour HH:MM:SS format.
\@ The time, in 12-hour am/pm format.

\u The username of the current user.

\v The version of Bash (e.g., 2.00)

\V The release of Bash, version + patchlevel (e.g., 2.00.0)

\w The current working directory.
\W The basename of $PWD.

\! The history number of this command.
\# The command number of this command.

\$ If you are not root, inserts a "$"; if you are root, you get a "#" (root uid = 0)

\nnn The character whose ASCII code is the octal value nnn.

\n A newline.
\r A carriage return.
\e An escape character (typically a color code).
\a A bell character.
\\ A backslash.

\[Begin a sequence of non-printing characters. (like color escape sequences). This allows bash to calculate word wrapping correctly.

\] End a sequence of non-printing characters.
Using single quotes instead of double quotes when exporting your PS variables is recommended, it makes the prompt a tiny bit faster to evaluate plu

[Nov 08, 2019] How to escape unicode characters in bash prompt correctly - Stack Overflow

Nov 08, 2019 | stackoverflow.com

How to escape unicode characters in bash prompt correctly Ask Question Asked 8 years, 2 months ago Active 9 months ago Viewed 6k times 7 2

Andy Ray ,Aug 18, 2011 at 19:08

I have a specific method for my bash prompt, let's say it looks like this:

```
CHAR="␣"
my_function="
   prompt=\" \[\$CHAR\]\"
   echo -e \$prompt"

PS1="\$(${my_function}) \$ "
```

To explain the above, I'm builidng my bash prompt by executing a function stored in a string, which was a decision made as the result of this question . Let's pretend like it works fine, because it does, except when unicode characters get involved

I am trying to find the proper way to escape a unicode character, because right now it messes with the bash line length. An easy way to test if it's broken is to type a long command, execute it, press CTRL-R and type to find it, and then pressing CTRL-A CTRL-E to jump to the beginning / end of the line. If the text gets garbled then it's not working.

I have tried several things to properly escape the unicode character in the function string, but nothing seems to be working.

Special characters like this work:

```
COLOR_BLUE=$(tput sgr0 && tput setaf 6)

my_function="
   prompt="\\[\$COLOR_BLUE\\] \"
   echo -e \$prompt"
```

Which is the main reason I made the prompt a function string. That escape sequence does NOT mess with the line length, it's just the unicode character.

Andy Ray ,Aug 23, 2011 at 2:09

The \[...\] sequence says to ignore this part of the string completely, which is useful when your prompt contains a zero-length sequence, such as a control sequence which changes the text color or the title bar, say. But in this case, you are printing a character, so the length of it is not zero. Perhaps you could work around this by, say, using a no-op escape sequence to fool Bash into calculating the correct line length, but it sounds like that way lies madness.

The correct solution would be for the line length calculations in Bash to correctly grok UTF-8 (or whichever Unicode encoding it is that you are using). Uhm, have you tried without the \[...\] sequence?

Edit: The following implements the solution I propose in the comments below. The cursor position is saved, then two spaces are printed, outside of \[...\] , then the cursor position is restored, and the Unicode character is printed on top of the two spaces. This assumes a fixed font width, with double width for the Unicode character.

```
PS1='\["`tput sc`"\]  \["`tput rc`"␣\] \$ '
```

At least in the OSX Terminal, Bash 3.2.17(1)-release, this passes cursory [sic] testing.

In the interest of transparency and legibility, I have ignored the requirement to have the prompt's functionality inside a function, and the color coding; this just changes the prompt to the character, space, dollar prompt, space. Adapt to suit your somewhat more complex needs.

tripleee ,Aug 23, 2011 at 7:01

@tripleee wins it, posting the final solution here because it's a pain to post code in comments:

```
CHAR="␉"
my_function="
   prompt=\" \\[`tput sc`\\]  \\[`tput rc`\\]\\[\$CHAR\\] \"
   echo -e \$prompt"

PS1="\$(${my_function}) \$ "
```

The trick as pointed out in @tripleee's link is the use of the commands tput sc and tput rc which save and then restore the cursor position. The code is effectively saving the cursor position, printing two spaces for width, restoring the cursor position to before the spaces, then printing the special character so that the width of the line is from the two spaces, not the character.

> ,

(Not the answer to your problem, but some pointers and general experience related to your issue.)

I see the behaviour you describe about cmd-line editing (Ctrl-R, ... Cntrl-A Ctrl-E ...) all the time, even without unicode chars.

At one work-site, I spent the time to figure out the diff between the terminals interpretation of the TERM setting VS the TERM definition used by the OS (well, stty I suppose).

NOW, when I have this problem, I escape out of my current attempt to edit the line, bring the line up again, and then immediately go to the 'vi' mode, which opens the vi editor. (press just the 'v' char, right?). All the ease of use of a full-fledged session of vi; why go with less ;-)?

Looking again at your problem description, when you say

```
my_function="
   prompt=\" \[\$CHAR\]\"
   echo -e \$prompt"
```

That is just a string definition, right? and I'm assuming your simplifying the problem definition by assuming this is the output of your my_function . It seems very likely in the steps of creating the function definition, calling the function AND using the values returned are a lot of opportunities for shell-quoting to not work the way you want it to.

If you edit your question to include the my_function definition, and its complete use (reducing your function to just what is causing the problem), it may be easier for others to help with this too. Finally, do you use set -vx regularly? It can help show how/wnen/what of variable expansions, you may find something there.

Failing all of those, look at Orielly termcap & terminfo . You may need to look at the man page for your local systems stty and related cmds AND you may do well to look for user groups specific to you Linux system (I'm assuming you use a Linux variant).

I hope this helps.

[Oct 23, 2019] Apply Tags To Linux Commands To Easily Retrieve Them From History

As bash store command in history with the tail comment it allows to implement tags.

Oct 23, 2019 | www.ostechnix.com

Let us take the following one-liner Linux command as an example.

```
$ find . -size +10M -type f -print0 | xargs -0 ls -Ssh | sort -z
```

For those wondering, the above command will find and list files bigger than 10 MB in the current directory and sort them by size. I admit that I couldn't remember this command. I guess some of you can't remember this command either. This is why we are going to apply a tag to such kind of commands.

To apply a tag, just type the command and add the comment (i.e. tag) at the end of the command as shown below.

```
$ find . -size +10M -type f -print0 | xargs -0 ls -Ssh | sort -z #ListFilesBiggerThanXSize
```

Here, **#ListFilesBiggerThanXSize** is the tag name to the above command. Make sure you have given a space between the command and tag name. Also, please use the tag name as simple, short and clear as possible to easily remember it later. Otherwise, you may need another tool to recall the tags.

To run it again, simply use the tag name like below.

```
$ !? #ListFilesBiggerThanXSize
```

Here, the ! (Exclamation mark) and ? (Question mark) operators are used to fetch and run the command which we tagged earlier from the BASH history.

[Aug 28, 2019] Echo Command in Linux with Examples

Notable quotes:

"... The -e parameter is used for the interpretation of backslashes ..."

"... The -n option is used for omitting trailing newline. ..."

Aug 28, 2019 | linoxide.com

The -e parameter is used for the interpretation of backslashes

... ...

To create a new line after each word in a string use the -e operator with the \n option as shown

```
$ echo -e "Linux \nis \nan \nopensource \noperating \nsystem"
```

... ...

Omit echoing trailing newline

The -n option is used for omitting trailing newline. This is shown in the example below

```
$ echo -n "Linux is an opensource operating system"
```

Sample Output

```
Linux is an opensource operating systemjames@buster:/$
```

[Aug 22, 2019] How To Display Bash History Without Line Numbers - OSTechNix

Aug 22, 2019 | www.ostechnix.com

"Method 2 – Using history command"

We can use the history command's write option to print the history without numbers like below.

```
$ history -w /dev/stdout
```

"Method 3 – Using history and cut commands"

One such way is to use history and cut commands like below.

```
$ history | cut -c 8-
```

[Aug 14, 2019] bash - PID background process - Unix Linux Stack Exchange

Aug 14, 2019 | unix.stackexchange.com

PID background process Ask Question Asked 2 years, 8 months ago Active 2 years, 8 months ago Viewed 2k times 2

Raul ,Nov 27, 2016 at 18:21

As I understand pipes and commands, bash takes each command, spawns a process for each one and connects stdout of the previous one with the stdin of the next one.

For example, in "ls -lsa | grep feb", bash will create two processes, and connect the output of "ls -lsa" to the input of "grep feb".

When you execute a background command like "sleep 30 &" in bash, you get the pid of the background process running your command. Surprisingly for me, when I wrote "ls -lsa | grep feb &" bash returned only one PID.

How should this be interpreted? A process runs both "ls -lsa" and "grep feb"? Several process are created but I only get the pid of one of them?

Raul ,Nov 27, 2016 at 19:21

Spawns 2 processes. The & displays the PID of the second process. Example below.

```
$ echo $$
13358
$ sleep 100 | sleep 200 &
[1] 13405
$ ps -ef|grep 13358
ec2-user 13358 13357  0 19:02 pts/0    00:00:00 -bash
ec2-user 13404 13358  0 19:04 pts/0    00:00:00 sleep 100
ec2-user 13405 13358  0 19:04 pts/0    00:00:00 sleep 200
ec2-user 13406 13358  0 19:04 pts/0    00:00:00 ps -ef
ec2-user 13407 13358  0 19:04 pts/0    00:00:00 grep --color=auto 13358
$
```

> ,

When you run a job in the background, bash prints the process ID of its subprocess, the one that runs the command in that job. If that job happens to create more subprocesses, that's none of the parent shell's business.

When the background job is a pipeline (i.e. the command is of the form `something1 | something2 &` , and not e.g. `{ something1 | something2; } &`), there's an optimization which is strongly suggested by POSIX and performed by most shells including bash: each of the elements of the pipeline are executed directly as subprocesses of the original shell. What POSIX mandates is that the variable $! is set to the last command in the pipeline in this case. In most shells, that last command is a subprocess of the original process, and so are the other commands in the pipeline.

When you run `ls -lsa | grep feb` , there are three processes involved: the one that runs the left-hand side of the pipe (a subshell that finishes setting up the pipe then executes `ls`), the one that runs the right-hand side of the pipe (a subshell that finishes setting up the pipe then executes `grep`), and the original process that waits for the pipe to finish.

You can watch what happens by tracing the processes:

```
$ strace -f -e clone,wait4,pipe,execve,setpgid bash --norc
execve("/usr/local/bin/bash", ["bash", "--norc"], [/* 82 vars */]) = 0
setpgid(0, 24084)                       = 0
bash-4.3$ sleep 10 | sleep 20 &
```

Note how the second sleep is reported and stored as $! , but the process group ID is the first sleep . Dash has the same oddity, ksh and mksh don't.

[Aug 14, 2019] unix - How to get PID of process by specifying process name and store it in a variable to use further - Stack Overflow

Aug 14, 2019 | stackoverflow.com

Nidhi ,Nov 28, 2014 at 0:54

```
pids=$(pgrep <name>)
```

will get you the pids of all processes with the given name. To kill them all, use

```
kill -9 $pids
```

To refrain from using a variable and directly kill all processes with a given name issue

```
pkill -9 <name>
```

panticz.de ,Nov 11, 2016 at 10:11

On a single line...

```
pgrep -f process_name | xargs kill -9
```

flazzarini ,Jun 13, 2014 at 9:54

Another possibility would be to use pidof it usually comes with most distributions. It will return you the PID of a given process by using it's name.

```
pidof process_name
```

This way you could store that information in a variable and execute kill -9 on it.

```
#!/bin/bash
pid=`pidof process_name`
kill -9 $pid
```

Pawel K ,Dec 20, 2017 at 10:27

use grep [n]ame to remove that grep -v name this is first... Sec using xargs in the way how it is up there is wrong to rnu whatever it is piped you have to use -i (interactive mode) otherwise you may have issues with the command.

ps axf | grep | grep -v grep | awk '{print "kill -9 " $1}' ? ps aux |grep [n]ame | awk '{print "kill -9 " $2}' ? isnt that better ?

[Aug 14, 2019] linux - How to get PID of background process - Stack Overflow

Highly recommended!

Aug 14, 2019 | stackoverflow.com

How to get PID of background process? Ask Question Asked 9 years, 8 months ago Active 7 months ago Viewed 238k times 336 64

pixelbeat ,Mar 20, 2013 at 9:11

I start a background process from my shell script, and I would like to kill this process when my script finishes.

How to get the PID of this process from my shell script? As far as I can see variable $! contains the PID of the current script, not the background process.

WiSaGaN ,Jun 2, 2015 at 14:40

You need to save the PID of the background process at the time you start it:

```
foo &
FOO_PID=$!
# do other stuff
kill $FOO_PID
```

You cannot use job control, since that is an interactive feature and tied to a controlling terminal. A script will not necessarily have a terminal attached at all so job control will not necessarily be available.

Phil ,Dec 2, 2017 at 8:01

You can use the jobs -l command to get to a particular jobL

```
^Z
[1]+  Stopped                 guard

my_mac:workspace r$ jobs -l
[1]+ 46841 Suspended: 18          guard
```

In this case, 46841 is the PID.

From help jobs :

-l Report the process group ID and working directory of the jobs.

jobs -p is another option which shows just the PIDs.

Timo ,Dec 2, 2017 at 8:03

- $$ is the current script's pid
- $! is the pid of the last background process

Here's a sample transcript from a bash session (%1 refers to the ordinal number of background process as seen from jobs):

```
$ echo $$
3748

$ sleep 100 &
[1] 192

$ echo $!
192

$ kill %1

[1]+  Terminated              sleep 100
```

lepe ,Dec 2, 2017 at 8:29

An even simpler way to kill all child process of a bash script:

```
pkill -P $$
```

The -P flag works the same way with pkill and pgrep - it gets child processes, only with pkill the child processes get killed and with pgrep child PIDs are printed to stdout.

Luis Ramirez ,Feb 20, 2013 at 23:11

this is what I have done. Check it out, hope it can help.

```
#!/bin/bash
#
# So something to show.
echo "UNO" >  UNO.txt
echo "DOS" >  DOS.txt
#
# Initialize Pid List
dPidLst=""
#
# Generate background processes
tail -f UNO.txt&
dPidLst="$dPidLst $!"
tail -f DOS.txt&
dPidLst="$dPidLst $!"
#
# Report process IDs
echo PID=$$
echo dPidLst=$dPidLst
#
# Show process on current shell
ps -f
#
# Start killing background processes from list
for dPid in $dPidLst
do
      echo killing $dPid. Process is still there.
      ps | grep $dPid
      kill $dPid
      ps | grep $dPid
      echo Just ran """ps""" command, $dPid must not show again.
done
```

Then just run it as: ./bgkill.sh with proper permissions of course

```
root@umsstd22 [P]:~# ./bgkill.sh
PID=23757
dPidLst= 23758 23759
UNO
DOS
UID        PID  PPID  C STIME TTY          TIME CMD
root      3937  3935  0 11:07 pts/5    00:00:00 -bash
root     23757  3937  0 11:55 pts/5    00:00:00 /bin/bash ./bgkill.sh
root     23758 23757  0 11:55 pts/5    00:00:00 tail -f UNO.txt
root     23759 23757  0 11:55 pts/5    00:00:00 tail -f DOS.txt
root     23760 23757  0 11:55 pts/5    00:00:00 ps -f
killing 23758. Process is still there.
23758 pts/5    00:00:00 tail
./bgkill.sh: line 24: 23758 Terminated              tail -f UNO.txt
Just ran 'ps' command, 23758 must not show again.
killing 23759. Process is still there.
23759 pts/5    00:00:00 tail
./bgkill.sh: line 24: 23759 Terminated              tail -f DOS.txt
Just ran 'ps' command, 23759 must not show again.
root@umsstd22 [P]:~# ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
```

```
root     3937  3935  0 11:07 pts/5    00:00:00 -bash
root    24200  3937  0 11:56 pts/5    00:00:00 ps -f
```

Phil ,Oct 15, 2013 at 18:22

You might also be able to use pstree:

```
pstree -p user
```

This typically gives a text representation of all the processes for the "user" and the -p option gives the process-id. It does not depend, as far as I understand, on having the processes be owned by the current shell. It also shows forks.

Phil ,Dec 4, 2018 at 9:46

pgrep can get you all of the child PIDs of a parent process. As mentioned earlier $$ is the current scripts PID. So, if you want a script that cleans up after itself, this should do the trick:

```
trap 'kill $( pgrep -P $$ | tr "\n" " " )' SIGINT SIGTERM EXIT
```

[Jul 26, 2019] Cheat.sh Shows Cheat Sheets On The Command Line Or In Your Code Editor>

The choice of shell as a programming language is strange, but the idea is good...

Notable quotes:

"... The tool is developed by Igor Chubin, also known for its console-oriented weather forecast service wttr.in , which can be used to retrieve the weather from the console using only cURL or Wget. ..."

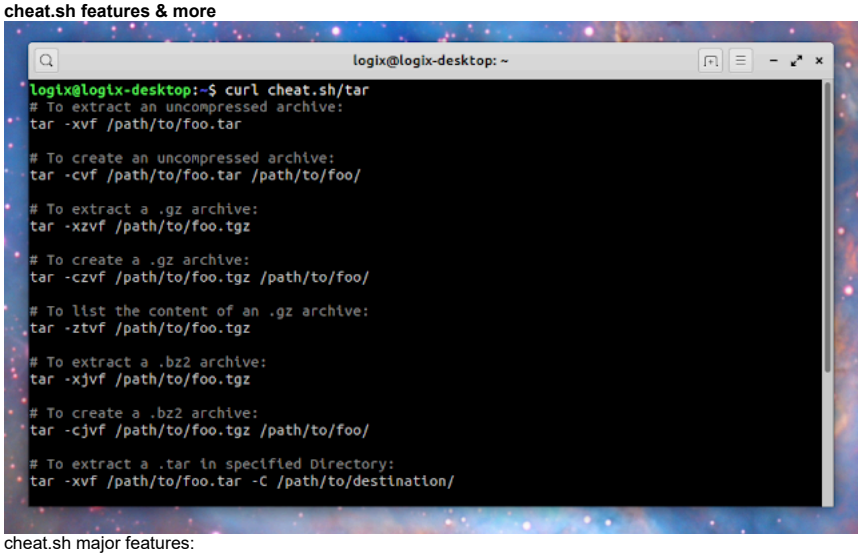
Jul 26, 2019 | www.linuxuprising.com

While it does have its own cheat sheet repository too, the project is actually concentrated around the creation of a unified mechanism to access well developed and maintained cheat sheet repositories.

The tool is developed by Igor Chubin, also known for its console-oriented weather forecast service wttr.in , which can be used to retrieve the weather from the console using only cURL or Wget.

It's worth noting that cheat.sh is not new. In fact it had its initial commit around May, 2017, and is a very popular repository on GitHub. But I personally only came across it recently, and I found it very useful, so I figured there must be some Linux Uprising readers who are not aware of this cool gem.

cheat.sh features & more



cheat.sh major features:

- **Supports 58 programming languages , several DBMSes, and more than 1000 most important UNIX/Linux commands**
- **Very fast, returns answers within 100ms**
- **Simple curl / browser interface**
- **An optional command line client (cht.sh) is available, which allows you to quickly search cheat sheets and easily copy snippets without leaving the terminal**
- **Can be used from code editors, allowing inserting code snippets without having to open a web browser, search for the code, copy it, then return to your code editor and paste it. It supports Vim, Emacs, Visual Studio Code, Sublime Text and IntelliJ Idea**
- **Comes with a special stealth mode in which any text you select (adding it into the selection buffer of X Window System or into the clipboard) is used as a search query by cht.sh, so you can get answers without touching any other keys**

The command line client features a special shell mode with a persistent queries context and readline support. It also has a query history, it integrates with the clipboard, supports tab completion for shells like Bash, Fish and Zsh, and it includes the stealth mode I mentioned in the cheat.sh features.

The web, curl and cht.sh (command line) interfaces all make use of https://cheat.sh/ but if you prefer, **you can self-host it .**

It should be noted that each editor plugin supports a different feature set (configurable server, multiple answers, toggle comments, and so on). You can view a feature comparison of each cheat.sh editor plugin on the Editors integration section of the project's GitHub page.

Want to contribute a cheat sheet? See the cheat.sh guide on editing or adding a new cheat sheet.

Interested in bookmarking commands instead? You may want to give Marker, a command bookmark manager for the console , a try.

cheat.sh curl / command line client usage examples
Examples of using cheat.sh using the curl interface (this requires having curl installed as you'd expect) from the command line:

Show the tar command cheat sheet:

```
curl cheat.sh/tar
```

Example with output:

```
$ curl cheat.sh/tar
# To extract an uncompressed archive:
tar -xvf /path/to/foo.tar

# To create an uncompressed archive:
tar -cvf /path/to/foo.tar /path/to/foo/

# To extract a .gz archive:
tar -xzvf /path/to/foo.tgz

# To create a .gz archive:
tar -czvf /path/to/foo.tgz /path/to/foo/

# To list the content of an .gz archive:
tar -ztvf /path/to/foo.tgz

# To extract a .bz2 archive:
tar -xjvf /path/to/foo.tgz

# To create a .bz2 archive:
tar -cjvf /path/to/foo.tgz /path/to/foo/

# To extract a .tar in specified Directory:
tar -xvf /path/to/foo.tar -C /path/to/destination/

# To list the content of an .bz2 archive:
tar -jtvf /path/to/foo.tgz

# To create a .gz archive and exclude all jpg,gif,... from the tgz
tar czvf /path/to/foo.tgz --exclude=\*.{jpg,gif,png,wmv,flv,tar.gz,zip} /path/to/foo/

# To use parallel (multi-threaded) implementation of compression algorithms:
tar -z ... -> tar -Ipigz ...
tar -j ... -> tar -Ipbzip2 ...
tar -J ... -> tar -Ipixz ...
```

cht.sh also works instead of cheat.sh:

```
curl cht.sh/tar
```

Want to search for a keyword in all cheat sheets? Use:

```
curl cheat.sh/~keyword
```

List the Python programming language cheat sheet for random list :

```
curl cht.sh/python/random+list
```

Example with output:

```
$ curl cht.sh/python/random+list
#  python - How to randomly select an item from a list?
#
#  Use random.choice
#  (https://docs.python.org/2/library/random.htmlrandom.choice):

import random

foo = ['a', 'b', 'c', 'd', 'e']
print(random.choice(foo))

#  For cryptographically secure random choices (e.g. for generating a
#  passphrase from a wordlist), use random.SystemRandom
#  (https://docs.python.org/2/library/random.htmlrandom.SystemRandom)
#  class:

import random

foo = ['battery', 'correct', 'horse', 'staple']
secure_random = random.SystemRandom()
print(secure_random.choice(foo))

#  [Pēteris Caune] [so/q/306400] [cc by-sa 3.0]
```

Replace python with some other programming language supported by cheat.sh, and random+list with the cheat sheet you want to show.

Want to eliminate the comments from your answer? Add ?Q at the end of the query (below is an example using the same /python/random+list):

```
$ curl cht.sh/python/random+list?Q
import random

foo = ['a', 'b', 'c', 'd', 'e']
```

```
print(random.choice(foo))

import random

foo = ['battery', 'correct', 'horse', 'staple']
secure_random = random.SystemRandom()
print(secure_random.choice(foo))
```

For more flexibility and tab completion you can use cht.sh, the command line cheat.sh client; you'll find instructions for how to install it further down this article.
Examples of using the cht.sh command line client:

Show the tar command cheat sheet:

```
cht.sh tar
```

List the Python programming language cheat sheet for random list :

```
cht.sh python random list
```

There is no need to use quotes with multiple keywords.

You can start the cht.sh client in a special shell mode using:

```
cht.sh --shell
```

And then you can start typing your queries. Example:

```
$ cht.sh --shell
cht.sh> bash loop
```

If all your queries are about the same programming language, you can start the client in the special shell mode, directly in that context. As an example, start it with the Bash context using:

```
cht.sh --shell bash
```

Example with output:

```
$ cht.sh --shell bash
cht.sh/bash> loop
.....
cht.sh/bash> switch case
```

Want to copy the previously listed answer to the clipboard? Type c , then press Enter to copy the whole answer, or type C and press Enter to copy it without comments.

Type help in the cht.sh interactive shell mode to see all available commands. Also look under the Usage section from the cheat.sh GitHub project page for more options and advanced usage.

How to install cht.sh command line client
You can use cheat.sh in a web browser, from the command line with the help of curl and without having to install anything else, as explained above, as a code editor plugin, or using its command line client which has some extra features, which I already mentioned. The steps below are for installing this cht.sh command line client.

If you'd rather install a code editor plugin for cheat.sh, see the Editors integration page.

1. Install dependencies.

To install the cht.sh command line client, the curl command line tool will be used, so this needs to be installed on your system. Another dependency is rlwrap , which is required by the cht.sh special shell mode. Install these dependencies as follows.

- Debian, Ubuntu, Linux Mint, Pop!_OS, and any other Linux distribution based on Debian or Ubuntu:

  ```
  sudo apt install curl rlwrap
  ```

- Fedora:

  ```
  sudo dnf install curl rlwrap
  ```

- Arch Linux, Manjaro:

  ```
  sudo pacman -S curl rlwrap
  ```

- openSUSE:

  ```
  sudo zypper install curl rlwrap
  ```

The packages seem to be named the same on most (if not all) Linux distributions, so if your Linux distribution is not on this list, just install the curl and rlwrap packages using your distro's package manager.

2. Download and install the cht.sh command line interface.

You can install this either for your user only (so only you can run it), or for all users:

- Install it for your user only. The command below assumes you have a ~/.bin folder added to your PATH (and the folder exists). If you have some other local folder in your PATH where you want to install cht.sh, change install path in the commands:

```
curl https://cht.sh/:cht.sh > ~/.bin/cht.sh

chmod +x ~/.bin/cht.sh
```

- Install it for all users (globally, in /usr/local/bin):

```
curl https://cht.sh/:cht.sh | sudo tee /usr/local/bin/cht.sh

sudo chmod +x /usr/local/bin/cht.sh
```

If the first command appears to have frozen displaying only the cURL output, press the Enter key and you'll be prompted to enter your password in order to save the file to /usr/local/bin .

You may also download and install the cheat.sh command completion for Bash or Zsh:

- Bash:

```
mkdir ~/.bash.d

curl https://cheat.sh/:bash_completion > ~/.bash.d/cht.sh

echo ". ~/.bash.d/cht.sh" >> ~/.bashrc
```

- Zsh:

```
mkdir ~/.zsh.d

curl https://cheat.sh/:zsh > ~/.zsh.d/_cht

echo 'fpath=(~/.zsh.d/ $fpath)' >> ~/.zshrc
```

Opening a new shell / terminal and it will load the cheat.sh completion.

[Jul 26, 2019] What Is /dev/null in Linux by Alexandru Andrei

Images removed...

Jul 23, 2019 | www.maketecheasier.com

... ...

In technical terms, "/dev/null" is a virtual device file. As far as programs are concerned, these are treated just like real files. Utilities can request data from this kind of source, and the operating system feeds them data. But, instead of reading from disk, the operating system generates this data dynamically. An example of such a file is "/dev/zero."

In this case, however, you will write to a device file. Whatever you write to "/dev/null" is discarded, forgotten, thrown into the void. To understand why this is useful, you must first have a basic understanding of standard output and standard error in Linux or *nix type operating systems.

Related : How to Use the Tee Command in Linux

stdout and stder

A command-line utility can generate two types of output. Standard output is sent to stdout. Errors are sent to stderr.

By default, stdout and stderr are associated with your terminal window (or console). This means that anything sent to stdout and stderr is normally displayed on your screen. But through shell redirections, you can change this behavior. For example, you can redirect stdout to a file. This way, instead of displaying output on the screen, it will be saved to a file for you to read later – or you can redirect stdout to a physical device, say, a digital LED or LCD display.

A full article about pipes and redirections is available if you want to learn more.

- With 2> you redirect standard error messages. Example: 2>/dev/null or 2>/home/user/error.log .
- With 1> you redirect standard output.
- With &> you redirect both standard error and standard output.

Related : 12 Useful Linux Commands for New User

Use /dev/null to Get Rid of Output You Don't Need

Since there are two types of output, standard output and standard error, the first use case is to filter out one type or the other. It's easier to understand through a practical example. Let's say you're looking for a string in "/sys" to find files that refer to power settings.

```
grep -r power /sys/
```

There will be a lot of files that a regular, non-root user cannot read. This will result in many "Permission denied" errors.

These clutter the output and make it harder to spot the results that you're looking for. Since "Permission denied" errors are part of stderr, you can redirect them to "/dev/null."

```
grep -r power /sys/ 2>/dev/null
```

As you can see, this is much easier to read.

In other cases, it might be useful to do the reverse: filter out standard output so you can only see errors.

```
ping google.com 1>/dev/null
```

The screenshot above shows that, without redirecting, ping displays its normal output when it can reach the destination machine. In the second command, nothing is displayed while the network is online, but as soon as it gets disconnected, only error messages are displayed.

You can redirect both stdout and stderr to two different locations.

```
ping google.com 1>/dev/null 2>error.log
```

In this case, stdout messages won't be displayed at all, and error messages will be saved to the "error.log" file.

Redirect All Output to /dev/null

Sometimes it's useful to get rid of all output. There are two ways to do this.

```
grep -r power /sys/ >/dev/null 2>&1
```

The string >/dev/null means "send stdout to /dev/null," and the second part, 2>&1 , means send stderr to stdout. In this case you have to refer to stdout as "&1" instead of simply "1." Writing "2>1" would just redirect stdout to a file named "1."

What's important to note here is that the order is important. If you reverse the redirect parameters like this:

```
grep -r power /sys/ 2>&1 >/dev/null
```

it won't work as intended. That's because as soon as 2>&1 is interpreted, stderr is sent to stdout and displayed on screen. Next, stdout is supressed when sent to "/dev/null." The final result is that you will see errors on the screen instead of suppressing all output. If you can't remember the correct order, there's a simpler redirect that is much easier to type:

```
grep -r power /sys/ &>/dev/null
```

In this case, &>/dev/null is equivalent to saying "redirect both stdout and stderr to this location."

Other Examples Where It Can Be Useful to Redirect to /dev/null

Say you want to see how fast your disk can read sequential data. The test is not extremely accurate but accurate enough. You can use dd for this, but dd either outputs to stdout or can be instructed to write to a file. With of=/dev/null you can tell dd to write to this virtual file. You don't even have to use shell redirections here. if= specifies the location of the input file to be read; of= specifies the name of the output file, where to write.

```
dd if=debian-disk.qcow2 of=/dev/null status=progress bs=1M iflag=direct
```

In some scenarios, you may want to see how fast you can download from a server. But you don't want to write to your disk unnecessarily. Simply enough, don't write to a regular file, write to "/dev/null."

```
wget -O /dev/null http://ftp.halifax.rwth-aachen.de/ubuntu-releases/18.04/ubuntu-18.04.2-desktop-amd64.iso
```

Conclusion

Hopefully, the examples in this article can inspire you to find your own creative ways to use "/dev/null."

Know an interesting use-case for this special device file? Leave a comment below and share the knowledge!

[Jan 29, 2019] hstr -- Bash and zsh shell history suggest box - easily view, navigate, search and manage your command history

This is quite useful command. RPM exists for CentOS7. You need to build on other versions.

Nov 17, 2018 | dvorka.github.io

View on GitHub

Configuration

Get most of HSTR by configuring it with:

```
hstr --show-configuration >> ~/.bashrc
```

Run hstr --show-configuration to determine what will be appended to your Bash profile. Don't forget to source ~/.bashrc to apply changes.

For more configuration options details please refer to:

- bind HSTR to a keyboard shortcut
  - Bash Emacs keymap (default)
  - Bash Vim keymap
  - zsh Emacs keymap (default)
- create hh alias for hstr
- het more colors
- choose default history view
- set filtering preferences
- configure commands blacklist
- disable confirm on delete
- tune verbosity
- history settings:
  - Bash history settings
  - zsh history settings

Check also configuration examples .

Binding HSTR to Keyboard Shortcut

Bash uses ***Emacs*** style keyboard shortcuts by default. There is also ***Vi*** mode. Find out how to bind HSTR to a keyboard shortcut based on the style you prefer below.

Check your active Bash keymap with:

```
bind -v | grep editing-mode
bind -v | grep keymap
```

To determine character sequence emitted by a pressed key in terminal, type Ctrl-v and then press the key. Check your current bindings using:

```
bind -S
```

Bash Emacs Keymap (default)

Bind HSTR to a Bash key e.g. to Ctrl-r :

```
bind '"\C-r": "\C-ahstr -- \C-j"'
```

or Ctrl-Altr :

```
bind '"\e\C-r":"\C-ahstr -- \C-j"'
```

or Ctrl-F12 :

```
bind '"\e[24;5~":"\C-ahstr -- \C-j"'
```

Bind HSTR to Ctrl-r only if it is interactive shell:

```
if [[ $- =~ .*i.* ]]; then bind '"\C-r": "\C-a hstr -- \C-j"'; fi
```

You can bind also other HSTR commands like --kill-last-command :

```
if [[ $- =~ .*i.* ]]; then bind '"\C-xk": "\C-a hstr -k \C-j"'; fi
```

Bash Vim Keymap

Bind HSTR to a Bash key e.g. to Ctrlr :

```
bind '"\C-r": "\e0ihstr -- \C-j"'
```

Zsh Emacs Keymap

Bind HSTR to a zsh key e.g. to Ctrlr :

```
bindkey -s "\C-r" "\eqhstr --\n"
```

Alias

If you want to make running of hstr from command line even easier, then define alias in your ~/.bashrc :

```
alias hh=hstr
```

Don't forget to source ~/.bashrc to be able to to use hh command.

Colors

Let HSTR to use colors:

```
export HSTR_CONFIG=hicolor
```

or ensure black and white mode:

```
export HSTR_CONFIG=monochromatic
```

Default History View

To show normal history by default (instead of metrics-based view, which is default) use:

```
export HSTR_CONFIG=raw-history-view
```

To show favorite commands as default view use:

```
export HSTR_CONFIG=favorites-view
```

Filtering

To use regular expressions based matching:

```
export HSTR_CONFIG=regexp-matching
```

To use substring based matching:

```
export HSTR_CONFIG=substring-matching
```

To use keywords (substrings whose order doesn't matter) search matching (default):

```
export HSTR_CONFIG=keywords-matching
```

Make search case sensitive (insensitive by default):

```
export HSTR_CONFIG=case-sensitive
```

Keep duplicates in raw-history-view (duplicate commands are discarded by default):

```
export HSTR_CONFIG=duplicates
```

Static favorites

Last selected favorite command is put the head of favorite commands list by default. If you want to disable this behavior and make favorite commands list static, then use the following configuration:

```
export HSTR_CONFIG=static-favorites
```

Skip favorites comments

If you don't want to show lines starting with # (comments) among favorites, then use the following configuration:

```
export HSTR_CONFIG=skip-favorites-comments
```

Blacklist

Skip commands when processing history i.e. make sure that these commands will ***not*** be shown in any view:

```
export HSTR_CONFIG=blacklist
```

Commands to be stored in ~/.hstr_blacklist file with trailing empty line. For instance:

```
cd
my-private-command
ls
ll
```

Confirm on Delete

Do not prompt for confirmation when deleting history items:

```
export HSTR_CONFIG=no-confirm
```

Verbosity

Show a message when deleting the last command from history:

```
export HSTR_CONFIG=verbose-kill
```

Show warnings:

```
export HSTR_CONFIG=warning
```

Show debug messages:

```
export HSTR_CONFIG=debug
```

Bash History Settings

Use the following Bash settings to get most out of HSTR.

Increase the size of history maintained by BASH - variables defined below increase the number of history items and history file size (default value is 500):

```
export HISTFILESIZE=10000
export HISTSIZE=${HISTFILESIZE}
```

Ensure syncing (flushing and reloading) of .bash_history with in-memory history:

```
export PROMPT_COMMAND="history -a; history -n; ${PROMPT_COMMAND}"
```

Force appending of in-memory history to .bash_history (instead of overwriting):

```
shopt -s histappend
```

Use leading space to hide commands from history:

```
export HISTCONTROL=ignorespace
```

Suitable for a sensitive information like passwords.

zsh History Settings

If you use zsh , set HISTFILE environment variable in ~/.zshrc :

```
export HISTFILE=~/.zsh_history
```

Examples

More colors with case sensitive search of history:

```
export HSTR_CONFIG=hicolor,case-sensitive
```

Favorite commands view in black and white with prompt at the bottom of the screen:

```
export HSTR_CONFIG=favorites-view,prompt-bottom
```

Keywords based search in colors with debug mode verbosity:

```
export HSTR_CONFIG=keywords-matching,hicolor,debug
```

[Jan 26, 2019] Ten Things I Wish I'd Known About about bash

Highly recommended!

Jan 06, 2018 | zwischenzugs.com

Intro

Recently I wanted to deepen my understanding of bash by researching as much of it as possible. Because I felt bash is an often-used (and under-understood) technology, I ended up writing a book on it .

A preview is available here .

You don't have to look hard on the internet to find plenty of useful one-liners in bash, or scripts. And there are guides to bash that seem somewhat intimidating through either their thoroughness or their focus on esoteric detail.

Here I've focussed on the things that either confused me or increased my power and productivity in bash significantly, and tried to communicate them (as in my book) in a way that emphasises getting the understanding right.

Enjoy!

1) `` vs $()

These two operators do the same thing. Compare these two lines:

```
$ echo `ls`
$ echo $(ls)
```

Why these two forms existed confused me for a long time.

If you don't know, both forms substitute the output of the command contained within it into the command.

The principal difference is that nesting is simpler.

Which of these is easier to read (and write)?

```
$ echo `echo \`echo \\\`echo inside\\\`\`
```

or:

```
$ echo $(echo $(echo $(echo inside)))
```

If you're interested in going deeper, see here or here .

2) globbing vs regexps

Another one that can confuse if never thought about or researched.

While globs and regexps can look similar, they are not the same.

Consider this command:

```
$ rename -n 's/(.*)/new$1/' *
```

The two asterisks are interpreted in different ways.

The first is ignored by the shell (because it is in quotes), and is interpreted as '0 or more characters' by the rename application. So it's interpreted as a regular expression.

The second is interpreted by the shell (because it is not in quotes), and gets replaced by a list of all the files in the current working folder. It is interpreted as a glob.

So by looking at man bash can you figure out why these two commands produce different output?

```
$ ls *
$ ls .*
```

The second looks even more like a regular expression. But it isn't!

3) Exit Codes

Not everyone knows that every time you run a shell command in bash, an 'exit code' is returned to bash.

Generally, if a command 'succeeds' you get an error code of 0 . If it doesn't succeed, you get a non-zero code. 1 is a 'general error', and others can give you more information (eg which signal killed it, for example).

But these rules don't always hold:

```
$ grep not_there /dev/null
$ echo $?
```

$? is a special bash variable that's set to the exit code of each command after it runs.

Grep uses exit codes to indicate whether it matched or not. I have to look up every time which way round it goes: does finding a match or not return 0 ?

Grok this and a lot will click into place in what follows.

4) if statements, [and [[

Here's another 'spot the difference' similar to the backticks one above.

What will this output?

```
if grep not_there /dev/null
then
   echo hi
else
   echo lo
fi
```

grep's return code makes code like this work more intuitively as a side effect of its use of exit codes.

Now what will this output?

a) hihi
b) lolo
c) something else

```
if [ $(grep not_there /dev/null) = '' ]
then
```

```
    echo -n hi
else
    echo -n lo
fi
if [[ $(grep not_there /dev/null) = '' ]]
then
    echo -n hi
else
    echo -n lo
fi
```

The difference between [and [[was another thing I never really understood. [is the original form for tests, and then [[was introduced, which is more flexible and intuitive. In the first if block above, the if statement barfs because the $(grep not_there /dev/null) is evaluated to nothing, resulting in this comparison:

```
[ = '' ]
```

which makes no sense. The double bracket form handles this for you.

This is why you occasionally see comparisons like this in bash scripts:

```
if [ x$(grep not_there /dev/null) = 'x' ]
```

so that if the command returns nothing it still runs. There's no need for it, but that's why it exists.

5) set s

Bash has configurable options which can be set on the fly. I use two of these all the time:

```
set -e
```

exits from a script if any command returned a non-zero exit code (see above).

This outputs the commands that get run as they run:

```
set -x
```

So a script might start like this:

```
#!/bin/bash
set -e
set -x
grep not_there /dev/null
echo $?
```

What would that script output?

6) <()

This is my favourite. It's so under-used, perhaps because it can be initially baffling, but I use it all the time.

It's similar to $() in that the output of the command inside is re-used.

In this case, though, the output is treated as a file. This file can be used as an argument to commands that take files as an argument.

Confused? Here's an example.

Have you ever done something like this?

```
$ grep somestring file1 > /tmp/a
$ grep somestring file2 > /tmp/b
$ diff /tmp/a /tmp/b
```

That works, but instead you can write:

```
diff <(grep somestring file1) <(grep somestring file2)
```

Isn't that neater?

7) Quoting

Quoting's a knotty subject in bash, as it is in many software contexts.

Firstly, variables in quotes:

```
A='123'
echo "$A"
echo '$A'
```

Pretty simple – double quotes dereference variables, while single quotes go literal.

So what will this output?

```
mkdir -p tmp
cd tmp
touch a
echo "*"
echo '*'
```

Surprised? I was.

8) Top three shortcuts

There are plenty of shortcuts listed in man bash , and it's not hard to find comprehensive lists. This list consists of the ones I use most often, in order of how often I use them.

Rather than trying to memorize them all, I recommend picking one, and trying to remember to use it until it becomes unconscious. Then take the next one. I'll skip over the most obvious ones (eg !! – repeat last command, and ~ – your home directory).

!$

I use this dozens of times a day. It repeats the last argument of the last command. If you're working on a file, and can't be bothered to re-type it command after command it can save a lot of work:

```
grep somestring /long/path/to/some/file/or/other.txt
vi !$
```

!:1-$

This bit of magic takes this further. It takes all the arguments to the previous command and drops them in. So:

```
grep isthere /long/path/to/some/file/or/other.txt
egrep !:1-$
fgrep !:1-$
```

The ! means 'look at the previous command', the : is a separator, and the 1 means 'take the first word', the - means 'until' and the $ means 'the last word'.

Note: you can achieve the same thing with !* . Knowing the above gives you the control to limit to a specific contiguous subset of arguments, eg with !:2-3 .

:h

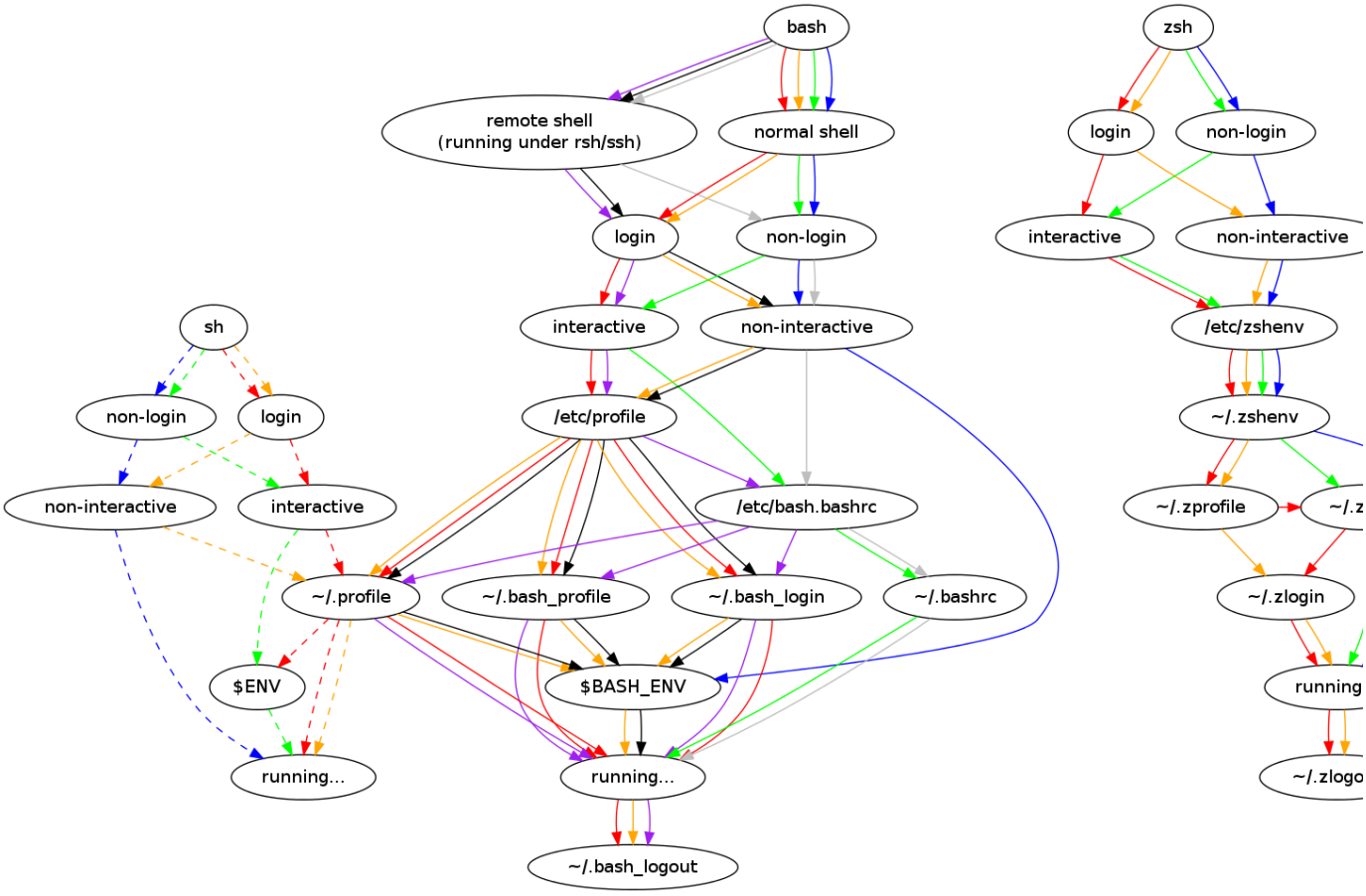
I use this one a lot too. If you put it after a filename, it will change that filename to remove everything up to the folder. Like this:

```
grep isthere /long/path/to/some/file/or/other.txt
cd !$:h
```

which can save a lot of work in the course of the day.

9) startup order

The order in which bash runs startup scripts can cause a lot of head-scratching. I keep this diagram handy (from this great page):



It shows which scripts bash decides to run from the top, based on decisions made about the context bash is running in (which decides the colour to follow).

So if you are in a local (non-remote), non-login, interactive shell (eg when you run bash itself from the command line), you are on the 'green' line, and these are the order of files read:

```
/etc/bash.bashrc
~/.bashrc
[bash runs, then terminates]
~/.bash_logout
```

This can save you a hell of a lot of time debugging.

10) getopts (cheapci)

If you go deep with bash, you might end up writing chunky utilities in it. If you do, then getting to grips with getopts can pay large dividends.

For fun, I once wrote a script called cheapci which I used to work like a Jenkins job.

The code here implements the reading of the two required, and 14 non-required arguments . Better to learn this than to build up a bunch of bespoke code that can get very messy pretty quickly as your utility grows.

This is based on some of the contents of my book Learn Bash the Hard Way , available at $7 :

[Nov 17, 2018] hh command man page

Later was renamed to hstr

Notable quotes:

"... By default it parses .bash-history file that is filtered as you type a command substring. ..."

"... Favorite and frequently used commands can be bookmarked ..."

Nov 17, 2018 | www.mankier.com

hh -- easily view, navigate, sort and use your command history with shell history suggest box.

Synopsis

hh [option] [arg1] [arg2]...
hstr [option] [arg1] [arg2]...

Description

hh uses shell history to provide suggest box like functionality for commands used in the past. ***By default it parses .bash-history file that is filtered as you type a command substring.*** Commands are not just filtered, but also ordered by a ranking algorithm that considers number of occurrences, length and timestamp. ***Favorite and frequently used commands can be bookmarked*** . In addition hh allows removal of commands from history - for instance with a typo or with a sensitive content.

Options

-h --help
Show help
-n --non-interactive
Print filtered history on standard output and exit
-f --favorites
Show favorites view immediately
-s --show-configuration
Show configuration that can be added to ~/.bashrc
-b --show-blacklist
Show blacklist of commands to be filtered out before history processing
-V --version
Show version information

Keys

pattern
Type to filter shell history.
Ctrl-e
Toggle regular expression and substring search.
Ctrl-t
Toggle case sensitive search.
Ctrl-/ , Ctrl-7
Rotate view of history as provided by Bash, ranked history ordered by the number of occurences/length/timestamp and favorites.
Ctrl-f
Add currently selected command to favorites.
Ctrl-l
Make search pattern lowercase or uppercase.
Ctrl-r , UP arrow, DOWN arrow, Ctrl-n , Ctrl-p
Navigate in the history list.
TAB , RIGHT arrow
Choose currently selected item for completion and let user to edit it on the command prompt.
LEFT arrow
Choose currently selected item for completion and let user to edit it in editor (fix command).
ENTER
Choose currently selected item for completion and execute it.
DEL
Remove currently selected item from the shell history.
BACSKSPACE , Ctrl-h
Delete last pattern character.
Ctrl-u , Ctrl-w
Delete pattern and search again.
Ctrl-x
Write changes to shell history and exit.
Ctrl-g
Exit with empty prompt.

Environment Variables

hh defines the following environment variables:

HH_CONFIG
Configuration options:

hicolor
Get more colors with this option (default is monochromatic).

monochromatic
Ensure black and white view.

prompt-bottom
Show prompt at the bottom of the screen (default is prompt at the top).

regexp
Filter command history using regular expressions (substring match is default)

substring
Filter command history using substring.

keywords
Filter command history using keywords - item matches if contains all keywords in pattern in any order.

casesensitive
Make history filtering case sensitive (it's case insensitive by default).

rawhistory
Show normal history as a default view (metric-based view is shown otherwise).

favorites
Show favorites as a default view (metric-based view is shown otherwise).

duplicates
Show duplicates in rawhistory (duplicates are discarded by default).

blacklist
Load list of commands to skip when processing history from ~/.hh_blacklist (built-in blacklist used otherwise).

big-keys-skip
Skip big history entries i.e. very long lines (default).

big-keys-floor
Use different sorting slot for big keys when building metrics-based view (big keys are skipped by default).

big-keys-exit
Exit (fail) on presence of a big key in history (big keys are skipped by default).

warning
Show warning.

debug
Show debug information.

Example:
export HH_CONFIG=hicolor,regexp,rawhistory

HH_PROMPT
Change prompt string which is user@host$ by default.

Example:
export HH_PROMPT="$ "

Files

~/.hh_favorites
Bookmarked favorite commands.
~/.hh_blacklist
Command blacklist.

Bash Configuration

Optionally add the following lines to ~/.bashrc:

```
export HH_CONFIG=hicolor         # get more colors
shopt -s histappend              # append new history items to .bash_history
export HISTCONTROL=ignorespace   # leading space hides commands from history
export HISTFILESIZE=10000        # increase history file size (default is 500)
export HISTSIZE=${HISTFILESIZE}  # increase history size (default is 500)
export PROMPT_COMMAND="history -a; history -n; ${PROMPT_COMMAND}"
# if this is interactive shell, then bind hh to Ctrl-r (for Vi mode check doc)
if [[ $- =~ .*i.* ]]; then bind '"\C-r": "\C-a hh -- \C-j"'; fi
```

The prompt command ensures synchronization of the history between BASH memory and history file.

ZSH Configuration

Optionally add the following lines to ~/.zshrc:

```
export HISTFILE=~/.zsh_history   # ensure history file visibility
export HH_CONFIG=hicolor         # get more colors
bindkey -s "\C-r" "\eqhh\n"  # bind hh to Ctrl-r (for Vi mode check doc, experiment with --)
```

Examples

hh git
Start `hh` and show only history items containing 'git'.
hh --non-interactive git
Print history items containing 'git' to standard output and exit.
hh --show-configuration >> ~/.bashrc
Append default hh configuration to your Bash profile.
hh --show-blacklist
Show blacklist configured for history processing.

Author

Written by Martin Dvorak <martin.dvorak@mindforger.com>

Bugs

Report bugs to https://github.com/dvorka/hstr/issues

See Also

history(1), bash(1), zsh(1)

Referenced By

The man page hstr(1) is an alias of hh(1).

[Oct 17, 2018] How to use arrays in bash script - LinuxConfig.org

Oct 17, 2018 | linuxconfig.org

Create indexed arrays on the fly We can create indexed arrays with a more concise syntax, by simply assign them some values:

```
$ my_array=(foo bar)
```

In this case we assigned multiple items at once to the array, but we can also insert one value at a time, specifying its index:

```
$ my_array[0]=foo
```

Array operations Once an array is created, we can perform some useful operations on it, like displaying its keys and values or modifying it by appending or removing elements: **Print the values of an array** To display all the values of an array we can use the following shell expansion syntax:

```
${my_array[@]}
```

Or even:

```
${my_array[*]}
```

Both syntax let us access all the values of the array and produce the same results, unless the expansion it's quoted. In this case a difference arises: in the first case, when using @ , the expansion will result in a word for each element of the array. This becomes immediately clear when performing a for loop . As an example, imagine we have an array with two elements, "foo" and "bar":

```
$ my_array=(foo bar)
```

Performing a for loop on it will produce the following result:

```
$ for i in "${my_array[@]}"; do echo "$i"; done
foo
bar
```

When using * , and the variable is quoted, instead, a single "result" will be produced, containing all the elements of the array:

```
$ for i in "${my_array[*]}"; do echo "$i"; done
foo bar
```

---

me name=

Print the keys of an array It's even possible to retrieve and print the keys used in an indexed or associative array, instead of their respective values. The syntax is almost identical, but relies on the use of the ! operator:

```
$ my_array=(foo bar baz)
$ for index in "${!my_array[@]}"; do echo "$index"; done
0
1
2
```

The same is valid for associative arrays:

```
$ declare -A my_array
$ my_array=([foo]=bar [baz]=foobar)
$ for key in "${!my_array[@]}"; do echo "$key"; done
baz
foo
```

As you can see, being the latter an associative array, we can't count on the fact that retrieved values are returned in the same order in which they were declared. **Getting the size of an array** We can retrieve the size of an array (the number of elements contained in it), by using a specific shell expansion:

```
$ my_array=(foo bar baz)
$ echo "the array contains ${#my_array[@]} elements"
the array contains 3 elements
```

We have created an array which contains three elements, "foo", "bar" and "baz", then by using the syntax above, which differs from the one we saw before to retrieve the array values only for the # character before the array name, we retrieved the number of the elements in the array instead of its content. **Adding elements to an array** As we saw, we can add elements to an indexed or associative array by specifying respectively their index or associative key. In the case of indexed arrays, we can also simply add an element, by appending to the end of the array, using the += operator:

```
$ my_array=(foo bar)
$ my_array+=(baz)
```

If we now print the content of the array we see that the element has been added successfully:

```
$ echo "${my_array[@]}"
foo bar baz
```

Multiple elements can be added at a time:

```
$ my_array=(foo bar)
$ my_array+=(baz foobar)
$ echo "${my_array[@]}"
foo bar baz foobar
```

To add elements to an associative array, we are bound to specify also their associated keys:

```
$ declare -A my_array

# Add single element
$ my_array[foo]="bar"

# Add multiple elements at a time
$ my_array+=([baz]=foobar [foobarbaz]=baz)
```

---

me name=

Deleting an element from the array To delete an element from the array we need to know it's index or its key in the case of an associative array, and use the unset command. Let's see an example:

```
$ my_array=(foo bar baz)
$ unset my_array[1]
$ echo ${my_array[@]}
foo baz
```

We have created a simple array containing three elements, "foo", "bar" and "baz", then we deleted "bar" from it running unset and referencing the index of "bar" in the array: in this case we know it was 1 , since bash arrays start at 0. If we check the indexes of the array, we can now see that 1 is missing:

```
$ echo ${!my_array[@]}
0 2
```

The same thing it's valid for associative arrays:

```
$ declare -A my_array
$ my_array+=([foo]=bar [baz]=foobar)
$ unset my_array[foo]
$ echo ${my_array[@]}
foobar
```

In the example above, the value referenced by the "foo" key has been deleted, leaving only "foobar" in the array.

Deleting an entire array, it's even simpler: we just pass the array name as an argument to the unset command without specifying any index or key:

```
$ unset my_array
$ echo ${!my_array[@]}
```

After executing unset against the entire array, when trying to print its content an empty result is returned: the array doesn't exist anymore. **Conclusions** In this tutorial we saw the difference between indexed and associative arrays in bash, how to initialize them and how to perform fundamental operations, like displaying their keys and values and appending or removing items. Finally we saw how to unset them completely. Bash syntax can sometimes be pretty weird, but using arrays in scripts can be really useful. When a script starts to become more complex than expected, my advice is, however, to switch to a more capable scripting language such as python.

[Oct 10, 2018] Bash History Display Date And Time For Each Command

Oct 10, 2018 | www.cyberciti.biz

1. **Abhijeet Vaidya** says: March 11, 2010 at 11:41 am End single quote is missing.
   Correct command is:

   ```
   echo 'export HISTTIMEFORMAT="%d/%m/%y %T "' >> ~/.bash_profile
   ```

2. **izaak** says: March 12, 2010 at 11:06 am I would also add
   ```
   $ echo 'export HISTSIZE=10000' >> ~/.bash_profile
   ```
   It's really useful, I think.

3. **Dariusz** says: March 12, 2010 at 2:31 pm you can add it to /etc/profile so it is available to all users. I also add:

   ```
   # Make sure all terminals save history
   shopt -s histappend histreedit histverify
   shopt -s no_empty_cmd_completion # bash>=2.04 only
   ```

   # Whenever displaying the prompt, write the previous line to disk:

   ```
   PROMPT_COMMAND='history -a'
   ```

   #Use GREP color features by default: This will highlight the matched words / regexes
   export GREP_OPTIONS='–color=auto'
   export GREP_COLOR='1;37;41′

4. **Babar Haq** says: March 15, 2010 at 6:25 am Good tip. We have multiple users connecting as root using ssh and running different commands. Is there a way to log the IP that command was run from?
   Thanks in advance.
   1. **Anthony** says: August 21, 2014 at 9:01 pm Just for anyone who might still find this thread (like I did today):

      export HISTTIMEFORMAT="%F %T : $(echo $SSH_CONNECTION | cut -d\ -f1) : "

      will give you the time format, plus the IP address culled from the ssh_connection environment variable (thanks for pointing that out, Cadrian, I never knew about that before), all right there in your history output.

      You could even add in $(whoami)@ right to get if you like (although if everyone's logging in with the root account that's not helpful).

5. **cadrian** says: March 16, 2010 at 5:55 pm Yup, you can export one of this

   env | grep SSH
   SSH_CLIENT=192.168.78.22 42387 22
   SSH_TTY=/dev/pts/0
   SSH_CONNECTION=192.168.78.22 42387 192.168.36.76 22

   As their bash history filename

   set |grep -i hist
   HISTCONTROL=ignoreboth
   HISTFILE=/home/cadrian/.bash_history
   HISTFILESIZE=1000000000
   HISTSIZE=10000000

   So in profile you can so something like HISTFILE=/root/.bash_history_$(echo $SSH_CONNECTION| cut -d\ -f1)

6. **TSI** says: March 21, 2010 at 10:29 am bash 4 can syslog every command bat afaik, you have to recompile it (check file config-top.h). See the news file of bash: http://tiswww.case.edu/php/chet/bash/NEWS
   If you want to safely export history of your luser, you can ssl-syslog them to a central syslog server.
7. **Dinesh Jadhav** says: November 12, 2010 at 11:00 am This is good command, It helps me a lot.
8. **Indie** says: September 19, 2011 at 11:41 am You only need to use

   ```
   export HISTTIMEFORMAT='%F %T '
   ```

   in your .bash_profile

9. **lalit jain** says: October 3, 2011 at 9:58 am -- show history with date & time

   # HISTTIMEFORMAT='%c '
   #history

10. **Sohail** says: January 13, 2012 at 7:05 am Hi
    Nice trick but unfortunately, the commands which were executed in the past few days also are carrying the current day's (today's) timestamp.

    Please advice.

    Regards

    1. **Raymond** says: March 15, 2012 at 9:05 am Hi Sohail,

       Yes indeed that will be the behavior of the system since you have just enabled on that day the HISTTIMEFORMAT feature. In other words, the system recall or record the commands which were inputted prior enabling of this feature. Hope this answers your concern.

       Thanks!

       1. **Raymond** says: March 15, 2012 at 9:08 am Hi Sohail,

          Yes, that will be the behavior of the system since you have just enabled on that day the HISTTIMEFORMAT feature. In other words, the system can't recall or record the commands which were inputted prior enabling of this feature, thus it will just reflect on the printed output (upon execution of "history") the current day and time. Hope this answers your concern.

          Thanks!

11. **Sohail** says: February 24, 2012 at 6:45 am Hi

    The command only lists the current date (Today) even for those commands which were executed on earlier days.

    Any solutions ?

    Regards

12. **nitiratna nikalje** says: August 24, 2012 at 5:24 pm hi vivek.do u know any openings for freshers in linux field? I m doing rhce course from rajiv banergy. My samba,nfs-nis,dhcp,telnet,ftp,http,ssh,squid,cron,quota and system administration is over.iptables ,sendmail and dns is remaining.

    -9029917299(Nitiratna)

13. **JMathew** says: August 26, 2012 at 10:51 pm Hi,

    Is there anyway to log username also along with the Command Which we typed

    Thanks in Advance

14. **suresh** says: May 22, 2013 at 1:42 pm How can i get full comman along with data and path as we het in history command.
15. **rajesh** says: December 6, 2013 at 5:56 am Thanks it worked..
16. **Krishan** says: February 7, 2014 at 6:18 am The command is not working properly. It is displaying the date and time of todays for all the commands where as I ran the some command three before.

    How come it is displaying the today date

17. **PR** says: April 29, 2014 at 5:18 pm Hi..

    I want to collect the history of particular user everyday and want to send an email.I wrote below script.
    for collecting everyday history by time shall i edit .profile file of that user
    echo 'export HISTTIMEFORMAT="%d/%m/%y %T "' >> ~/.bash_profile
    Script:

    ```
    #!/bin/bash
    #This script sends email of particular user
    history >/tmp/history
    if [ -s /tmp/history ]
    then
        mailx -s "history 29042014"  </tmp/history
          fi
    rm /tmp/history
    #END OF THE SCRIPT
    ```

    Can any one suggest better way to collect particular user history for everyday

18. **lefty.crupps** says: October 24, 2014 at 7:10 pm Love it, but using the ISO date format is always recommended (YYYY-MM-DD), just as every other sorted group goes from largest sorting (Year) to smallest sorting (day)
    https://en.wikipedia.org/wiki/ISO_8601#Calendar_dates

    In that case, myne looks like this:
    echo 'export HISTTIMEFORMAT="%YY-%m-%d/ %T "' >> ~/.bashrc

    Thanks for the tip!

    1. **lefty.crupps** says: October 24, 2014 at 7:11 pm please delete post 33, my command is messed up.
19. **lefty.crupps** says: October 24, 2014 at 7:11 pm Love it, but using the ISO date format is always recommended (YYYY-MM-DD), just as every other sorted group goes from largest sorting (Year) to smallest sorting (day)
    https://en.wikipedia.org/wiki/ISO_8601#Calendar_dates

    In that case, myne looks like this:
    echo â€˜export HISTTIMEFORMAT=â€%Y-%m-%d %T â€œâ€˜ >> ~/.bashrc

    Thanks for the tip!

20. **Vanathu** says: October 30, 2014 at 1:01 am its show only current date for all the command history
    1. **lefty.crupps** says: October 30, 2014 at 2:08 am it's marking all of your current history with today's date. Try checking again in a few days.
21. **tinu** says: October 14, 2015 at 3:30 pm Hi All,

I Have enabled my history with the command given :
echo 'export HISTTIMEFORMAT="%d/%m/%y %T "' >> ~/.bash_profile

i need to know how i can add the ip's also , from which the commands are fired to the system.

[Jun 09, 2018] How to use the history command in Linux Opensource.com

Jun 09, 2018 | opensource.com

Changing an executed command

history also allows you to rerun a command with different syntax. For example, if I wanted to change my previous command history | grep dnf to history | grep ssh , I can execute the following at the prompt:

```
$ ^dnf^ssh^
```

history will rerun the command, but replace dnf with ssh , and execute it.

Removing history

There may come a time that you want to remove some or all the commands in your history file. If you want to delete a particular command, enter history -d <line number> . To clear the entire contents of the history file, execute history -c .

The history file is stored in a file that you can modify, as well. Bash shell users will find it in their Home directory as .bash_history .

Next steps

There are a number of other things that you can do with history :

- Set the size of your history buffer to a certain number of commands
- Record the date and time for each line in history
- Prevent certain commands from being recorded in history

For more information about the history command and other interesting things you can do with it, take a look at the GNU Bash Manual .

[Jun 01, 2018] Introduction to Bash arrays by Robert Aboukhalil

Jun 01, 2018 | opensource.com

... ...

Looping through arrays

Although in the examples above we used integer indices in our arrays, let's consider two occasions when that won't be the case: First, if we wanted the $i -th element of the array, where $i is a variable containing the index of interest, we can retrieve that element using: echo ${allThreads[$i]} . Second, to output all the elements of an array, we replace the numeric index with the @ symbol (you can think of @ as standing for all): echo ${allThreads[@]} .

Looping through array elements

With that in mind, let's loop through $allThreads and launch the pipeline for each value of --threads :

```
for t in ${allThreads[@]} ; do
. / pipeline --threads $t
done
```

Looping through array indices

Next, let's consider a slightly different approach. Rather than looping over array ***elements*** , we can loop over array ***indices*** :

```
for i in ${!allThreads[@]} ; do
. / pipeline --threads ${allThreads[$i]}
done
```

Let's break that down: As we saw above, ${allThreads[@]} represents all the elements in our array. Adding an exclamation mark to make it ${!allThreads[@]} will return the list of all array indices (in our case 0 to 7). In other words, the for loop is looping through all indices $i and reading the $i -th element from $allThreads to set the value of the --threads parameter.

This is much harsher on the eyes, so you may be wondering why I bother introducing it in the first place. That's because there are times where you need to know both the index and the value within a loop, e.g., if you want to ignore the first element of an array, using indices saves you from creating an additional variable that you then increment inside the loop.

Populating arrays

So far, we've been able to launch the pipeline for each --threads of interest. Now, let's assume the output to our pipeline is the runtime in seconds. We would like to capture that output at each iteration and save it in another array so we can do various manipulations with it at the end.

Some useful syntax

But before diving into the code, we need to introduce some more syntax. First, we need to be able to retrieve the output of a Bash command. To do so, use the following syntax: output=$(./my_script.sh) , which will store the output of our commands into the variable $output .

The second bit of syntax we need is how to append the value we just retrieved to an array. The syntax to do that will look familiar:

```
myArray+=( "newElement1" "newElement2" )
```

The parameter sweep

Putting everything together, here is our script for launching our parameter sweep:

```
allThreads = ( 1 2 4 8 16 32 64 128 )
allRuntimes = ()
for t in ${allThreads[@]} ; do
runtime =$ ( . / pipeline --threads $t )
```

```
allRuntimes+= ( $runtime )
done
```

And voilà!

What else you got?

In this article, we covered the scenario of using arrays for parameter sweeps. But I promise there are more reasons to use Bash arrays -- here are two more examples.

Log alerting

In this scenario, your app is divided into modules, each with its own log file. We can write a cron job script to email the right person when there are signs of trouble in certain modules:

```
# List of logs and who should be notified of issues
logPaths = ( "api.log" "auth.log" "jenkins.log" "data.log" )
logEmails = ( "jay@email" "emma@email" "jon@email" "sophia@email" )

# Look for signs of trouble in each log
for i in ${!logPaths[@]} ;
do
log = ${logPaths[$i]}
stakeholder = ${logEmails[$i]}
numErrors =$ ( tail -n 100 " $log " | grep "ERROR" | wc -l )

# Warn stakeholders if recently saw > 5 errors
if [[ " $numErrors " -gt 5 ]] ;
then
emailRecipient = " $stakeholder "
emailSubject = "WARNING: ${log} showing unusual levels of errors"
emailBody = " ${numErrors} errors found in log ${log} "
echo " $emailBody " | mailx -s " $emailSubject " " $emailRecipient "
fi
done
```

API queries

Say you want to generate some analytics about which users comment the most on your Medium posts. Since we don't have direct database access, SQL is out of the question, but we can use APIs!

To avoid getting into a long discussion about API authentication and tokens, we'll instead use JSONPlaceholder , a public-facing API testing service, as our endpoint. Once we query each post and retrieve the emails of everyone who commented, we can append those emails to our results array:

```
endpoint = "https://jsonplaceholder.typicode.com/comments"
allEmails = ()

# Query first 10 posts
for postId in { 1 .. 10 } ;
do
# Make API call to fetch emails of this posts's commenters
response =$ ( curl " ${endpoint} ?postId= ${postId} " )

# Use jq to parse the JSON response into an array
allEmails+= ( $ ( jq '.[].email' <<< " $response " ) )
done
```

Note here that I'm using the jq tool to parse JSON from the command line. The syntax of jq is beyond the scope of this article, but I highly recommend you look into it.

As you might imagine, there are countless other scenarios in which using Bash arrays can help, and I hope the examples outlined in this article have given you some food for thought. If you have other examples to share from your own work, please leave a comment below.

But wait, there's more!

Since we covered quite a bit of array syntax in this article, here's a summary of what we covered, along with some more advanced tricks we did not cover:

Syntax	Result
arr=()	Create an empty array
arr=(1 2 3)	Initialize array
${arr[2]}	Retrieve third element
${arr[@]}	Retrieve all elements
${!arr[@]}	Retrieve array indices
${#arr[@]}	Calculate array size
arr[0]=3	Overwrite 1st element
arr+=(4)	Append value(s)
str=$(ls)	Save ls output as a string
arr=($(ls))	Save ls output as an array of files
${arr[@]:s:n}	Retrieve elements at indices n to s+n

One last thought

As we've discovered, Bash arrays sure have strange syntax, but I hope this article convinced you that they are extremely powerful. Once you get the hang of the syntax, you'll find yourself using Bash arrays quite often.

... ...

Robert Aboukhalil is a Bioinformatics Software Engineer. In his work, he develops cloud applications for the analysis and interactive visualization of genomics data. Robert holds a Ph.D. in Bioinformatics from Cold Spring Harbor Laboratory and a B.Eng. in Computer Engineering from McGill.

[May 28, 2018] Useful Linux Command Line Bash Shortcuts You Should Know

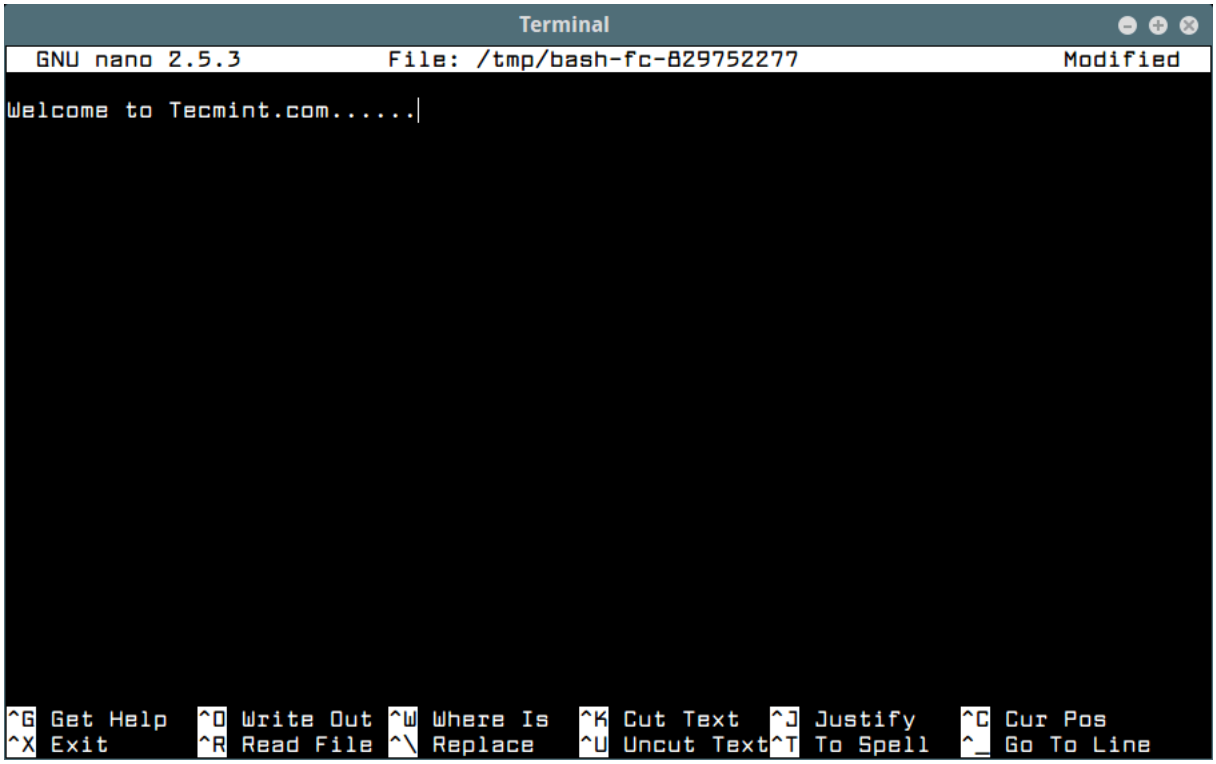
May 28, 2018 | www.tecmint.com

In this article, we will share a number of Bash command-line shortcuts useful for any Linux user. These shortcuts allow you to easily and in a fast manner, perform certain activities such as accessing and running previously executed commands, opening an editor, editing/deleting/changing text on the command line, moving the cursor, controlling processes etc. on the command line.

Although this article will mostly benefit Linux beginners getting their way around with command line basics, those with intermediate skills and advanced users might also find it practically helpful. We will group the bash keyboard shortcuts according to categories as follows.

Launch an Editor

Open a terminal and press Ctrl+X and Ctrl+E to open an editor (nano editor) with an empty buffer. Bash will try to launch the editor defined by the $EDITOR environment variable.



Nano Editor **Controlling The Screen**

These shortcuts are used to control terminal screen output:

- Ctrl+L – clears the screen (same effect as the " clear " command).
- Ctrl+S – pause all command output to the screen. If you have executed a command that produces verbose, long output, use this to pause the output scrolling down the screen.
- Ctrl+Q – resume output to the screen after pausing it with Ctrl+S .

Move Cursor on The Command Line

The next shortcuts are used for moving the cursor within the command-line:

- Ctrl+A or Home – moves the cursor to the start of a line.
- Ctrl+E or End – moves the cursor to the end of the line.
- Ctrl+B or Left Arrow – moves the cursor back one character at a time.
- Ctrl+F or Right Arrow – moves the cursor forward one character at a time.
- Ctrl + Left Arrow or Alt+B or Esc and then B – moves the cursor back one word at a time.
- Ctrl + Right Arrow or Alt+C or Esc and then F – moves the cursor forward one word at a time.

Search Through Bash History

The following shortcuts are used for searching for commands in the bash history:

- Up arrow key – retrieves the previous command. If you press it constantly, it takes you through multiple commands in history, so you can find the one you want. Use the Down arrow to move in the reverse direction through the history.
- Ctrl+P and Ctrl+N – alternatives for the Up and Down arrow keys, respectively.
- Ctrl+R – starts a reverse search, through the bash history, simply type characters that should be unique to the command you want to find in the history.
- Ctrl+S – launches a forward search, through the bash history.
- Ctrl+G – quits reverse or forward search, through the bash history.

Delete Text on the Command Line

The following shortcuts are used for deleting text on the command line:

- Ctrl+D or Delete – remove or deletes the character under the cursor.
- Ctrl+K – removes all text from the cursor to the end of the line.
- Ctrl+X and then Backspace – removes all the text from the cursor to the beginning of the line.

Transpose Text or Change Case on the Command Line

These shortcuts will transpose or change the case of letters or words on the command line:

- Ctrl+T – transposes the character before the cursor with the character under the cursor.
- Esc and then T – transposes the two words immediately before (or under) the cursor.
- Esc and then U – transforms the text from the cursor to the end of the word to uppercase.
- Esc and then L – transforms the text from the cursor to the end of the word to lowercase.
- Esc and then C – changes the letter under the cursor (or the first letter of the next word) to uppercase, leaving the rest of the word unchanged.

Working With Processes in Linux

The following shortcuts help you to control running Linux processes.

- Ctrl+Z – suspend the current foreground process. This sends the SIGTSTP signal to the process. You can get the process back to the foreground later using the fg process_name (or %bgprocess_number like %1 , %2 and so on) command.

- Ctrl+C – interrupt the current foreground process, by sending the SIGINT signal to it. The default behavior is to terminate a process gracefully, but the process can either honor or ignore it.
- Ctrl+D – exit the bash shell (same as running the exit command).

Learn more about: All You Need To Know About Processes in Linux [Comprehensive Guide]

Bash Bang (!) Commands

In the final part of this article, we will explain some useful ! (bang) operations:

- !! – execute last command.
- !top – execute the most recent command that starts with 'top' (e.g. !).
- !top:p – displays the command that !top would run (also adds it as the latest command in the command history).
- !$ – execute the last word of the previous command (same as Alt + ., e.g. if last command is ' cat tecmint.txt ', then !$ would try to run ' tecmint.txt ').
- !$:p – displays the word that !$ would execute.
- !* – displays the last word of the previous command.
- !*:p – displays the last word that !* would substitute.

For more information, see the bash man page:

```
$ man bash
```

That's all for now! In this article, we shared some common and useful Bash command-line shortcuts and operations. Use the comment form below to make any additions or ask questions.

[Oct 31, 2017] High-speed Bash by Tom Ryder

Notable quotes:

"... One of my favourite technical presentations I've read online has been Hal Pomeranz's Unix Command-Line Kung Fu , a catalogue of shortcuts and efficient methods of doing very clever things with the Bash shell. None of these are grand arcane secrets, but they're things that are often forgotten in the course of daily admin work, when you find yourself typing something you needn't, or pressing up repeatedly to find something you wrote for which you could simply search your command history. ..."

Jan 24, 2012 | sanctum.geek.nz

One of my favourite technical presentations I've read online has been Hal Pomeranz's Unix Command-Line Kung Fu , a catalogue of shortcuts and efficient methods of doing very clever things with the Bash shell. None of these are grand arcane secrets, but they're things that are often forgotten in the course of daily admin work, when you find yourself typing something you needn't, or pressing up repeatedly to find something you wrote for which you could simply search your command history.

I highly recommend reading the whole thing, as I think even the most experienced shell users will find there are useful tidbits in there that would make their lives easier and their time with the shell more productive, beyond simpler things like tab completion.

Here, I'll recap two of the things I thought were the most simple and useful items in the presentation for general shell usage, and see if I can add a little value to them with reference to the Bash manual.

History with Ctrl+R

For many shell users, finding a command in history means either pressing the up arrow key repeatedly, or perhaps piping a history call through grep . It turns out there's a much nicer way to do this, using Bash's built-in history searching functionality; if you press Ctrl+R and start typing a search pattern, the most recent command matching that pattern will automatically be inserted on your current line, at which point you can adapt it as you need, or simply press Enter to run it again. You can keep pressing Ctrl+R to move further back in your history to the next-most recent match. On my shell, if I search through my history for git , I can pull up what I typed for a previous commit:

```
(reverse-i-search)`git': git commit -am "Pulled up-to-date colors."
```

This functionality isn't actually exclusive to Bash; you can establish a history search function in quite a few tools that use GNU Readline, including the MySQL client command line.

You can search forward through history in the same way with Ctrl+S, but it's likely you'll have to fix up a couple of terminal annoyances first.

Additionally, if like me you're a Vim user and you don't really like having to reach for the arrow keys, or if you're on a terminal where those keys are broken for whatever reason, you can browse back and forth within your command history with Ctrl+P (previous) and Ctrl+N (next). These are just a few of the Emacs-style shortcuts that GNU Readline provides; check here for a more complete list .

Repeating commands with !!

The last command you ran in Bash can be abbreviated on the next line with two exclamation marks:

```
$ echo "Testing."
Testing.
$ !!
Testing.
```

You can use this to simply repeat a command over and over again, although for that you really should be using watch , but more interestingly it turns out this is very handy for building complex pipes in stages. Suppose you were building a pipeline to digest some data generated from a program like netstat , perhaps to determine the top 10 IP addresses that are holding open the most connections to a server. You might be able to build a pipeline like this:

```
# netstat -ant
# !! | awk '{print $5}'
# !! | sort
# !! | uniq -c
# !! | sort -rn
# !! | sed 10q
```

Similarly, you can repeat the last ***argument*** from the previous command line using !$, which is useful if you're doing a set of operations on one file, such as checking it out via RCS, editing it, and checking it back in:

```
$ co -l file.txt
$ vim !$
$ ci -u !$
```

Or if you happen to want to work on a set of arguments, you can repeat ***all*** of the arguments from the previous command using !* :

```
$ touch a.txt b.txt c.txt
$ rm !*
```

When you remember to user these three together, they can save you a lot of typing, and will really increase your accuracy because you won't be at risk of mistyping any of the commands or arguments. Naturally, however, it pays to be careful what you're running through rm !

[Oct 31, 2017] Learning the content of /bin and /usr/bin by Tom Ryder

Mar 16, 2012 | sanctum.geek.nz

When you have some spare time, something instructive to do that can help fill gaps in your Unix knowledge and to get a better idea of the programs installed on your system and what they can do is a simple whatis call, run over all the executable files in your /bin and /usr/bin directories.

This will give you a one-line summary of the file's function if available from man pages.

```
tom@conan:/bin$ whatis *
bash (1) - GNU Bourne-Again SHell
bunzip2 (1) - a block-sorting file compressor, v1.0.4
busybox (1) - The Swiss Army Knife of Embedded Linux
bzcat (1) - decompresses files to stdout
...

tom@conan:/usr/bin$ whatis *
[ (1)                - check file types and compare values
2to3 (1)             - Python2 to Python3 converter
2to3-2.7 (1)         - Python2 to Python3 converter
411toppm (1)         - convert Sony Mavica .411 image to ppm
...
```

It also works on many of the files in other directories, such as /etc :

```
tom@conan:/etc$ whatis *
acpi (1)             - Shows battery status and other ACPI information
adduser.conf (5)     - configuration file for adduser(8) and addgroup(8)
adjtime (3)          - correct the time to synchronize the system clock
aliases (5)          - Postfix local alias database format
...
```

Because packages often install more than one binary and you're only in the habit of using one or two of them, this process can tell you about programs on your system that you may have missed, particularly standard tools that solve common problems. As an example, I first learned about watch this way, having used a clunky solution with for loops with sleep calls to do the same thing many times before.

[Oct 31, 2017] Shell config subfiles by Tom Ryder

Notable quotes:

"... Note that we unset the config variable after we're done, otherwise it'll be in the namespace of our shell where we don't need it. You may also wish to check for the existence of the ~/.bashrc.d directory, check there's at least one matching file inside it, or check that the file is readable before attempting to source it, depending on your preference. ..."

"... Thanks to commenter oylenshpeegul for correcting the syntax of the loops. ..."

Jan 30, 2015 | sanctum.geek.nz

Large shell startup scripts (.bashrc , .profile) over about fifty lines or so with a lot of options, aliases, custom functions, and similar tweaks can get cumbersome to manage over time, and if you keep your dotfiles under version control it's not terribly helpful to see large sets of commits just editing the one file when it could be more instructive if broken up into files by section.

Given that shell configuration is just shell code, we can apply the source builtin (or the . builtin for POSIX sh) to load several files at the end of a .bashrc , for example:

```
source ~/.bashrc.options
source ~/.bashrc.aliases
source ~/.bashrc.functions
```

This is a better approach, but it still binds us into using those filenames; we still have to edit the ~/.bashrc file if we want to rename them, or remove them, or add new ones.

Fortunately, UNIX-like systems have a common convention for this, the .d directory suffix, in which sections of configuration can be stored to be read by a main configuration file dynamically. In our case, we can create a new directory ~/.bashrc.d :

```
$ ls ~/.bashrc.d
options.bash
aliases.bash
functions.bash
```

With a slightly more advanced snippet at the end of ~/.bashrc , we can then load every file with the suffix .bash in this directory:

```
# Load any supplementary scripts
for config in "$HOME"/.bashrc.d/*.bash ; do
  source "$config"
```

```
done
unset -v config
```

Note that we unset the config variable after we're done, otherwise it'll be in the namespace of our shell where we don't need it. You may also wish to check for the existence of the ~/.bashrc.d directory, check there's at least one matching file inside it, or check that the file is readable before attempting to source it, depending on your preference.

The same method can be applied with .profile to load all scripts with the suffix .sh in ~/.profile.d , if we want to write in POSIX sh , with some slightly different syntax:

```
# Load any supplementary scripts
for config in "$HOME"/.profile.d/*.sh ; do
    . "$config"
done
unset -v config
```

Another advantage of this method is that if you have your dotfiles under version control, you can arrange to add extra snippets on a per-machine basis unversioned, without having to update your .bashrc file.

Here's my implementation of the above method, for both .bashrc and .profile :

- .bashrc
- .bashrc.d
- .profile
- .profile.d

Thanks to commenter oylenshpeegul for correcting the syntax of the loops.

[Oct 31, 2017] Better Bash history by Tom Ryder

Feb 21, 2012 | sanctum.geek.nz

By default, the Bash shell keeps the history of your most recent session in the .bash_history file, and the commands you've issued in your current session are also available with a history call. These defaults are useful for keeping track of what you've been up to in the shell on any given machine, but with disks much larger and faster than they were when Bash was designed, a little tweaking in your .bashrc file can record history more permanently, consistently, and usefully. **Append history instead of rewriting it**

You should start by setting the histappend option, which will mean that when you close a session, your history will be ***appended*** to the .bash_history file rather than overwriting what's in there.

```
shopt -s histappend
```

Allow a larger history file

The default maximum number of commands saved into the .bash_history file is a rather meager 500. If you want to keep history further back than a few weeks or so, you may as well bump this up by explicitly setting $HISTSIZE to a much larger number in your .bashrc . We can do the same thing with the $HISTFILESIZE variable.

```
HISTFILESIZE=1000000
HISTSIZE=1000000
```

The man page for Bash says that HISTFILESIZE can be unset to stop truncation entirely, but unfortunately this doesn't work in .bashrc files due to the order in which variables are set; it's therefore more straightforward to simply set it to a very large number.

If you're on a machine with resource constraints, it might be a good idea to occasionally archive old .bash_history files to speed up login and reduce memory footprint.

Don't store specific lines

You can prevent commands that start with a space from going into history by setting $HISTCONTROL to ignorespace . You can also ignore duplicate commands, for example repeated du calls to watch a file grow, by adding ignoredups . There's a shorthand to set both in ignoreboth .

```
HISTCONTROL=ignoreboth
```

You might also want to remove the use of certain commands from your history, whether for privacy or readability reasons. This can be done with the $HISTIGNORE variable. It's common to use this to exclude ls calls, job control builtins like bg and fg , and calls to history itself:

```
HISTIGNORE='ls:bg:fg:history'
```

Record timestamps

If you set $HISTTIMEFORMAT to something useful, Bash will record the timestamp of each command in its history. In this variable you can specify the format in which you want this timestamp displayed when viewed with history . I find the full date and time to be useful, because it can be sorted easily and works well with tools like cut and awk .

```
HISTTIMEFORMAT='%F %T '
```

Use one command per line

To make your .bash_history file a little easier to parse, you can force commands that you entered on more than one line to be adjusted to fit on only one with the cmdhist option:

```
shopt -s cmdhist
```

Store history immediately

By default, Bash only records a session to the .bash_history file on disk when the session terminates. This means that if you crash or your session terminates improperly, you lose the history up to that point. You can fix this by recording each line of history as you issue it, through the $PROMPT_COMMAND variable:

```
PROMPT_COMMAND='history -a'
```

[Oct 31, 2017] Bash history expansion by Tom Ryder

Notable quotes:

"... Thanks to commenter Mihai Maruseac for pointing out a bug in the examples. ..."

Aug 16, 2012 | sanctum.geek.nz

Setting the Bash option histexpand allows some convenient typing shortcuts using Bash history expansion . The option can be set with either of these:

```
$ set -H
$ set -o histexpand
```

It's likely that this option is already set for all interactive shells, as it's on by default. The manual, man bash , describes these features as follows:

```
-H  Enable ! style history substitution. This option is on
   by default when the shell is interactive.
```

You may have come across this before, perhaps to your annoyance, in the following error message that comes up whenever ! is used in a double-quoted string, or without being escaped with a backslash:

```
$ echo "Hi, this is Tom!"
bash: !": event not found
```

If you don't want the feature and thereby make ! into a normal character, it can be disabled with either of these:

```
$ set +H
$ set +o histexpand
```

History expansion is actually a very old feature of shells, having been available in csh before Bash usage became common.

This article is a good followup to Better Bash history , which among other things explains how to include dates and times in history output, as these examples do.

Basic history expansion

Perhaps the best known and most useful of these expansions is using !! to refer to the previous command. This allows repeating commands quickly, perhaps to monitor the progress of a long process, such as disk space being freed while deleting a large file:

```
$ rm big_file &
[1] 23608
$ du -sh .
3.9G    .
$ !!
du -sh .
3.3G    .
```

It can also be useful to specify the full filesystem path to programs that aren't in your $PATH :

```
$ hdparm
-bash: hdparm: command not found
$ /sbin/!!
/sbin/hdparm
```

In each case, note that the command itself is printed as expanded, and then run to print the output on the following line.

History by absolute index

However, !! is actually a specific example of a more general form of history expansion. For example, you can supply the history item number of a specific command to repeat it, after looking it up with history :

```
$ history | grep expand
 3951  2012-08-16 15:58:53  set -o histexpand
$ !3951
set -o histexpand
```

You needn't enter the !3951 on a line by itself; it can be included as any part of the command, for example to add a prefix like sudo :

```
$ sudo !3850
```

If you include the escape string \! as part of your Bash prompt , you can include the current command number in the prompt before the command, making repeating commands by index a lot easier as long as they're still visible on the screen.

History by relative index

It's also possible to refer to commands ***relative*** to the current command. To subtitute the second-to-last command, we can type !-2 . For example, to check whether truncating a file with sed worked correctly:

```
$ wc -l bigfile.txt
267 bigfile.txt
$ printf '%s\n' '11,$d' w | ed -s bigfile.txt
$ !-2
wc -l bigfile.txt
10 bigfile.txt
```

This works further back into history, with !-3 , !-4 , and so on.

Expanding for historical arguments

In each of the above cases, we're substituting for the whole command line. There are also ways to get specific tokens, or ***words*** , from the command if we want that. To get the ***first*** argument of a particular command in the history, use the !^ token:

```
$ touch a.txt b.txt c.txt
$ ls !^
ls a.txt
a.txt
```

To get the ***last*** argument, add !$:

```
$ touch a.txt b.txt c.txt
$ ls !$
ls c.txt
c.txt
```

To get ***all*** arguments (but not the command itself), use !* :

```
$ touch a.txt b.txt c.txt
$ ls !*
ls a.txt b.txt c.txt
a.txt  b.txt  c.txt
```

This last one is particularly handy when performing several operations on a group of files; we could run du and wc over them to get their size and character count, and then perhaps decide to delete them based on the output:

```
$ du a.txt b.txt c.txt
4164    a.txt
5184    b.txt
8356    c.txt
$ wc !*
wc a.txt b.txt c.txt
16689    94038  4250112 a.txt
20749   117100  5294592 b.txt
33190   188557  8539136 c.txt
70628   399695 18083840 total
$ rm !*
rm a.txt b.txt c.txt
```

These work not just for the preceding command in history, but also absolute and relative command numbers:

```
$ history 3
 3989  2012-08-16 16:30:59  wc -l b.txt
 3990  2012-08-16 16:31:05  du -sh c.txt
 3991  2012-08-16 16:31:12  history 3
$ echo !3989^
echo -l
-l
$ echo !3990$
echo c.txt
c.txt
$ echo !-1*
echo c.txt
c.txt
```

More generally, you can use the syntax !n:w to refer to any specific argument in a history item by number. In this case, the first word, usually a command or builtin, is word 0 :

```
$ history | grep bash
 4073  2012-08-16 20:24:53  man bash
$ !4073:0
man
What manual page do you want?
$ !4073:1
bash
```

You can even select ranges of words by separating their indices with a hyphen:

```
$ history | grep apt-get
 3663  2012-08-15 17:01:30  sudo apt-get install gnome
$ !3663:0-1 purge !3663:3
sudo apt-get purge gnome
```

You can include ^ and $ as start and endpoints for these ranges, too. 3* is a shorthand for 3-$, meaning "all arguments from the third to the last."

Expanding history by string

You can also refer to a previous command in the history that starts with a specific string with the syntax !string :

```
$ !echo
echo c.txt
c.txt
$ !history
history 3
 4011  2012-08-16 16:38:28  rm a.txt b.txt c.txt
 4012  2012-08-16 16:42:48  echo c.txt
 4013  2012-08-16 16:42:51  history 3
```

If you want to match any part of the command line, not just the start, you can use !?string? :

```
$ !?bash?
man bash
```

Be careful when using these, if you use them at all. By default it will run the most recent command matching the string ***immediately*** , with no prompting, so it might be a problem if it doesn't match the command you expect.

Checking history expansions before running

If you're paranoid about this, Bash allows you to audit the command as expanded before you enter it, with the histverify option:

```
$ shopt -s histverify
$ !rm
$ rm a.txt b.txt c.txt
```

This option works for any history expansion, and may be a good choice for more cautious administrators. It's a good thing to add to one's .bashrc if so.

If you don't need this set all the time, but you do have reservations at some point about running a history command, you can arrange to print the command without running it by adding a :p suffix:

```
$ !rm:p
rm important-file
```

In this instance, the command was expanded, but thankfully not actually run.

Substituting strings in history expansions

To get really in-depth, you can also perform substitutions on arbitrary commands from the history with !!:gs/pattern/replacement/ . This is getting pretty baroque even for Bash, but it's possible you may find it useful at some point:

```
$ !!:gs/txt/mp3/
rm a.mp3 b.mp3 c.mp3
```

If you only want to replace the first occurrence, you can omit the g :

```
$ !!:s/txt/mp3/
rm a.mp3 b.txt c.txt
```

Stripping leading directories or trailing files

If you want to chop a filename off a long argument to work with the directory, you can do this by adding an :h suffix, kind of like a dirname call in Perl:

```
$ du -sh /home/tom/work/doc.txt
$ cd !$:h
cd /home/tom/work
```

To do the opposite, like a basename call in Perl, use :t :

```
$ ls /home/tom/work/doc.txt
$ document=!$:t
document=doc.txt
```

Stripping extensions or base names

A bit more esoteric, but still possibly useful; to strip a file's extension, use :r :

```
$ vi /home/tom/work/doc.txt
$ stripext=!$:r
stripext=/home/tom/work/doc
```

To do the opposite, to get only the extension, use :e :

```
$ vi /home/tom/work/doc.txt
$ extonly=!$:e
extonly=.txt
```

Quoting history

If you're performing substitution not to execute a command or fragment but to use it as a string, it's likely you'll want to ***quote*** it. For example, if you've just found through experiment and trial and error an ideal ffmpeg command line to accomplish some task, you might want to save it for later use by writing it to a script:

```
$ ffmpeg -f alsa -ac 2 -i hw:0,0 -f x11grab -r 30 -s 1600x900 \
> -i :0.0+1600,0 -acodec pcm_s16le -vcodec libx264 -preset ultrafast \
> -crf 0 -threads 0 "$(date +%Y%m%d%H%M%S)".mkv
```

To make sure all the escaping is done correctly, you can write the command into the file with the :q modifier:

```
$ echo '#!/usr/bin/env bash' >ffmpeg.sh
$ echo !ffmpeg:q >>ffmpeg.sh
```

In this case, this will prevent Bash from executing the command expansion "$(date ...)" , instead writing it literally to the file as desired. If you build a lot of complex commands interactively that you later write to scripts once completed, this feature is really helpful and saves a lot of cutting and pasting.

Thanks to commenter Mihai Maruseac for pointing out a bug in the examples.

[Oct 31, 2017] Prompt directory shortening by Tom Ryder

Notable quotes:

"... If you're using Bash version 4.0 or above (bash --version), you can save a bit of terminal space by setting the PROMPT_DIRTRIM variable for the shell. This limits the length of the tail end of the \w and \W expansions to that number of path elements: ..."

Nov 07, 2014 | sanctum.geek.nz

The common default of some variant of \h:\w\$ for a Bash prompt PS1 string includes the \w escape character, so that the user's current working directory appears in the prompt, but with $HOME shortened to a tilde:

```
tom@sanctum:~$
tom@sanctum:~/Documents$
tom@sanctum:/usr/local/nagios$
```

This is normally very helpful, particularly if you leave your shell for a time and forget where you are, though of course you can always call the pwd shell builtin. However it can get annoying for very deep directory hierarchies, particularly if you're using a smaller terminal window:

```
tom@sanctum:/chroot/apache/usr/local/perl/app-library/lib/App/Library/Class:~$
```

If you're using Bash version 4.0 or above (bash --version), you can save a bit of terminal space by setting the PROMPT_DIRTRIM variable for the shell. This limits the length of the tail end of the \w and \W expansions to that number of path elements:

```
tom@sanctum:/chroot/apache/usr/local/app-library/lib/App/Library/Class$ PROMPT_DIRTRIM=3
tom@sanctum:.../App/Library/Class$
```

This is a good thing to include in your ~/.bashrc file if you often find yourself deep in directory trees where the upper end of the hierarchy isn't of immediate interest to you. You can remove the effect again by unsetting the variable:

```
tom@sanctum:.../App/Library/Class$ unset PROMPT_DIRTRIM
tom@sanctum:/chroot/apache/usr/local/app-library/lib/App/Library/Class$
```

[Oct 20, 2017] Simple logical operators in Bash - Stack Overflow

Notable quotes:

"... Backquotes (` `) are old-style form of command substitution, with some differences: in this form, backslash retains its literal meaning except when followed by $, ` , or \ , and the first backquote not preceded by a backslash terminates the command substitution; whereas in the $( ) form, all characters between the parentheses make up the command, none are treated specially. ..."

"... Double square brackets delimit a Conditional Expression. And, I find the following to be a good reading on the subject: "(IBM) Demystify test, [, [[, ((, and if-then-else" ..."

Oct 20, 2017 | stackoverflow.com

Amit , Jun 7, 2011 at 19:18

I have a couple of variables and I want to check the following condition (written out in words, then my failed attempt at bash scripting):

```
if varA EQUALS 1 AND ( varB EQUALS "t1" OR varB EQUALS "t2" ) then

do something

done.
```

And in my failed attempt, I came up with:

```
if (($varA == 1)) && ( (($varB == "t1")) || (($varC == "t2")) );
  then
    scale=0.05
  fi
```

Best answer Gilles

What you've written actually almost works (it would work if all the variables were numbers), but it's not an idiomatic way at all.

- ( ) parentheses indicate a subshell . What's inside them isn't an expression like in many other languages. It's a list of commands (just like outside parentheses). These commands are executed in a separate subprocess, so any redirection, assignment, etc. performed inside the parentheses has no effect outside the parentheses.
  - With a leading dollar sign, $( ) is a command substitution : there is a command inside the parentheses, and the output from the command is used as part of the command line (after extra expansions unless the substitution is between double quotes, but that's another story).
- { } braces are like parentheses in that they group commands, but they only influence parsing, not grouping. The program x=2; { x=4; }; echo $x prints 4, whereas x=2; (x=4); echo $x prints 2. (Also braces require spaces around them and a semicolon before closing, whereas parentheses don't. That's just a syntax quirk.)
  - With a leading dollar sign, ${VAR} is a parameter expansion , expanding to the value of a variable, with possible extra transformations.
- (( )) double parentheses surround an arithmetic instruction , that is, a computation on integers, with a syntax resembling other programming languages. This syntax is mostly used for assignments and in conditionals.
  - The same syntax is used in arithmetic expressions $(( )) , which expand to the integer value of the expression.
- [[ ]] double brackets surround conditional expressions . Conditional expressions are mostly built on operators such as -n $variable to test if a variable is empty and -e $file to test if a file exists. There are also string equality operators: "$string1" = "$string2" (beware that the right-hand side is a pattern, e.g. [[$foo = a*]] tests if $foo starts with a while [[$foo = "a*"]] tests if $foo is exactly a*), and the familiar ! , && and || operators for negation, conjunction and disjunction as well as parentheses for grouping.
  - Note that you need a space around each operator (e.g. [["$x" = "$y"]] , not ~~[["$x"="$y"]]~~), and a space or a character like ; both inside and outside the brackets (e.g. [[-n $foo]] , not ~~[[-n $foo]]~~).
- [ ] single brackets are an alternate form of conditional expressions with more quirks (but older and more portable). Don't write any for now; start worrying about them when you find scripts that contain them.

This is the idiomatic way to write your test in bash:

```
if [[ $varA = 1 && ($varB = "t1" || $varC = "t2") ]]; then
```

If you need portability to other shells, this would be the way (note the additional quoting and the separate sets of brackets around each individual test):

```
if [ "$varA" = 1 ] && { [ "$varB" = "t1" ] || [ "$varC" = "t2" ]; }; then
```

Will Sheppard , Jun 19, 2014 at 11:07

It's better to use == to differentiate the comparison from assigning a variable (which is also =) – Will Sheppard Jun 19 '14 at 11:07

Cbhihe , Apr 3, 2016 at 8:05

+1 @WillSheppard for yr reminder of proper style. Gilles, don't you need a semicolon after yr closing curly bracket and before "then" ? I always thought if , then , else and fi could not be on the same line... As in:

```
if [ "$varA" = 1 ] && { [ "$varB" = "t1" ] || [ "$varC" = "t2" ]; }; then
```

– Cbhihe Apr 3 '16 at 8:05

Rockallite , Jan 19 at 2:41

Backquotes (` `) are old-style form of command substitution, with some differences: in this form, backslash retains its literal meaning except when followed by $, ` , or \ , and the first backquote not preceded by a backslash terminates the command substitution; whereas in the $( ) form, all characters between the parentheses make up the command, none are treated specially.

– Rockallite Jan 19 at 2:41

Peter A. Schneider , Aug 28 at 13:16

You could emphasize that single brackets have completely different semantics inside and outside of double brackets. (Because you start with explicitly pointing out the subshell semantics but then only as an aside mention the grouping semantics as part of conditional expressions. Was confusing to me for a second when I looked at your idiomatic example.) – Peter A. Schneider Aug 28 at 13:16

matchew , Jun 7, 2011 at 19:29

very close

```
if (( $varA == 1 )) && [[ $varB == 't1' || $varC == 't2' ]];
  then
    scale=0.05
  fi
```

should work.

breaking it down

```
(( $varA == 1 ))
```

is an integer comparison where as

```
$varB == 't1'
```

is a string comparison. otherwise, I am just grouping the comparisons correctly.

Double square brackets delimit a Conditional Expression. And, I find the following to be a good reading on the subject: "(IBM) Demystify test, [, [[, ((, and if-then-else"

Peter A. Schneider , Aug 28 at 13:21

Just to be sure: The quoting in 't1' is unnecessary, right? Because as opposed to arithmetic instructions in double parentheses, where t1 would be a variable, t1 in a conditional expression in double brackets is just a literal string.

I.e., [[$varB == 't1']] is exactly the same as [[$varB == t1]] , right? – Peter A. Schneider Aug 28 at 13:21

[Oct 19, 2017] Bash One-Liners bashoneliners.com

Oct 19, 2017 | www.bashoneliners.com

Kill a process running on port 8080

```
$ lsof -i :8080 | awk 'NR > 1 {print $2}' | xargs --no-run-if-empty kill
```

-- by Janos on Sept. 1, 2017, 8:31 p.m.

Make a new folder and cd into it.

```
$ mkcd(){ NAME=$1; mkdir -p "$NAME"; cd "$NAME"; }
```

-- by PrasannaNatarajan on Aug. 3, 2017, 6:49 a.m.

Go up to a particular folder

```
$ alias ph='cd ${PWD%/public_html*}/public_html'
```

-- by Jab2870 on July 18, 2017, 6:07 p.m.

Explanation

I work on a lot of websites and often need to go up to the public_html folder.

This command creates an alias so that however many folders deep I am, I will be taken up to the correct folder.

alias ph='....' : This creates a ***shortcut*** so that when command ph is typed, the part between the quotes is executed

cd ... : This changes directory to the directory specified

PWD : This is a global bash variable that contains the current directory

${...%/public_html*} : This removes /public_html and anything after it from the specified string

Finally, /public_html at the end is appended onto the string.

So, to sum up, when ph is run, we ask bash to change the directory to the current working directory with anything after public_html removed.

Open another terminal at current location

```
$ $TERMINAL & disown
```

-- by Jab2870 on July 18, 2017, 3:04 p.m.

Explanation

Opens another terminal window at the current location.

Use Case

I often cd into a directory and decide it would be useful to open another terminal in the same folder, maybe for an editor or something. Previously, I would open the terminal and repeat the CD command.

I have aliased this command to open so I just type open and I get a new terminal already in my desired folder.

The & disown part of the command stops the new terminal from being dependant on the first meaning that you can still use the first and if you close the first, the second will remain open. **Limitations**

It relied on you having the $TERMINAL global variable set. If you don't have this set you could easily change it to something like the following:

gnome-terminal & disown or konsole & disown

Preserve your fingers from cd ..; cd ..; cd..; cd..;

```
$ up(){ DEEP=$1; for i in $(seq 1 ${DEEP:-"1"}); do cd ../; done; }
```

-- by alireza6677 on June 28, 2017, 5:40 p.m.

Generate a sequence of numbers

```
$ echo {01..10}
```

-- by Elkku on March 1, 2015, 12:04 a.m.

Explanation

This example will print:

```
01 02 03 04 05 06 07 08 09 10
```

While the original one-liner is indeed IMHO the canonical way to loop over numbers, the brace expansion syntax of Bash 4.x has some kick-ass features such as correct padding of the number with leading zeros. **Limitations**

The zero-padding feature works only in Bash >=4.

Tweet

Related one-liners

Generate a sequence of numbers

```
$ for ((i=1; i<=10; ++i)); do echo $i; done
```

-- by Janos on Nov. 4, 2014, 12:29 p.m.

Explanation

This is similar to seq , but portable. seq does not exist in all systems and is not recommended today anymore. Other variations to emulate various uses with seq :

```
# seq 1 2 10
for ((i=1; i<=10; i+=2)); do echo $i; done

# seq -w 5 10
for ((i=5; i<=10; ++i)); do printf '%02d\n' $i; done
```

Find recent logs that contain the string "Exception"

```
$ find . -name '*.log' -mtime -2 -exec grep -Hc Exception {} \; | grep -v :0$
```

-- by Janos on July 19, 2014, 7:53 a.m.

Explanation

The find :

- -name '*.log' -- match files ending with .log
- -mtime -2 -- match files modified within the last 2 days
- -exec CMD ARGS \; -- for each file found, execute command, where {} in ARGS will be replaced with the file's path

The grep :

- -c is to print the count of the matches instead of the matches themselves
- -H is to print the name of the file, as grep normally won't print it when there is only one filename argument
- The output lines will be in the format path:count . Files that didn't match "Exception" will still be printed, with 0 as count
- The second grep filters the output of the first, excluding lines that end with :0 (= the files that didn't contain matches)

Extra tips:

- Change "Exception" to the typical relevant failure indicator of your application
- Add -i for grep to make the search case insensitive
- To make the find match strictly only files, add -type f
- Schedule this as a periodic job, and pipe the output to a mailer, for example | mailx -s 'error counts' yourmail@example.com

Remove offending key from known_hosts file with one swift move

```
$ sed -i 18d .ssh/known_hosts
```

-- by EvaggelosBalaskas on Jan. 16, 2013, 2:29 p.m.

Explanation

Using sed to remove a specific line.

The -i parameter is to edit the file in-place. **Limitations**

This works as posted in GNU sed . In BSD sed , the -i flag requires a parameter to use as the suffix of a backup file. You can set it to empty to not use a backup file:

[Oct 16, 2017] Indenting Here-Documents - bash Cookbook

Oct 16, 2017 | www.safaribooksonline.com

Indenting Here-Documents Problem

The here-document is great, but it's messing up your shell script's formatting. You want to be able to indent for readability. **Solution**

Use <<- and then you can use tab characters (only!) at the beginning of lines to indent this portion of your shell script.

```
$ cat myscript.sh
...
    grep $1 <<-'EOF'
      lots of data
      can go here
      it's indented with tabs
      to match the script's indenting
      but the leading tabs are
      discarded when read
      EOF
    ls
...
$
```

Discussion

The hyphen just after the << is enough to tell ***bash*** to ignore the leading tab characters. This is for ***tab*** characters only and not arbitrary white space. This is especially important with the EOF or any other marker designation. If you have spaces there, it will not recognize the EOF as your ending marker, and the "here" data will continue through to the end of the file (swallowing the rest of your script). Therefore, you may want to always left-justify the EOF (or other marker) just to be safe, and let the formatting go on this one line.

[Oct 16, 2017] Indenting bourne shell here documents

Oct 16, 2017 | prefetch.net

The Bourne shell provides here documents to allow block of data to be passed to a process through STDIN. The typical format for a here document is something similar to this:

command <<ARBITRARY_TAG
data to pass 1
data to pass 2
ARBITRARY_TAG

This will send the data between the ARBITRARY_TAG statements to the standard input of the process. In order for this to work, you need to make sure that the data is not indented. If you indent it for readability, you will get a syntax error similar to the following:

./test: line 12: syntax error: unexpected end of file

To allow your here documents to be indented, you can append a "-" to the end of the redirection strings like so:

```
if [ "${STRING}" = "SOMETHING" ]
then
    somecommand <<-EOF
    this is a string1
    this is a string2
    this is a string3
    EOF
fi
```

You will need to use tabs to indent the data, but that is a small price to pay for added readability. Nice!

[Oct 09, 2017] TMOUT - Auto Logout Linux Shell When There Isn't Any Activity by Aaron Kili

Oct 07, 2017 | www.tecmint.com

... .

To enable automatic user logout, we will be using the TMOUT shell variable, which terminates a user's login shell in case there is no activity for a given number of seconds that you can specify.

To enable this globally (system-wide for all users), set the above variable in the /etc/profile shell initialization file.

[Sep 27, 2017] Arithmetic Evaluation

Sep 27, 2017 | mywiki.wooledge.org

Bash has several different ways to say we want to do arithmetic instead of string operations. Let's look at them one by one.

The first way is the let command:

```
$ unset a; a=4+5
$ echo $a
4+5
$ let a=4+5
$ echo $a
9
```

You may use spaces, parentheses and so forth, if you quote the expression:

```
$ let a='(5+2)*3'
```

For a full list of operators availabile, see help let or the manual.

Next, the actual ***arithmetic evaluation*** compound command syntax:

```
$ ((a=(5+2)*3))
```

This is equivalent to let , but we can also use it as a ***command*** , for example in an if statement:

```
$ if (($a == 21)); then echo 'Blackjack!'; fi
```

Operators such as == , < , > and so on cause a comparison to be performed, inside an arithmetic evaluation. If the comparison is "true" (for example, 10 > 2 is true in arithmetic -- but not in strings!) then the compound command exits with status 0. If the comparison is false, it exits with status 1. This makes it suitable for testing things in a script.

Although not a compound command, an ***arithmetic substitution*** (or ***arithmetic expression***) syntax is also available:

```
$ echo "There are $(($rows * $columns)) cells"
```

Inside $((...)) is an ***arithmetic context*** , just like with ((...)) , meaning we do arithmetic (multiplying things) instead of string manipulations (concatenating $rows , space, asterisk, space, $columns). $((...)) is also portable to the POSIX shell, while ((...)) is not.

Readers who are familiar with the C programming language might wish to know that ((...)) has many C-like features. Among them are the ternary operator:

```
$ ((abs = (a >= 0) ? a : -a))
```

and the use of an integer value as a truth value:

```
$ if ((flag)); then echo "uh oh, our flag is up"; fi
```

Note that we used variables inside ((...)) without prefixing them with $ -signs. This is a special syntactic shortcut that Bash allows inside arithmetic evaluations and arithmetic expressions.

There is one final thing we must mention about ((flag)) . Because the inside of ((...)) is C-like, a variable (or expression) that evaluates to ***zero*** will be considered ***false*** for the purposes of the arithmetic evaluation. Then, because the evaluation is false, it will ***exit*** with a status of 1. Likewise, if the expression inside ((...)) is ***non-zero*** , it will be considered ***true*** ; and since the evaluation is true, it will ***exit*** with status 0. This is potentially ***very*** confusing, even to experts, so you should take some time to think about this. Nevertheless, when things are used the way they're intended, it makes sense in the end:

```
$ flag=0     # no error
$ while read line; do
>   if [[ $line = *err* ]]; then flag=1; fi
> done < inputfile
$ if ((flag)); then echo "oh no"; fi
```

[Sep 27, 2017] Integer ASCII value to character in BASH using printf

Sep 27, 2017 | stackoverflow.com

user14070 , asked May 20 '09 at 21:07

Character to value works:

```
$ printf "%d\n" \'A
65
$
```

I have two questions, the first one is most important:

- How do I take 65 and turn it into A?
- \'A converts an ASCII character to its value using printf. Is the syntax ***specific*** to ***printf*** or is it used anywhere else in BASH? (Such small strings are hard to Google for.)

broaden , answered Nov 18 '09 at 10:10

One line

```
printf "\x$(printf %x 65)"
```

Two lines

```
set $(printf %x 65)
printf "\x$1"
```

Here is one if you do not mind using awk

```
awk 'BEGIN{printf "%c", 65}'
```

mouviciel , answered May 20 '09 at 21:12

For this kind of conversion, I use perl:

```
perl -e 'printf "%c\n", 65;'
```

user2350426 , answered Sep 22 '15 at 23:16

This works (with the value in octal):

```
$ printf '%b' '\101'
A
```

even for (some: don't go over 7) sequences:

```
$ printf '%b' '\'{101..107}
ABCDEFG
```

A general construct that allows (decimal) values in any range is:

```
$ printf '%b' $(printf '\\%03o' {65..122})
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
```

Or you could use the hex values of the characters:

```
$ printf '%b' $(printf '\\x%x' {65..122})
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
```

You also could get the character back with xxd (use hexadecimal values):

```
$ echo "41" | xxd -p -r
A
```

That is, one action is the reverse of the other:

```
$ printf "%x" "'A" | xxd -p -r
A
```

And also works with several hex values at once:

```
$ echo "41 42 43 44 45 46 47 48 49 4a" | xxd -p -r
ABCDEFGHIJ
```

or sequences (printf is used here to get hex values):

```
$ printf '%x' {65..90} | xxd -r -p
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Or even use awk:

```
$ echo 65 | awk '{printf("%c",$1)}'
A
```

even for sequences:

```
$ seq 65 90 | awk '{printf("%c",$1)}'
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

David Hu , answered Dec 1 '11 at 9:43

For your second question, it seems the leading-quote syntax (\'A) is specific to printf :

> If the leading character is a single-quote or double-quote, the value shall be the numeric value in the underlying codeset of the character following the single-quote or double-quote.

From http://pubs.opengroup.org/onlinepubs/009695399/utilities/printf.html

Naaff , answered May 20 '09 at 21:21

One option is to directly input the character you're interested in using hex or octal notation:

```
printf "\x41\n"
printf "\101\n"
```

MagicMercury86 , answered Feb 21 '12 at 22:49

If you want to save the ASCII value of the character: (I did this in BASH and it worked)

```
{
char="A"

testing=$( printf "%d" "'${char}" )

echo $testing}
```

output: 65

chand , answered Nov 20 '14 at 10:05

Here's yet another way to convert 65 into A (via octal):

```
help printf  # in Bash
man bash | less -Ip '^[[:blank:]]*printf'

printf "%d\n" "'A'
printf "%d\n" "'A"

printf '%b\n' "$(printf '\%03o' 65)"
```

To search in man bash for \' use (though futile in this case):

```
man bash | less -Ip "\\\'"  # press <n> to go through the matches
```

,

If you convert 65 to hexadecimal it's 0x41 :

```
$ echo -e "\x41" A
```

[Sep 27, 2017] linux - How to convert DOS-Windows newline (CRLF) to Unix newline in a Bash script

Notable quotes:

"... Technically '1' is your program, b/c awk requires one when given option. ..."

Koran Molovik , asked Apr 10 '10 at 15:03

How can I programmatically (i.e., not using vi) convert DOS/Windows newlines to Unix?

The dos2unix and unix2dos commands are not available on certain systems. How can I emulate these with commands like sed / awk / tr ?

Jonathan Leffler , answered Apr 10 '10 at 15:13

You can use tr to convert from DOS to Unix; however, you can only do this safely if CR appears in your file only as the first byte of a CRLF byte pair. This is usually the case. You then use:

```
tr -d '\015' <DOS-file >UNIX-file
```

Note that the name DOS-file is different from the name UNIX-file ; if you try to use the same name twice, you will end up with no data in the file.

You can't do it the other way round (with standard 'tr').

If you know how to enter carriage return into a script (control-V , control-M to enter control-M), then:

```
sed 's/^M$//'     # DOS to Unix
sed 's/$/^M/'     # Unix to DOS
```

where the '^M' is the control-M character. You can also use the bash ANSI-C Quoting mechanism to specify the carriage return:

```
sed $'s/\r$//'    # DOS to Unix
sed $'s/$/\r/'    # Unix to DOS
```

However, if you're going to have to do this very often (more than once, roughly speaking), it is far more sensible to install the conversion programs (e.g. dos2unix and unix2dos , or perhaps dtou and utod) and use them.

ghostdog74 , answered Apr 10 '10 at 15:21

```
tr -d "\r" < file
```

take a look here for examples using sed :

```
# IN UNIX ENVIRONMENT: convert DOS newlines (CR/LF) to Unix format.
sed 's/.$//'               # assumes that all lines end with CR/LF
sed 's/^M$//'              # in bash/tcsh, press Ctrl-V then Ctrl-M
sed 's/\x0D$//'            # works on ssed, gsed 3.02.80 or higher

# IN UNIX ENVIRONMENT: convert Unix newlines (LF) to DOS format.
sed "s/$/`echo -e \\\r`/"            # command line under ksh
sed 's/$'"/`echo \\\r`/"             # command line under bash
sed "s/$/`echo \\\r`/"               # command line under zsh
sed 's/$/\r/'                        # gsed 3.02.80 or higher
```

Use sed -i for in-place conversion e.g. sed -i 's/..../' file .

Steven Penny , answered Apr 30 '14 at 10:02

Doing this with POSIX is tricky:

- POSIX Sed does not support \r or \15 . Even if it did, the in place option -i is not POSIX
- POSIX Awk does support \r and \15 , however the -i inplace option is not POSIX
- ***d2u*** and ***dos2unix*** are not POSIX utilities , but ***ex*** is
- POSIX ex does not support \r , \15 , \n or \12

To remove carriage returns:

```
ex -bsc '%!awk "{sub(/\r/,\"\")}1"' -cx file
```

To add carriage returns:

```
ex -bsc '%!awk "{sub(/$/,\"\r\")}1"' -cx file
```

Norman Ramsey , answered Apr 10 '10 at 22:32

This problem can be solved with standard tools, but there are sufficiently many traps for the unwary that I recommend you install the flip command, which was written over 20 years ago by Rahul Dhesi, the author of zoo . It does an excellent job converting file formats while, for example, avoiding the inadvertant destruction of binary files, which is a little too easy if you just race around altering every CRLF you see...

Gordon Davisson , answered Apr 10 '10 at 17:50

The solutions posted so far only deal with part of the problem, converting DOS/Windows' CRLF into Unix's LF; the part they're missing is that DOS use CRLF as a line ***separator*** , while Unix uses LF as a line ***terminator*** . The difference is that a DOS file (usually) won't have anything after the last line in the file, while Unix will. To do the conversion properly, you need to add that final LF (unless the file is zero-length, i.e. has no lines in it at all). My favorite incantation for this (with a little added logic to handle Mac-style CR-separated files, and not molest files that're already in unix format) is a bit of perl:

```
perl -pe 'if ( s/\r\n?/\n/g ) { $f=1 }; if ( $f || ! $m ) { s/([^\n])\z/$1\n/ }; $m=1' PCfile.txt
```

Note that this sends the Unixified version of the file to stdout. If you want to replace the file with a Unixified version, add perl's -i flag.

codaddict , answered Apr 10 '10 at 15:09

Using AWK you can do:

```
awk '{ sub("\r$", ""); print }' dos.txt > unix.txt
```

Using Perl you can do:

```
perl -pe 's/\r$//' < dos.txt > unix.txt
```

anatoly techtonik , answered Oct 31 '13 at 9:40

If you don't have access to dos2unix , but can read this page, then you can copy/paste dos2unix.py from here.

```
#!/usr/bin/env python
"""\
convert dos linefeeds (crlf) to unix (lf)
usage: dos2unix.py <input> <output>
"""
import sys

if len(sys.argv[1:]) != 2:
  sys.exit(__doc__)

content = ''
outsize = 0
with open(sys.argv[1], 'rb') as infile:
  content = infile.read()
with open(sys.argv[2], 'wb') as output:
  for line in content.splitlines():
    outsize += len(line) + 1
    output.write(line + '\n')

print("Done. Saved %s bytes." % (len(content)-outsize))
```

Cross-posted from superuser .

nawK , answered Sep 4 '14 at 0:16

An even simpler awk solution w/o a program:

```
awk -v ORS='\r\n' '1' unix.txt > dos.txt
```

Technically '1' is your program, b/c awk requires one when given option.

UPDATE : After revisiting this page for the first time in a long time I realized that no one has yet posted an internal solution, so here is one:

```
while IFS= read -r line;
do printf '%s\n' "${line%$'\r'}";
done < dos.txt > unix.txt
```

Santosh , answered Mar 12 '15 at 22:36

This worked for me

```
tr "\r" "\n" < sampledata.csv > sampledata2.csv
```

ThorSummoner , answered Jul 30 '15 at 17:38

Super duper easy with PCRE;

As a script, or replace $@ with your files.

```
#!/usr/bin/env bash
perl -pi -e 's/\r\n/\n/g' -- $@
```

This will overwrite your files in place!

I recommend only doing this with a backup (version control or otherwise)

Ashley Raiteri , answered May 19 '14 at 23:25

For Mac osx if you have homebrew installed [http://brew.sh/][1]

```
brew install dos2unix

for csv in *.csv; do dos2unix -c mac ${csv}; done;
```

Make sure you have made copies of the files, as this command will modify the files in place. The -c mac option makes the switch to be compatible with osx.

lzc , answered May 31 '16 at 17:15

TIMTOWTDI!

```
perl -pe 's/\r\n/\n/; s/([^\n])\z/$1\n/ if eof' PCfile.txt
```

Based on @GordonDavisson

One must consider the possibility of [noeol] ...

kazmer , answered Nov 6 '16 at 23:30

You can use awk. Set the record separator (RS) to a regexp that matches all possible newline character, or characters. And set the output record separator (ORS) to the unix-style newline character.

```
awk 'BEGIN{RS="\r|\n|\r\n|\n\r";ORS="\n"}{print}' windows_or_macos.txt > unix.txt
```

user829755 , answered Jul 21 at 9:21

interestingly in my git-bash on windows sed "" did the trick already:

```
$ echo -e "abc\r" >tst.txt
$ file tst.txt
tst.txt: ASCII text, with CRLF line terminators
$ sed -i "" tst.txt
$ file tst.txt
tst.txt: ASCII text
```

My guess is that sed ignores them when reading lines from input and always writes unix line endings on output.

Gannet , answered Jan 24 at 8:38

As an extension to Jonathan Leffler's Unix to DOS solution, to safely convert to DOS when you're unsure of the file's current line endings:

```
sed '/^M$/! s/$/^M/'
```

This checks that the line does not already end in CRLF before converting to CRLF.

vmsnomad , answered Jun 23 at 18:37

Had just to ponder that same question (on Windows-side, but equally applicable to linux.) Surprisingly nobody mentioned a very much automated way of doing CRLF<->LF conversion for text-files using good old zip -ll option (Info-ZIP):

```
zip -ll textfiles-lf.zip files-with-crlf-eol.*
unzip textfiles-lf.zip
```

NOTE: this would create a zip file preserving the original file names but converting the line endings to LF. Then unzip would extract the files as zip'ed, that is with their original names (but with LF-endings), thus prompting to overwrite the local original files if any.

Relevant excerpt from the zip --help :

```
zip --help
...
-l   convert LF to CR LF (-ll CR LF to LF)
```

I tried sed 's/^M$//' file.txt on OSX as well as several other methods (http://www.thingy-ma-jig.co.uk/blog/25-11-2010/fixing-dos-line-endings or http://hintsforums.macworld.com/archive/index.php/t-125.html). None worked, the file remained unchanged (btw Ctrl-v Enter was needed to reproduce ^M). In the end I used TextWrangler. Its not strictly command line but it works and it doesn't complain.

[Aug 29, 2017] How to view the `.bash_history` file via command line

Aug 29, 2017 | askubuntu.com

If you actually need the output of the .bash_history **file** , replace history with

cat ~/.bash_history in all of the **commands** below.

If you actually want the **commands** without numbers in front, use this **command** instead of history :

```
history | cut -d' ' -f 4-
```

[Jul 29, 2017] Preserve bash history in multiple terminal windows - Unix Linux Stack Exchange

Jul 29, 2017 | unix.stackexchange.com

Oli , asked Aug 26 '10 at 13:04

I consistently have more than one terminal open. Anywhere from two to ten, doing various bits and bobs. Now let's say I restart and open up another set of terminals. Some remember certain things, some forget.

I want a history that:

- Remembers everything from every terminal
- Is instantly accessible from every terminal (eg if I ls in one, switch to another already-running terminal and then press up, ls shows up)
- Doesn't forget command if there are spaces at the front of the command.

Anything I can do to make bash work more like that?

Pablo R., answered Aug 26 '10 at 14:37

```
# Avoid duplicates
export HISTCONTROL=ignoredups:erasedups
# When the shell exits, append to the history file instead of overwriting it
shopt -s histappend

# After each command, append to the history file and reread it
export PROMPT_COMMAND="${PROMPT_COMMAND:+$PROMPT_COMMAND$'\n'}history -a; history -c; history -r"
```

kch, answered Sep 19 '08 at 17:49

So, this is all my history-related .bashrc thing:

```
export HISTCONTROL=ignoredups:erasedups  # no duplicate entries
export HISTSIZE=100000                   # big big history
export HISTFILESIZE=100000               # big big history
shopt -s histappend                      # append to history, don't overwrite it

# Save and reload the history after each command finishes
export PROMPT_COMMAND="history -a; history -c; history -r; $PROMPT_COMMAND"
```

Tested with bash 3.2.17 on Mac OS X 10.5, bash 4.1.7 on 10.6.

lesmana, answered Jun 16 '10 at 16:11

Here is my attempt at Bash session history sharing. This will enable history sharing between bash sessions in a way that the history counter does not get mixed up and history expansion like !number will work (with some constraints).

Using Bash version 4.1.5 under Ubuntu 10.04 LTS (Lucid Lynx).

```
HISTSIZE=9000
HISTFILESIZE=$HISTSIZE
HISTCONTROL=ignorespace:ignoredups

_bash_history_sync() {
  builtin history -a         #1
  HISTFILESIZE=$HISTSIZE     #2
  builtin history -c         #3
  builtin history -r         #4
}

history() {                  #5
  _bash_history_sync
  builtin history "$@"
}

PROMPT_COMMAND=_bash_history_sync
```

Explanation:

1. Append the just entered line to the $HISTFILE (default is .bash_history). This will cause $HISTFILE to grow by one line.
2. Setting the special variable $HISTFILESIZE to some value will cause Bash to truncate $HISTFILE to be no longer than $HISTFILESIZE lines by removing the oldest entries.
3. Clear the history of the running session. This will reduce the history counter by the amount of $HISTSIZE .
4. Read the contents of $HISTFILE and insert them in to the current running session history. this will raise the history counter by the amount of lines in $HISTFILE . Note that the line count of $HISTFILE is not necessarily $HISTFILESIZE .
5. The history() function overrides the builtin history to make sure that the history is synchronised before it is displayed. This is necessary for the history expansion by number (more about this later).

More explanation:

- Step 1 ensures that the command from the current running session gets written to the global history file.
- Step 4 ensures that the commands from the other sessions gets read in to the current session history.
- Because step 4 will raise the history counter, we need to reduce the counter in some way. This is done in step 3.
- In step 3 the history counter is reduced by $HISTSIZE . In step 4 the history counter is raised by the number of lines in $HISTFILE . In step 2 we make sure that the line count of $HISTFILE is exactly $HISTSIZE (this means that $HISTFILESIZE must be the same as $HISTSIZE).

About the constraints of the history expansion:

When using history expansion by number, you should always look up the number immediately before using it. That means no bash prompt display between looking up the number and using it. That usually means no enter and no ctrl+c.

Generally, once you have more than one Bash session, there is no guarantee whatsoever that a history expansion by number will retain its value between two Bash prompt displays. Because when PROMPT_COMMAND is executed the history from all other Bash sessions are integrated in the history of the current session. If any other bash session has a new command then the history numbers of the current session will be different.

I find this constraint reasonable. I have to look the number up every time anyway because I can't remember arbitrary history numbers.

Usually I use the history expansion by number like this

```
$ history | grep something #note number
$ !number
```

I recommend using the following Bash options.

```
## reedit a history substitution line if it failed
shopt -s histreedit
## edit a recalled history line before executing
shopt -s histverify
```

Strange bugs:

Running the history command piped to anything will result that command to be listed in the history twice. For example:

```
$ history | head
$ history | tail
$ history | grep foo
$ history | true
$ history | false
```

All will be listed in the history twice. I have no idea why.

Ideas for improvements:

- Modify the function _bash_history_sync() so it does not execute every time. For example it should not execute after a CTRL+C on the prompt. I often use CTRL+C to discard a long command line when I decide that I do not want to execute that line. Sometimes I have to use CTRL+C to stop a Bash completion script.
- Commands from the current session should always be the most recent in the history of the current session. This will also have the side effect that a given history number keeps its value for history entries from this session.

Maciej Piechotka, answered Aug 26 '10 at 13:20

I'm not aware of any way using bash . But it's one of the most popular features of zsh .
Personally I prefer zsh over bash so I recommend trying it.

Here's the part of my .zshrc that deals with history:

```
SAVEHIST=10000 # Number of entries
HISTSIZE=10000
HISTFILE=~/.zsh/history # File
setopt APPEND_HISTORY # Don't erase history
setopt EXTENDED_HISTORY # Add additional data to history like timestamp
setopt INC_APPEND_HISTORY # Add immediately
setopt HIST_FIND_NO_DUPS # Don't show duplicates in search
setopt HIST_IGNORE_SPACE # Don't preserve spaces. You may want to turn it off
setopt NO_HIST_BEEP # Don't beep
setopt SHARE_HISTORY # Share history between session/terminals
```

Chris Down, answered Nov 25 '11 at 15:46

To do this, you'll need to add two lines to your ~/.bashrc :

```
shopt -s histappend
PROMPT_COMMAND="history -a;history -c;history -r;"
$PROMPT_COMMAND
```

From man bash :

> If the histappend shell option is enabled (see the description of shopt under SHELL BUILTIN COMMANDS below), the lines are appended to the history file, otherwise the history file is over-written.

Schof, answered Sep 19 '08 at 19:38

You can edit your BASH prompt to run the "history -a" and "history -r" that Muerr suggested:

```
savePS1=$PS1
```

(in case you mess something up, which is almost guaranteed)

```
PS1=$savePS1`history -a;history -r`
```

(note that these are back-ticks; they'll run history -a and history -r on every prompt. Since they don't output any text, your prompt will be unchanged.

Once you've got your PS1 variable set up the way you want, set it permanently it in your ~/.bashrc file.

If you want to go back to your original prompt while testing, do:

```
PS1=$savePS1
```

I've done basic testing on this to ensure that it sort of works, but can't speak to any side-effects from running history -a;history -r on every prompt.

pts, answered Mar 25 '11 at 17:40

If you need a bash or zsh history synchronizing solution which also solves the problem below, then see it at http://ptspts.blogspot.com/2011/03/how-to-automatically-synchronize-shell.html

The problem is the following: I have two shell windows A and B. In shell window A, I run sleep 9999 , and (without waiting for the sleep to finish) in shell window B, I want to be able to see sleep 9999 in the bash history.

The reason why most other solutions here won't solve this problem is that they are writing their history changes to the the history file using PROMPT_COMMAND or PS1 , both of which are executing too late, only after the sleep 9999 command has finished.

jtimberman, answered Sep 19 '08 at 17:38

You can use history -a to append the current session's history to the histfile, then use history -r on the other terminals to read the histfile.

jmanning2k, answered Aug 26 '10 at 13:59

I can offer a fix for that last one: make sure the env variable HISTCONTROL does not specify "ignorespace" (or "ignoreboth").

But I feel your pain with multiple concurrent sessions. It simply isn't handled well in bash.

Toby, answered Nov 20 '14 at 14:53

Here's an alternative that I use. It's cumbersome but it addresses the issue that @axel_c mentioned where sometimes you may want to have a separate history instance in each terminal (one for make, one for monitoring, one for vim, etc).

I keep a separate appended history file that I constantly update. I have the following mapped to a hotkey:

```
history | grep -v history >> ~/master_history.txt
```

This appends all history from the current terminal to a file called master_history.txt in your home dir.

I also have a separate hotkey to search through the master history file:

```
cat /home/toby/master_history.txt | grep -i
```

I use cat | grep because it leaves the cursor at the end to enter my regex. A less ugly way to do this would be to add a couple of scripts to your path to accomplish these tasks, but hotkeys work for my purposes. I also periodically will pull history down from other hosts I've worked on and append that history to my master_history.txt file.

It's always nice to be able to quickly search and find that tricky regex you used or that weird perl one-liner you came up with 7 months ago.

Yarek T, answered Jul 23 '15 at 9:05

Right, So finally this annoyed me to find a decent solution:

```
# Write history after each command
_bash_history_append() {
    builtin history -a
}
PROMPT_COMMAND="_bash_history_append; $PROMPT_COMMAND"
```

What this does is sort of amalgamation of what was said in this thread, except that I don't understand why would you reload the global history after every command. I very rarely care about what happens in other terminals, but I always run series of commands, say in one terminal:

```
make
ls -lh target/*.foo
scp target/artifact.foo vm:~/
```

(Simplified example)

And in another:

```
pv ~/test.data | nc vm:5000 >> output
less output
mv output output.backup1
```

No way I'd want the command to be shared

rouble, answered Apr 15 at 17:43

Here is my enhancement to @lesmana's answer. The main difference is that concurrent windows don't share history. This means you can keep working in your windows, without having context from other windows getting loaded into your current windows.

If you explicitly type 'history', OR if you open a new window then you get the history from all previous windows.

Also, I use this strategy to archive every command ever typed on my machine.

```
# Consistent and forever bash history
HISTSIZE=100000
HISTFILESIZE=$HISTSIZE
HISTCONTROL=ignorespace:ignoredups

_bash_history_sync() {
  builtin history -a         #1
  HISTFILESIZE=$HISTSIZE     #2
}

_bash_history_sync_and_reload() {
  builtin history -a         #1
  HISTFILESIZE=$HISTSIZE     #2
  builtin history -c         #3
  builtin history -r         #4
}

history() {                  #5
  _bash_history_sync_and_reload
  builtin history "$@"
}

export HISTTIMEFORMAT="%y/%m/%d %H:%M:%S   "
PROMPT_COMMAND='history 1 >> ${HOME}/.bash_eternal_history'
PROMPT_COMMAND=_bash_history_sync;$PROMPT_COMMAND
```

simotek, answered Jun 1 '14 at 6:02

I have written a script for setting a history file per session or task its based off the following.

```
# write existing history to the old file
history -a

# set new historyfile
export HISTFILE="$1"
export HISET=$1

# touch the new file to make sure it exists
```

```
touch $HISTFILE
# load new history file
history -r $HISTFILE
```

It doesn't necessary save every history command but it saves the ones that i care about and its easier to retrieve them then going through every command. My version also lists all history files and provides the ability to search through them all.

Full source: https://github.com/simotek/scripts-config/blob/master/hiset.sh

Litch, answered Aug 11 '15 at 0:15

I chose to put history in a file-per-tty, as multiple people can be working on the same server - separating each session's commands makes it easier to audit.

```
# Convert /dev/nnn/X or /dev/nnnX to "nnnX"
HISTSUFFIX=`tty | sed 's/\///g;s/^dev//g'`
# History file is now .bash_history_pts0
HISTFILE=".bash_history_$HISTSUFFIX"
HISTTIMEFORMAT="%y-%m-%d %H:%M:%S "
HISTCONTROL=ignoredups:ignorespace
shopt -s histappend
HISTSIZE=1000
HISTFILESIZE=5000
```

History now looks like:

```
user@host:~# test 123
user@host:~# test 5451
user@host:~# history
1  15-08-11 10:09:58 test 123
2  15-08-11 10:10:00 test 5451
3  15-08-11 10:10:02 history
```

With the files looking like:

```
user@host:~# ls -la .bash*
-rw------- 1 root root  4275 Aug 11 09:42 .bash_history_pts0
-rw------- 1 root root    75 Aug 11 09:49 .bash_history_pts1
-rw-r--r-- 1 root root  3120 Aug 11 10:09 .bashrc
```

fstang, answered Sep 10 '16 at 19:30

Here I will point out one problem with

```
export PROMPT_COMMAND="${PROMPT_COMMAND:+$PROMPT_COMMAND$'\n'}history -a; history -c; history -r"
```

and

```
PROMPT_COMMAND="$PROMPT_COMMAND;history -a; history -n"
```

If you run source ~/.bashrc, the $PROMPT_COMMAND will be like

```
"history -a; history -c; history -r history -a; history -c; history -r"
```

and

```
"history -a; history -n history -a; history -n"
```

This repetition occurs each time you run 'source ~/.bashrc'. You can check PROMPT_COMMAND after each time you run 'source ~/.bashrc' by running 'echo $PROMPT_COMMAND'.

You could see some commands are apparently broken: "history -n history -a". But the good news is that it still works, because other parts still form a valid command sequence (Just involving some extra cost due to executing some commands repetitively. And not so clean.)

Personally I use the following simple version:

```
shopt -s histappend
PROMPT_COMMAND="history -a; history -c; history -r"
```

which has most of the functionalities while no such issue as mentioned above.

Another point to make is: there is really nothing magic . PROMPT_COMMAND is just a plain bash environment variable. The commands in it get executed before you get bash prompt (the $ sign). For example, your PROMPT_COMMAND is "echo 123", and you run "ls" in your terminal. The effect is like running "ls; echo 123".

```
$ PROMPT_COMMAND="echo 123"
```

output (Just like running 'PROMPT_COMMAND="echo 123"; $PROMPT_COMMAND'):

```
123
```

Run the following:

```
$ echo 3
```

output:

```
3
123
```

"history -a" is used to write the history commands in memory to ~/.bash_history

"history -c" is used to clear the history commands in memory

"history -r" is used to read history commands from ~/.bash_history to memory

See history command explanation here: http://ss64.com/bash/history.html

PS: As other users have pointed out, export is unnecessary. See: using export in .bashrc

Hopping Bunny, answered May 13 '15 at 4:48

Here is the snippet from my .bashrc and short explanations wherever needed:

```
# The following line ensures that history logs screen commands as well
shopt -s histappend

# This line makes the history file to be rewritten and reread at each bash prompt
PROMPT_COMMAND="$PROMPT_COMMAND;history -a; history -n"
# Have lots of history
HISTSIZE=100000         # remember the last 100000 commands
HISTFILESIZE=100000     # start truncating commands after 100000 lines
HISTCONTROL=ignoreboth  # ignoreboth is shorthand for ignorespace and     ignoredups
```

The HISTFILESIZE and HISTSIZE are personal preferences and you can change them as per your tastes.

Mulki, answered Jul 24 at 20:49

This works for ZSH

```
############################################################################
# History Configuration for ZSH
############################################################################
HISTSIZE=10000              #How many lines of history to keep in memory
HISTFILE=~/.zsh_history     #Where to save history to disk
SAVEHIST=10000              #Number of history entries to save to disk
#HISTDUP=erase              #Erase duplicates in the history file
setopt    appendhistory     #Append history to the history file (no overwriting)
setopt    sharehistory      #Share history across terminals
setopt    incappendhistory  #Immediately append to the history file, not just when a term is killed
```

[Jul 29, 2017] shell - How does this bash code detect an interactive session - Stack Overflow

Notable quotes:

"... ', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If parameter is an array variable subscripted with '@' or ' ..."

Jul 29, 2017 | stackoverflow.com

user1284631, asked Jun 5 '13 at 8:44

Following some issues with scp (it did not like the presence of the bash bind command in my .bashrc file, apparently), I followed the advice of a clever guy on the Internet (I just cannot find that post right now) that put at the top of its .bashrc file this:

```
[[ ${-#*} != ${-} ]] || return
```

in order to make sure that the bash initialization is NOT executed unless in interactive session.

Now, that works. However, I am not able to figure how it works. Could you enlighten me?

According to this answer, the $- is the current options set for the shell and I know that the ${} is the so-called "substring" syntax for expanding variables.

However, I do not understand the ${-#*i} part. And why $-#*i is not the same as ${-#*i} .

blue, answered Jun 5 '13 at 8:49

> parameter#word}
>
> $parameter##word}
>
> The word is expanded to produce a pattern just as in filename expansion. If the pattern matches the beginning of the expanded value of parameter, then the result of the expansion is the expanded value of parameter with the shortest matching pattern (the '#' case) or the longest matching pattern (the '##' case) deleted.
>
> If parameter is '@' or ' ***', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If parameter is an array variable subscripted with '@' or '*** ', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

Source: http://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html

So basically what happens in ${-#*i} is that *i is expanded, and if it matches the beginning of the value of $- , then the result of the whole expansion is $- with the shortest matching pattern between *i and $- deleted.

Example

```
VAR "baioasd"
echo ${VAR#*i};
```

outputs oasd .

In your case

If shell is interactive, $- will contain the letter 'i', so when you strip the variable $- of the pattern *i you will get a string that is different from the original $- ([[${-#*i} != ${-}]] yelds true). If shell is not interactive, $- does not contain the letter 'i' so the pattern *i does not match anything in $- and [[${-#*i} != $-]] yelds false, and the return statement is executed.

perreal, answered Jun 5 '13 at 8:53

See this :

> To determine within a startup script whether or not Bash is running interactively, test the value of the '-' special parameter. It contains i when the shell is interactive

Your substitution removes the string up to, and including the i and tests if the substituted version is equal to the original string. They will be different if there is i in the ${-} .

[Jul 26, 2017] I feel stupid declare not found in bash scripting

A single space can make a huge difference in bash :-)

www.linuxquestions.org

Mohtek

I feel stupid: declare not found in bash scripting? I was anxious to get my feet wet, and I'm only up to my toes before I'm stuck...this seems very very easy but I'm not sure what I've done wrong. Below is the script and its output. What the heck am I missing?

```
_____________________________________________________
#!/bin/bash
declare -a PROD[0]="computers" PROD[1]="HomeAutomation"
printf "${ PROD[*]}"
_____________________________________________________
```

products.sh: 6: declare: not found
products.sh: 8: Syntax error: Bad substitution

wjevans_7d1@yahoo.co

I ran what you posted (but at the command line, not in a script, though that should make no significant difference), and got this:

Code:

-bash: ${ PROD[*]}: bad substitution

In other words, I couldn't reproduce your first problem, the "declare: not found" error. Try the declare command by itself, on the command line.

And I got rid of the "bad substitution" problem when I removed the space which is between the ${ and the PROD on the printf line.

Hope this helps.

blackhole54

The previous poster identified your second problem.

As far as your first problem goes ... I am not a bash guru although I have written a number of bash scripts. So far I have found no need for declare statements. I suspect that you might not need it either. But if you do want to use it, the following does work:

Code:

```
#!/bin/bash

declare -a PROD
PROD[0]="computers"
PROD[1]="HomeAutomation"
printf "${PROD[*]}\n"
```

EDIT: My original post was based on an older version of bash. When I tried the declare statement you posted I got an error message, but one that was different from yours. I just tried it on a newer version of bash, and your declare statement worked fine. So it might depend on the version of bash you are running. What I posted above runs fine on both versions.

[Jul 26, 2017] Associative array declaration gotcha

Jul 26, 2017 | unix.stackexchange.com

bash silently does function return on (re-)declare of global associative read-only array - Unix & Linux Stack Exchange

Ron Burk :

Obviously cut out of a much more complex script that was more meaningful:

```
#!/bin/bash

function InitializeConfig(){
   declare -r -g -A SHCFG_INIT=( [a]=b )
   declare -r -g -A SHCFG_INIT=( [c]=d )
   echo "This statement never gets executed"
}

set -o xtrace

InitializeConfig
echo "Back from function"
```

The output looks like this:

```
ronburk@ubuntu:~/ubucfg$ bash bug.sh
+ InitializeConfig
+ SHCFG_INIT=([a]=b)
+ declare -r -g -A SHCFG_INIT
+ SHCFG_INIT=([c]=d)
```

```
+ echo 'Back from function'
Back from function
```

Bash seems to silently execute a function return upon the second declare statement. Starting to think this really is a new bug, but happy to learn otherwise.

Other details:

```
Machine: x86_64
OS: linux-gnu
Compiler: gcc
Compilation CFLAGS:  -DPROGRAM='bash' -DCONF_HOSTTYPE='x86_64' -DCONF_OSTYPE='linux-gnu' -DCONF_MACHTYPE='x86_64-
uname output: Linux ubuntu 3.16.0-38-generic #52~14.04.1-Ubuntu SMP Fri May 8 09:43:57 UTC 2015 x86_64 x86_64 x86_64 GNU/Lin$
Machine Type: x86_64-pc-linux-gnu

Bash Version: 4.3
Patch Level: 11
Release Status: release
```

bash array readonly

share improve this question edited Jun 14 '15 at 17:43 asked Jun 14 '15 at 7:05 118

By gum, you're right! Then I get readonly warning on second declare, which is reasonable, and the function completes. The xtrace output is also interesting; implies declare without single quotes is really treated as two steps. Ready to become superstitious about always single-quoting the argument to declare . Hard to see how popping the function stack can be anything but a bug, though. – Ron Burk Jun 14 '15 at 23:58

Weird. Doesn't happen in bash 4.2.53(1). – choroba Jun 14 '15 at 7:22

I can reproduce this problem with bash version 4.3.11 (Ubuntu 14.04.1 LTS). It works fine with bash 4.2.8 (Ubuntu 11.04). – Cyrus Jun 14 '15 at 7:34

Maybe related: unix.stackexchange.com/q/56815/116972 I can get expected result with declare -r -g -A 'SHCFG_INIT=([a]=b)' . – yaegashi Jun 14 '15 at 23:22

add a comment |

I found this thread in bug-bash@gnu.org related to test -v on an assoc array. In short, bash implicitly did test -v SHCFG_INIT[0] in your script. I'm not sure this behavior got introduced in 4.3.

You might want to use declare -p to workaround this...

```
if  declare p SHCFG_INIT >/dev/null >& ; then
    echo "looks like SHCFG_INIT not defined"
fi
====
```

Well, rats. I think your answer is correct, but also reveals I'm really asking two separate questions when I thought they were probably the same issue. Since the title better reflects what turns out to be the "other" question, I'll leave this up for a while and see if anybody knows what's up with the mysterious implicit function return... Thanks! – Ron Burk Jun 14 '15 at 17:01

Edited question to focus on the remaining issue. Thanks again for the answer on the "-v" issue with associative arrays. – Ron Burk Jun 14 '15 at 17:55

Accepting this answer. Complete answer is here plus your comments above plus (IMHO) there's a bug in this version of bash (can't see how there can be any excuse for popping the function stack without warning). Thanks for your excellent research on this! – Ron Burk Jun 21 '15 at 19:31

[Jul 26, 2017] Typing variables: declare or typeset

Jul 26, 2017 | www.tldp.org

The *declare* or *typeset* builtins , which are exact synonyms, permit modifying the properties of variables. This is a very weak form of the *typing* [1] available in certain programming languages. The *declare* command is specific to version 2 or later of Bash. The *typeset* command also works in ksh scripts.

declare/typeset options

-r *readonly*

(**declare -r var1** works the same as **readonly var1**)

This is the rough equivalent of the **C** *const* type qualifier. An attempt to change the value of a *readonly* variable fails with an error message.

```
declare -r var1=1
echo "var1 = $var1"   # var1 = 1

(( var1++ ))        # x.sh: line 4: var1: readonly variable
```

-i *integer*

```
declare -i number
# The script will treat subsequent occurrences of "number" as an integer.

number=3
echo "Number = $number"     # Number = 3

number=three
echo "Number = $number"     # Number = 0
# Tries to evaluate the string "three" as an integer.
```

Certain arithmetic operations are permitted for declared integer variables without the need for expr or let .

```
n=6/3
echo "n = $n"      # n = 6/3

declare -i n
n=6/3
echo "n = $n"      # n = 2
```

-a *array*

```
declare -a indices
```

The variable *indices* will be treated as an array .

-f *function(s)*

```
declare -f
```

A **declare -f** line with no arguments in a script causes a listing of all the functions previously defined in that script.

```
declare -f function_name
```

A **declare -f function_name** in a script lists just the function named.

-x export

```
declare -x var3
```

This declares a variable as available for exporting outside the environment of the script itself.

-x var=$value

```
declare -x var3=373
```

The **declare** command permits assigning a value to a variable in the same statement as setting its properties.

Example 9-10. Using *declare* to type variables

```
#!/bin/bash

func1 ()
{
 echo This is a function.
}

declare -f       # Lists the function above.

echo

declare -i var1   # var1 is an integer.
var1=2367
echo "var1 declared as $var1"
var1=var1+1      # Integer declaration eliminates the need for 'let'.
echo "var1 incremented by 1 is $var1."
# Attempt to change variable declared as integer.
echo "Attempting to change var1 to floating point value, 2367.1."
var1=2367.1      # Results in error message, with no change to variable.
echo "var1 is still $var1"

echo

declare -r var2=13.36        # 'declare' permits setting a variable property
                             #+ and simultaneously assigning it a value.
echo "var2 declared as $var2" # Attempt to change readonly variable.
var2=13.37                   # Generates error message, and exit from script.

echo "var2 is still $var2"    # This line will not execute.

exit 0                       # Script will not exit here.
```

Using the *declare* builtin restricts the scope of a variable.

```
foo ()
{
FOO="bar"
}

bar ()
{
foo
echo $FOO
}
```

```
bar   # Prints bar.
```

However . . .

```
foo (){
declare FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar  # Prints nothing.


# Thank you, Michael Iatrou, for pointing this out.
```

9.2.1. Another use for *declare*

The *declare* command can be helpful in identifying variables, environmental or otherwise. This can be especially useful with arrays .

```
bash$

declare | grep HOME

HOME=/home/bozo
bash$

zzy=68

bash$

declare | grep zzy

zzy=68
bash$

Colors=([0]="purple" [1]="reddish-orange" [2]="light green")

bash$

echo ${Colors[@]}

purple reddish-orange light green
bash$

declare | grep Colors

Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
```

Notes

[1] In this context, *typing* a variable means to classify it and restrict its properties. For example, a variable *declared* or *typed* as an integer is no longer available for string operations .

```
declare -i intvar

intvar=23
echo "$intvar"   # 23
intvar=stringval
echo "$intvar"   # 0
```

[Jul 25, 2017] Arrays in bash 4.x

Jul 25, 2017 | wiki.bash-hackers.org

Purpose An array is a parameter that holds mappings from keys to values. Arrays are used to store a collection of parameters into a parameter. Arrays (in any programming language) are a useful and common composite data structure, and one of the most important scripting features in Bash and other shells.

Here is an abstract representation of an array named NAMES . The indexes go from 0 to 3.

```
NAMES
 0: Peter
 1: Anna
 2: Greg
 3: Jan
```

Instead of using 4 separate variables, multiple related variables are grouped grouped together into ***elements*** of the array, accessible by their ***key*** . If you want the second name, ask for index 1 of the array NAMES . **Indexing** Bash supports two different types of ksh-like one-dimensional arrays. Multidimensional arrays are not implemented .

- ***Indexed arrays*** use positive integer numbers as keys. Indexed arrays are always sparse , meaning indexes are not necessarily contiguous. All syntax used for both assigning and dereferencing indexed arrays is an arithmetic evaluation context (see Referencing). As in C and many other languages, the numerical array indexes start at 0 (zero). Indexed arrays are the most common, useful, and portable type. Indexed arrays were first introduced to Bourne-like shells by ksh88. Similar, partially compatible syntax was inherited by many derivatives including Bash. Indexed arrays always carry the -a attribute.
- ***Associative arrays*** (sometimes known as a "hash" or "dict") use arbitrary nonempty strings as keys. In other words, associative arrays allow you to look up a value from a table based upon its corresponding string label. Associative arrays are always unordered , they merely ***associate*** key-value pairs. If you retrieve multiple values from the array at once, you can't count on them coming out in the same order you put them in. Associative arrays always carry the -A attribute, and unlike indexed arrays, Bash requires that they always be declared explicitly (as indexed arrays are the default, see declaration). Associative arrays were first introduced in ksh93, and similar mechanisms were later adopted by Zsh and Bash version 4. These three are currently the only POSIX-compatible shells with any associative array support.

Syntax Referencing To accommodate referring to array variables and their individual elements, Bash extends the parameter naming scheme with a subscript suffix. Any valid ordinary scalar parameter name is also a valid array name: [[:alpha:]_][[:alnum:]_]* . The parameter name may be followed by an optional subscript enclosed in square brackets to refer to a member of the array.

The overall syntax is arrname[subscript] - where for indexed arrays, subscript is any valid arithmetic expression, and for associative arrays, any nonempty string. Subscripts are first processed for parameter and arithmetic expansions, and command and process substitutions. When used within parameter expansions or as an argument to the unset builtin, the special subscripts * and @ are also accepted which act upon arrays analogously to the way the @ and * special parameters act upon the positional parameters. In parsing the subscript, bash ignores any text that follows the closing bracket up to the end of the parameter name.

With few exceptions, names of this form may be used anywhere ordinary parameter names are valid, such as within arithmetic expressions , parameter expansions , and as arguments to builtins that accept parameter names. An ***array*** is a Bash parameter that has been given the -a (for indexed) or -A (for associative) ***attributes*** . However, any regular (non-special or positional) parameter may be validly referenced using a subscript, because in most contexts, referring to the zeroth element of an array is synonymous with referring to the array name without a subscript.

```
# "x" is an ordinary non-array parameter.
$ x=hi; printf '%s ' "$x" "${x[0]}"; echo "${_[0]}"
hi hi hi
```

The only exceptions to this rule are in a few cases where the array variable's name refers to the array as a whole. This is the case for the unset builtin (see destruction) and when declaring an array without assigning any values (see declaration). **Declaration** The following explicitly give variables array attributes, making them arrays:

Syntax	Description
ARRAY=()	Declares an indexed array ARRAY and initializes it to be empty. This can also be used to empty an existing array.
ARRAY[0]=	Generally sets the first element of an indexed array. If no array ARRAY existed before, it is created.
declare -a ARRAY	Declares an indexed array ARRAY . An existing array is not initialized.
declare -A ARRAY	Declares an associative array ARRAY . This is the one and only way to create associative arrays.

Storing values Storing values in arrays is quite as simple as storing values in normal variables.

Syntax	Description
ARRAY[N]=VALUE	Sets the element N of the indexed array ARRAY to VALUE . N can be any valid arithmetic expression
ARRAY[STRING]=VALUE	Sets the element indexed by STRING of the associative array ARRAY .
ARRAY=VALUE	As above. If no index is given, as a default the zeroth element is set to VALUE . Careful, this is even true of associative arrays - there is no error if no key is specified, and the value is assigned to string index "0".
ARRAY=(E1 E2)	Compound array assignment - sets the whole array ARRAY to the given list of elements indexed sequentially starting at zero. The array is unset before assignment unless the += operator is used. When the list is empty (ARRAY=()), the array will be set to an empty array. This method obviously does not use explicit indexes. An associative array can not be set like that! Clearing an associative array using ARRAY=() works.
ARRAY=([X]=E1 [Y]=E2)	Compound assignment for indexed arrays with index-value pairs declared individually (here for example X and Y). X and Y are arithmetic expressions. This syntax can be combined with the above - elements declared without an explicitly specified index are assigned sequentially starting at either the last element with an explicit index, or zero.
ARRAY=([S1]=E1 [S2]=E2)	Individual mass-setting for associative arrays . The named indexes (here: S1 and S2) are strings.
ARRAY+=(E1 E2)	Append to ARRAY.

As of now, arrays can't be exported. **Getting values** article about parameter expansion and check the notes about arrays.

Syntax	Description
${ARRAY[N]}	Expands to the value of the index N in the indexed array ARRAY . If N is a negative number, it's treated as the offset from the maximum assigned index (can't be used for assignment) - 1
${ARRAY[S]}	Expands to the value of the index S in the associative array ARRAY .
"${ARRAY[@]}" ${ARRAY[@]} "${ARRAY[*]}" ${ARRAY[*]}	Similar to mass-expanding positional parameters , this expands to all elements. If unquoted, both subscripts * and @ expand to the same result, if quoted, @ expands to all elements individually quoted, * expands to all elements quoted as a whole.
"${ARRAY[@]:N:M}" ${ARRAY[@]:N:M} "${ARRAY[*]:N:M}" ${ARRAY[*]:N:M}	Similar to what this syntax does for the characters of a single string when doing substring expansion , this expands to M elements starting with element N . This way you can mass-expand individual indexes. The rules for quoting and the subscripts * and @ are the same as above for the other mass-expansions.

For clarification: When you use the subscripts @ or * for mass-expanding, then the behaviour is exactly what it is for $@ and $* when mass-expanding the positional parameters . You should read this article to understand what's going on. **Metadata**

Syntax	Description
${#ARRAY[N]}	Expands to the length of an individual array member at index N (stringlength
${#ARRAY[STRING]}	Expands to the length of an individual associative array member at index STRING (stringlength)
${#ARRAY[@]} ${#ARRAY[*]}	Expands to the number of elements in ARRAY
${!ARRAY[@]} ${!ARRAY[*]}	Expands to the indexes in ARRAY since BASH 3.0

Destruction The unset builtin command is used to destroy (unset) arrays or individual elements of arrays.

Syntax	Description
unset -v ARRAY unset -v ARRAY[@] unset -v ARRAY[*]	Destroys a complete array
unset -v ARRAY[N]	Destroys the array element at index N
unset -v ARRAY[STRING]	Destroys the array element of the associative array at index STRING

It is best to explicitly specify -v when unsetting variables with unset.

pathname expansion to occur due to the presence of glob characters.

Example: You are in a directory with a file named x1 , and you want to destroy an array element x[1] , with

```
unset 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, always quote the array name and index:

```
unset -v 'x[1]'
```

This applies generally to all commands which take variable names as arguments. Single quotes preferred.

Usage Numerical Index Numerical indexed arrays are easy to understand and easy to use. The Purpose and Indexing chapters above more or less explain all the needed background theory.

Now, some examples and comments for you.

Let's say we have an array sentence which is initialized as follows:

```
sentence=(Be liberal in what you accept, and conservative in what you send)
```

Since no special code is there to prevent word splitting (no quotes), every word there will be assigned to an individual array element. When you count the words you see, you should get 12. Now let's see if Bash has the same opinion:

```
$ echo ${#sentence[@]}
12
```

Yes, 12. Fine. You can take this number to walk through the array. Just subtract 1 from the number of elements, and start your walk at 0 (zero)

```
((n_elements=${#sentence[@]}, max_index=n_elements - 1))

for ((i = 0; i <= max_index; i++)); do
  echo "Element $i: '${sentence[i]}'"
done
```

You always have to remember that, it seems newbies have problems sometimes. Please understand that numerical array indexing begins at 0 (zero)

The method above, walking through an array by just knowing its number of elements, only works for arrays where all elements are set, of course. If one element in the middle is removed, then the calculation is nonsense, because the number of elements doesn't correspond to the highest used index anymore (we call them " ***sparse arrays*** "). **Associative (Bash 4)** Associative arrays (or ***hash tables***) are not much more complicated than numerical indexed arrays. The numerical index value (in Bash a number starting at zero) just is replaced with an arbitrary string:

```
# declare -A, introduced with Bash 4 to declare an associative array
declare -A sentence

sentence[Begin]='Be liberal in what'
sentence[Middle]='you accept, and conservative'
sentence[End]='in what you send'
sentence['Very end']=...
```

Beware: don't rely on the fact that the elements are ordered in memory like they were declared, it could look like this:

```
# output from 'set' command
sentence=([End]="in what you send" [Middle]="you accept, and conservative " [Begin]="Be liberal in what " ["Very end"]="...")
```

This effectively means, you can get the data back with "${sentence[@]}" , of course (just like with numerical indexing), but you can't rely on a specific order. If you want to store ordered data, or re-order data, go with numerical indexes. For associative arrays, you usually query known index values:

```
for element in Begin Middle End "Very end"; do
   printf "%s" "${sentence[$element]}"
done
printf "\n"
```

A nice code example: Checking for duplicate files using an associative array indexed with the SHA sum of the files:

```
# Thanks to Tramp in #bash for the idea and the code

unset flist; declare -A flist;
```

```
while read -r sum fname; do
  if [[ ${flist[$sum]} ]]; then
    printf 'rm -- "%s" # Same as >%s<\n' "$fname" "${flist[$sum]}"
  else
    flist[$sum]="$fname"
  fi
done < <(find . -type f -exec sha256sum {} +) >rmdups
```

Integer arrays Any type attributes applied to an array apply to all elements of the array. If the integer attribute is set for either indexed or associative arrays, then values are considered as arithmetic for both compound and ordinary assignment, and the += operator is modified in the same way as for ordinary integer variables.

```
~ $ ( declare -ia 'a=(2+4 [2]=2+2 [a[2]]="a[2]")' 'a+=(42 [a[4]]+=3)'; declare -p a )
declare -ai a='([0]="6" [2]="4" [4]="7" [5]="42")'
```

a[0] is assigned to the result of 2+4 . a[1] gets the result of 2+2 . The last index in the first assignment is the result of a[2] , which has already been assigned as 4 , and its value is also given a[2] .

This shows that even though any existing arrays named a in the current scope have already been unset by using = instead of += to the compound assignment, arithmetic variables within keys can self-reference any elements already assigned within the same compound-assignment. With integer arrays this also applies to expressions to the right of the = . (See evaluation order , the right side of an arithmetic assignment is typically evaluated first in Bash.)

The second compound assignment argument to declare uses += , so it appends after the last element of the existing array rather than deleting it and creating a new array, so a[5] gets 42 .

Lastly, the element whose index is the value of a[4] (4), gets 3 added to its existing value, making a[4] == 7 . Note that having the integer attribute set this time causes += to add, rather than append a string, as it would for a non-integer array.

The single quotes force the assignments to be evaluated in the environment of declare . This is important because attributes are only applied to the assignment after assignment arguments are processed. Without them the += compound assignment would have been invalid, and strings would have been inserted into the integer array without evaluating the arithmetic. A special-case of this is shown in the next section.

eval , but there are differences.) 'Todo: ' Discuss this in detail.

Indirection Arrays can be expanded indirectly using the indirect parameter expansion syntax. Parameters whose values are of the form: name[index] , name[@] , or name[*] when expanded indirectly produce the expected results. This is mainly useful for passing arrays (especially multiple arrays) by name to a function.

This example is an "isSubset"-like predicate which returns true if all key-value pairs of the array given as the first argument to isSubset correspond to a key-value of the array given as the second argument. It demonstrates both indirect array expansion and indirect key-passing without eval using the aforementioned special compound assignment expansion.

```
isSubset() {
  local -a 'xkeys=("${!"$1"[@]}")' 'ykeys=("${!"$2"[@]}")'
  set -- "${@/%/[key]}"

  (( ${#xkeys[@]} <= ${#ykeys[@]} )) || return 1

  local key
  for key in "${xkeys[@]}"; do
    [[ ${!2+_} && ${!1} == ${!2} ]] || return 1
  done
}

main() {
  # "a" is a subset of "b"
  local -a 'a=({0..5})' 'b=({0..10})'
  isSubset a b
  echo $? # true

  # "a" contains a key not in "b"
  local -a 'a=([5]=5 {6..11})' 'b=({0..10})'
  isSubset a b
  echo $? # false

  # "a" contains an element whose value != the corresponding member of "b"
  local -a 'a=([5]=5 6 8 9 10)' 'b=({0..10})'
  isSubset a b
  echo $? # false
}

main
```

This script is one way of implementing a crude multidimensional associative array by storing array definitions in an array and referencing them through indirection. The script takes two keys and dynamically calls a function whose name is resolved from the array.

```
callFuncs() {
  # Set up indirect references as positional parameters to minimize local name collisions.
  set -- "${@:1:3}" ${2+'a["$1"]' "$1"'["$2"]'}

  # The only way to test for set but null parameters is unfortunately to test each individually.
  local x
  for x; do
    [[ $x ]] || return 0
  done

  local -A a=(
    [foo]='([r]=f [s]=g [t]=h)'
    [bar]='([u]=i [v]=j [w]=k)'
    [baz]='([x]=l [y]=m [z]=n)'
    ) ${4+${a["$1"]+"${1}=${!3}"}} # For example, if "$1" is "bar" then define a new array: bar=([u]=i [v]=j [w]=k)

  ${4+${a["$1"]+"${!4-:}"}} # Now just lookup the new array. for inputs: "bar" "v", the function named "j" will be called, which prints "j" to stdout.
```

```
}

main() {
   # Define functions named {f..n} which just print their own names.
   local fun='() { echo "$FUNCNAME"; }' x

   for x in {f..n}; do
      eval "${x}${fun}"
   done

   callFuncs "$@"
}

main "$@"
```

Bugs and Portability Considerations

- Arrays are not specified by POSIX. One-dimensional indexed arrays are supported using similar syntax and semantics by most Korn-like shells.
- Associative arrays are supported via typeset -A in Bash 4, Zsh, and Ksh93.
- In Ksh93, arrays whose types are not given explicitly are not necessarily indexed. Arrays defined using compound assignments which specify subscripts are associative by default. In Bash, associative arrays can ***only*** be created by explicitly declaring them as associative, otherwise they are always indexed. In addition, ksh93 has several other compound structures whose types can be determined by the compound assignment syntax used to create them.
- In Ksh93, using the = compound assignment operator unsets the array, including any attributes that have been set on the array prior to assignment. In order to preserve attributes, you must use the += operator. However, declaring an associative array, then attempting an a=( ) style compound assignment without specifying indexes is an error. I can't explain this inconsistency.

```
$ ksh -c 'function f { typeset -a a; a=([0]=foo [1]=bar); typeset -p a; }; f' # Attribute is lost, and since subscripts are given, we default to associative.
typeset -A a=([0]=foo [1]=bar)
$ ksh -c 'function f { typeset -a a; a+=([0]=foo [1]=bar); typeset -p a; }; f' # Now using += gives us the expected results.
typeset -a a=(foo bar)
$ ksh -c 'function f { typeset -A a; a=(foo bar); typeset -p a; }; f' # On top of that, the reverse does NOT unset the attribute. No idea why.
ksh: f: line 1: cannot append index array to associative array a
```

- Only Bash and mksh support compound assignment with mixed explicit subscripts and automatically incrementing subscripts. In ksh93, in order to specify individual subscripts within a compound assignment, all subscripts must be given (or none). Zsh doesn't support specifying individual subscripts at all.
- Appending to a compound assignment is a fairly portable way to append elements after the last index of an array. In Bash, this also sets append mode for all individual assignments within the compound assignment, such that if a lower subscript is specified, subsequent elements will be appended to previous values. In ksh93, it causes subscripts to be ignored, forcing appending everything after the last element. (Appending has different meaning due to support for multi-dimensional arrays and nested compound datastructures.)

```
$ ksh -c 'function f { typeset -a a; a+=(foo bar baz); a+=([3]=blah [0]=bork [1]=blarg [2]=zooj); typeset -p a; }; f' # ksh93 forces appending to the ar
typeset -a a=(foo bar baz '[3]=blah' '[0]=bork' '[1]=blarg' '[2]=zooj')
$ bash -c 'function f { typeset -a a; a+=(foo bar baz); a+=(blah [0]=bork blarg zooj); typeset -p a; }; f' # Bash applies += to every individual subscrip
declare -a a='([0]="foobork" [1]="barblarg" [2]="bazzooj" [3]="blah")'
$ mksh -c 'function f { typeset -a a; a+=(foo bar baz); a+=(blah [0]=bork blarg zooj); typeset -p a; }; f' # Mksh does like Bash, but clobbers previous
set -A a
typeset a[0]=bork
typeset a[1]=blarg
typeset a[2]=zooj
typeset a[3]=blah
```

- In Bash and Zsh, the alternate value assignment parameter expansion (${arr[idx]:=foo}) evaluates the subscript twice, first to determine whether to expand the alternate, and second to determine the index to assign the alternate to. See evaluation order .

```
$ : ${_[$(echo $RANDOM >&2)1]:=$(echo hi >&2)}
13574
hi
14485
```

- In Zsh, arrays are indexed starting at 1 in its default mode. Emulation modes are required in order to get any kind of portability.
- Zsh and mksh do not support compound assignment arguments to typeset .
- Ksh88 didn't support modern compound array assignment syntax. The original (and most portable) way to assign multiple elements is to use the set -A name arg1 arg2 syntax. This is supported by almost all shells that support ksh-like arrays except for Bash. Additionally, these shells usually support an optional -s argument to set which performs lexicographic sorting on either array elements or the positional parameters. Bash has no built-in sorting ability other than the usual comparison operators.

```
$ ksh -c 'set -A arr -- foo bar bork baz; typeset -p arr' # Classic array assignment syntax
typeset -a arr=(foo bar bork baz)
$ ksh -c 'set -sA arr -- foo bar bork baz; typeset -p arr' # Native sorting!
typeset -a arr=(bar baz bork foo)
$ mksh -c 'set -sA arr -- foo "[3]=bar" "[2]=baz" "[7]=bork"; typeset -p arr' # Probably a bug. I think the maintainer is aware of it.
set -A arr
typeset arr[2]=baz
typeset arr[3]=bar
typeset arr[7]=bork
typeset arr[8]=foo
```

- Evaluation order for assignments involving arrays varies significantly depending on context. Notably, the order of evaluating the subscript or the value first can change in almost every shell for both expansions and arithmetic variables. See evaluation order for details.
- Bash 4.1.* and below cannot use negative subscripts to address array indexes relative to the highest-numbered index. You must use the subscript expansion, i.e. "${arr[@]:(-n):1}" , to expand the nth-last element (or the next-highest indexed after n if arr[n] is unset). In Bash 4.2, you may expand (but not assign to) a negative index. In Bash 4.3, ksh93, and zsh, you may both assign and expand negative offsets.
- ksh93 also has an additional slice notation: "${arr[n..m]}" where n and m are arithmetic expressions. These are needed for use with multi-dimensional arrays.
- Assigning or referencing negative indexes in mksh causes wrap-around. The max index appears to be UINT_MAX , which would be addressed by arr[-1] .
- So far, Bash's -v var test doesn't support individual array subscripts. You may supply an array name to test whether an array is defined, but can't check an element. ksh93's -v supports both. Other shells lack a -v test.

Bugs

- Fixed in 4.3 Bash 4.2.* and earlier considers each chunk of a compound assignment, including the subscript for globbing. The subscript part is considered quoted, but any unquoted glob characters on the right-hand side of the [ ]= will be clumped with the subscript and counted as a glob. Therefore, you must

quote anything on the right of the = sign. This is fixed in 4.3, so that each subscript assignment statement is expanded following the same rules as an ordinary assignment. This also works correctly in ksh93.

```
$ touch '[1]=a'; bash -c 'a=([1]=*); echo "${a[@]}"'
[1]=a
```

mksh has a similar but even worse problem in that the entire subscript is considered a glob.

```
$ touch 1=a; mksh -c 'a=([123]=*); print -r -- "${a[@]}"'
1=a
```

- Fixed in 4.3 In addition to the above globbing issue, assignments preceding "declare" have an additional effect on brace and pathname expansion.

```
$ set -x; foo=bar declare arr=( {1..10} )
+ foo=bar
+ declare 'a=(1)' 'a=(2)' 'a=(3)' 'a=(4)' 'a=(5)'

$ touch xy=foo
$ declare x[y]=*
+ declare 'x[y]=*'
$ foo=bar declare x[y]=*
+ foo=bar
+ declare xy=foo
```

Each word (the entire assignment) is subject to globbing and brace expansion. This appears to trigger the same strange expansion mode as let , eval , other declaration commands, and maybe more.

- Fixed in 4.3 Indirection combined with another modifier expands arrays to a single word.

```
$ a=({a..c}) b=a[@]; printf '<%s> ' "${!b}"; echo; printf '<%s> ' "${!b/%/foo}"; echo
<a> <b> <c>
<a b cfoo>
```

- Fixed in 4.3 Process substitutions are evaluated within array indexes. Zsh and ksh don't do this in any arithmetic context.

```
# print "moo"
dev=fd=1 _[1<(echo moo >&2)]=

# Fork bomb
${dev[${dev='dev[1>(${dev[dev]})]'}]}
```

Evaluation order Here are some of the nasty details of array assignment evaluation order. You can use this testcase code to generate these results.

```
Each testcase prints evaluation order for indexed array assignment
contexts. Each context is tested for expansions (represented by digits) and
arithmetic (letters), ordered from left to right within the expression. The
output corresponds to the way evaluation is re-ordered for each shell:

a[ $1 a ]=${b[ $2 b ]:=${c[ $3 c ]}}             No attributes
a[ $1 a ]=${b[ $2 b ]:=c[ $3 c ]}               typeset -ia a
a[ $1 a ]=${b[ $2 b ]:=c[ $3 c ]}               typeset -ia b
a[ $1 a ]=${b[ $2 b ]:=c[ $3 c ]}               typeset -ia a b
(( a[ $1 a ] = b[ $2 b ] ${c[ $3 c ]} ))        No attributes
(( a[ $1 a ] = ${b[ $2 b ]:=c[ $3 c ]} ))        typeset -ia b
a+=( [ $1 a ]=${b[ $2 b ]:=${c[ $3 c ]}} [ $4 d ]=$(( $5 e )) ) typeset -a a
a+=( [ $1 a ]=${b[ $2 b ]:=c[ $3 c ]} [ $4 d ]=${5}e ) typeset -ia a

bash: 4.2.42(1)-release
2 b 3 c 2 b 1 a
2 b 3 2 b 1 a c
2 b 3 2 b c 1 a
2 b 3 2 b c 1 a c
1 2 3 c b a
1 2 b 3 2 b c c a
1 2 b 3 c 2 b 4 5 e a d
1 2 b 3 2 b 4 5 a c d e

ksh93: Version AJM 93v- 2013-02-22
1 2 b b a
1 2 b b a
1 2 b b a
1 2 b b a
1 2 3 c b a
1 2 b b a
1 2 b b a 4 5 e d
1 2 b b a 4 5 d e

mksh: @(#)MIRBSD KSH R44 2013/02/24
2 b 3 c 1 a
2 b 3 1 a c
2 b 3 c 1 a
2 b 3 c 1 a
1 2 3 c a b
1 2 b 3 c a
1 2 b 3 c 4 5 e a d
1 2 b 3 4 5 a c d e

zsh: 5.0.2
2 b 3 c 2 b 1 a
2 b 3 2 b 1 a c
```

```
2 b 1 a
2 b 1 a
1 2 3 c b a
1 2 b a
1 2 b 3 c 2 b 4 5 e
1 2 b 3 2 b 4 5
```

See also

- Parameter expansion (contains sections for arrays)
- The classic for-loop (contains some examples to iterate over arrays)
- The declare builtin command
- BashFAQ 005 - How can I use array variables? - A very detailed discussion on arrays with many examples.
- BashSheet - Arrays - Bashsheet quick-reference on Greycat's wiki.

[Jul 25, 2017] Local variables

Notable quotes:

"... completely local and separate ..."

Jul 25, 2017 | wiki.bash-hackers.org

local to a function:

- Using the local keyword, or
- Using declare (which will ***detect*** when it was called from within a function and make the variable(s) local).

```
myfunc
()
local
var
=VALUE

# alternative, only when used INSIDE a function
declare
var
=VALUE

...
```

The ***local*** keyword (or declaring a variable using the declare command) tags a variable to be treated ***completely local and separate*** inside the function where it was declared:

```
foo
=external

printvalue
()
local
foo
=internal

echo
$foo



# this will print "external"
echo
$foo


# this will print "internal"

printvalue

# this will print - again - "external"
echo
$foo
```

[Jul 25, 2017] Environment variables

Notable quotes:

"... environment variables ..."

"... including the environment variables ..."

Jul 25, 2017 | wiki.bash-hackers.org

The environment space is not directly related to the topic about scope, but it's worth mentioning.

Every UNIX® process has a so-called ***environment*** . Other items, in addition to variables, are saved there, the so-called ***environment variables*** . When a child process is created (in Bash e.g. by simply executing another program, say ls to list files), the whole environment ***including the environment variables*** is copied to the new process. Reading that from the other side means: Only variables that are part of the environment are available in the child process.

A variable can be tagged to be part of the environment using the export command:

```
# create a new variable and set it:
# -> This is a normal shell variable, not an environment variable!
myvariable
"Hello world."


# make the variable visible to all child processes:
# -> Make it an environment variable: "export" it
export
 myvariable
```

Remember that the ***exported*** variable is a copy . There is no provision to "copy it back to the parent." See the article about Bash in the process tree !

[1)] under specific circumstances, also by the shell itself

[Jul 25, 2017] Block commenting

Jul 25, 2017 | wiki.bash-hackers.org

: (colon) and input redirection. The : does nothing, it's a pseudo command, so it does not care about standard input. In the following code example, you want to test mail and logging, but not dump the database, or execute a shutdown:

```
#!/bin/bash
# Write info mails, do some tasks and bring down the system in a safe way
echo
"System halt requested"
 mail
-s
"System halt"
 netadmin
example.com
logger
-t
 SYSHALT
"System halt requested"


##### The following "code block" is effectively ignored

:
<<
"SOMEWORD"
etc
init.d
mydatabase clean_stop
mydatabase_dump
var
db
db1
mnt
fsrv0
backups
db1
logger
-t
 SYSHALT
"System halt: pre-shutdown actions done, now shutting down the system"

shutdown
-h
 NOW
SOMEWORD
##### The ignored codeblock ends here
```

What happened? The : pseudo command was given some input by redirection (a here-document) - the pseudo command didn't care about it, effectively, the entire block was ignored.

The here-document-tag was quoted here to avoid substitutions in the "commented" text! Check redirection with here-documents for more

[Jul 25, 2017] Doing specific tasks: concepts, methods, ideas

Notable quotes:

"... under construction! ..."

Jul 25, 2017 | wiki.bash-hackers.org

- Simple locking (against parallel run)
- Rudimentary config files for your scripts
- Editing files with ed(1)
- Collapsing Functions
- Illustrated Redirection Tutorial
- Calculate with dc(1)
- Introduction to pax - the POSIX archiver
- Small getopts tutorial (***under construction!***)
- Dissect a bad oneliner An example of a bad oneliner, breakdown and fix (by kojoro)
- Write tests for ./your-script.sh by using bashtest util

[Jul 25, 2017] Keeping persistent history in bash

Jul 25, 2017 | eli.thegreenplace.net

June 11, 2013 at 19:27 Tags Linux , Software & Tools

Update (Jan 26, 2016): I posted a short update about my usage of persistent history.

For someone spending most of his time in front of a Linux terminal, history is very important. But traditional bash history has a number of limitations, especially when multiple terminals are involved (I sometimes have dozens open). Also it's not very good at preserving just the history you're interested in across reboots.

There are many approaches to improve the situation; here I want to discuss one I've been using very successfully in the past few months - a simple "persistent history" that keeps track of history across terminal instances, saving it into a dot-file in my home directory (~/.persistent_history). All commands, from all terminal instances, are saved there, forever. I found this tremendously useful in my work - it saves me time almost every day.

Why does it go into a ***separate*** history and not the ***main*** one which is accessible by all the existing history manipulation tools? Because IMHO the latter is still worthwhile to be kept separate for the simple need of bringing up recent commands in a single terminal, without mixing up commands from other terminals. While the terminal is open, I want the press "Up" and get the previous command, even if I've executed a 1000 other commands in other terminal instances in the meantime.

Persistent history is very easy to set up. Here's the relevant portion of my ~/.bashrc :

```
log_bash_persistent_history()
{
  [[
    $(history 1) =~ ^\ *[0-9]+\ +([^\ ]+\ [^\ ]+)\ +(.*)$
  ]]
  local date_part="${BASH_REMATCH[1]}"
  local command_part="${BASH_REMATCH[2]}"
  if [ "$command_part" != "$PERSISTENT_HISTORY_LAST" ]
  then
    echo $date_part "|" "$command_part" >> ~/.persistent_history
    export PERSISTENT_HISTORY_LAST="$command_part"
  fi
}

# Stuff to do on PROMPT_COMMAND
run_on_prompt_command()
{
    log_bash_persistent_history
}

PROMPT_COMMAND="run_on_prompt_command"
```

The format of the history file created by this is:

```
2013-06-09 17:48:11 | cat ~/.persistent_history
2013-06-09 17:49:17 | vi /home/eliben/.bashrc
2013-06-09 17:49:23 | ls
```

Note that an environment variable is used to avoid useless duplication (i.e. if I run ls twenty times in a row, it will only be recorded once).

OK, so we have ~/.persistent_history , how do we ***use*** it? First, I should say that it's not used very often, which kind of connects to the point I made earlier about separating it from the much higher-use regular command history. Sometimes I just look into the file with vi or tail , but mostly this alias does the trick for me:

```
alias phgrep='cat ~/.persistent_history|grep --color'
```

The alias name mirrors another alias I've been using for ages:

```
alias hgrep='history|grep --color'
```

Another tool for managing persistent history is a trimmer. I said earlier this file keeps the history "forever", which is a scary word - what if it grows too large? Well, first of all - worry not. At work my history file grew to about 2 MB after 3 months of heavy usage, and 2 MB is pretty small these days. Appending to the end of a file is very, very quick (I'm pretty sure it's a constant-time operation) so the size doesn't matter much. But trimming is easy:

```
tail -20000 ~/.persistent_history | tee ~/.persistent_history
```

Trims to the last 20000 lines. This should be sufficient for at least a couple of months of history, and your workflow should not really rely on more than that :-)

Finally, what's the use of having a tool like this without employing it to collect some useless statistics. Here's a histogram of the 15 most common commands I've used on my home machine's terminal over the past 3 months:

```
ls      : 865
vi      : 863
hg      : 741
cd      : 512
ll      : 289
pss     : 245
hst     : 200
python  : 168
make    : 167
```

```
git      : 148
time     : 94
python3  : 88
./python : 88
hpu      : 82
cat      : 80
```

Some explanation: hst is an alias for hg st . hpu is an alias for hg pull -u . pss is my awesome pss tool , and is the reason why you don't see any calls to grep and find in the list. The proportion of Mercurial vs. git commands is likely to change in the very

[Jul 24, 2017] Bash history handling with multiple terminals

Add to your Prompt command history -a to preserve history from multiple terminals. This is a very neat trick !!!

get=

Bash history handling with multiple terminals

The **bash** session that is saved is the one for the **terminal** that is closed the latest. If you want to save the commands for every session, you could use the trick explained here.

```
export PROMPT_COMMAND='history -a'
```

To quote the manpage: "If set, the value is executed as a command prior to issuing each primary prompt."

So every time my command has finished, it appends the unwritten **history** item to ~/.bash

ATTENTION: If you use multiple shell sessions and do not use this trick, you need to write the history manually to preserver it using the command history -a

See also:

- https://unix.stackexchange.com/questions/1288/preserve-bash-history-in-multiple-terminal-windows
- http://northernmost.org/blog/flush-bash_history-after-each-command/comment-page-1/index.html#comment-640
- Keeping persistent history in bash - Eli Bendersky's website

[Mar 13, 2017] 6.3 Arrays

Notable quotes:

"... aname val1 val2 val3 ..."

"... aname ..."

"... type ..."

"... ad hoc ..."

Mar 13, 2017 | name="KSH-CH-6-SECT-3">

So far we have seen two types of variables: character strings and integers. The third type of variable the Korn shell supports is an ***array*** . As you may know, an array is like a list of things; you can refer to specific elements in an array with integer ***indices*** , so that ***a[i]*** refers to the ***i*** th element of array ***a*** .

The Korn shell provides an array facility that, while useful, is much more limited than analogous features in conventional programming languages. In particular, arrays can be only one-dimensional (i.e., no arrays of arrays), and they are limited to 1024 elements. Indices can start at 0.

There are two ways to assign values to elements of an array. The first is the most intuitive: you can use the standard shell variable assignment syntax with the array index in brackets (**[]**). For example:

```
nicknames[2]=bob
nicknames[3]=ed
```

puts the values **bob** and **ed** into the elements of the array **nicknames** with indices 2 and 3, respectively. As with regular shell variables, values assigned to array elements are treated as character strings unless the assignment is preceded by **let** .

The second way to assign values to an array is with a variant of the **set** statement, which we saw in Chapter 3, Customizing Your Environment . The statement:

```
set -A

aname val1 val2 val3

 ...
```

creates the array ***aname*** (if it doesn't already exist) and assigns ***val1*** to ***aname[0]*** , ***val2*** to ***aname[1]*** , etc. As you would guess, this is more convenient for loading up an array with an initial set of values.

To extract a value from an array, use the syntax **${** ***aname*** **[** ***i*** **]}** . For example, **${nicknames[2]}** has the value "bob". The index ***i*** can be an arithmetic expression-see above. If you use * in place of the index, the value will be all elements, separated by spaces. Omitting the index is the same as specifying index 0.

Now we come to the somewhat unusual aspect of Korn shell arrays. Assume that the only values assigned to **nicknames** are the two we saw above. If you type **print** " **${nicknames[** * **]}"** , you will see the output:

```
bob ed
```

In other words, **nicknames[0]** and **nicknames[1]** don't exist. Furthermore, if you were to type:

```
nicknames[9]=pete
nicknames[31]=ralph
```

and then type **print '' ${nicknames[*]}"** , the output would look like this:

```
bob ed pete ralph
```

This is why we said "the elements of **nicknames** with indices 2 and 3" earlier, instead of "the 2nd and 3rd elements of **nicknames** ". Any array elements with unassigned values just don't exist; if you try to access their values, you will get null strings.

You can preserve whatever whitespace you put in your array elements by using '' **$** { ***aname*** **[@]** } '' (with the double quotes) instead of **$** { ***aname*** **[*]** } '' , just as you can with '' **$@** '' instead of **$** * .

The shell provides an operator that tells you how many elements an array has defined: **${#** ***aname*** **[*] }** . Thus **${#nicknames[*] }** has the value 4. Note that you need the **[*]** because the name of the array alone is interpreted as the 0th element. This means, for example, that **${#nicknames}** equals the length of **nicknames[0]** (see Chapter 4). Since **nicknames[0]** doesn't exist, the value of **${#nicknames}** is 0, the length of the null string.

To be quite frank, we feel that the Korn shell's array facility is of little use to shell programmers. This is partially because it is so limited, but mainly because shell programming tasks are much more often oriented toward character strings and text than toward numbers. If you think of an array as a mapping from integers to values (i.e., put in a number, get out a value), then you can see why arrays are "number-dominated" data structures.

Nevertheless, we can find useful things to do with arrays. For example, here is a cleaner solution to Task 5-4, in which a user can select his or her terminal type (**TERM** environment variable) at login time. Recall that the "user-friendly" version of this code used **select** and a **case** statement:

```
print 'Select your terminal type:'
PS3='terminal? '
select term in
   'Givalt GL35a' \
   'Tsoris T-2000' \
   'Shande 531' \
   'Vey VT99'
do
   case $REPLY in
      1 ) TERM=gl35a ;;
      2 ) TERM=t2000 ;;
      3 ) TERM=s531 ;;
      4 ) TERM=vt99 ;;
      * ) print "invalid." ;;
   esac
   if [[ -n $term ]]; then
      print "TERM is $TERM"
      break
   fi
done
```

We can eliminate the entire **case** construct by taking advantage of the fact that the **select** construct stores the user's number choice in the variable **REPLY** . We just need a line of code that stores all of the possibilities for **TERM** in an array, in an order that corresponds to the items in the **select** menu. Then we can use **$REPLY** to index the array. The resulting code is:

```
set -A termnames gl35a t2000 s531 vt99
print 'Select your terminal type:'
PS3='terminal? '
select term in
   'Givalt GL35a' \
   'Tsoris T-2000' \
   'Shande 531' \
   'Vey VT99'
do
   if [[ -n $term ]]; then
      TERM=${termnames[REPLY-1]}
      print "TERM is $TERM"
      break
   fi
done
```

This code sets up the array **termnames** so that **${termnames[0]}** is "gl35a", **${termnames[1]}** is "t2000", etc. The line **TERM=${termnames[REPLY-1]}** essentially replaces the entire **case** construct by using **REPLY** to index the array.

Notice that the shell knows to interpret the text in an array index as an arithmetic expression, as if it were enclosed in **((** and **))** , which in turn means that variable need not be preceded by a dollar sign (**$**). We have to subtract 1 from the value of **REPLY** because array indices start at 0, while **select** menu item numbers start at 1.

6.3.1 typeset

The final Korn shell feature that relates to the kinds of values that variables can hold is the **typeset** command. If you are a programmer, you might guess that **typeset** is used to specify the ***type*** of a variable (integer, string, etc.); you'd be partially right.

typeset is a rather ***ad hoc*** collection of things that you can do to variables that restrict the kinds of values they can take. Operations are specified by options to **typeset** ; the basic syntax is:

```
typeset

-o varname

[=

value

]
```

Options can be combined; multiple ***varname*** s can be used. If you leave out ***varname*** , the shell prints a list of variables for which the given option is turned on.

The options available break down into two basic categories:

1. String formatting operations, such as right- and left-justification, truncation, and letter case control.
2. Type and attribute functions that are of primary interest to advanced programmers.

6.3.2 Local Variables in Functions

typeset without options has an important meaning: if a **typeset** statement is inside a function definition, then the variables involved all become ***local*** to that function (in addition to any properties they may take on as a result of **typeset** options). The ability to define variables that are local to "subprogram" units (procedures, functions, subroutines, etc.) is necessary for writing large programs, because it helps keep subprograms independent of the main program and of each other.

If you just want to declare a variable local to a function, use **typeset** without any options. For example:

```
function afunc {
   typeset diffvar
   samevar=funcvalue
   diffvar=funcvalue
   print "samevar is $samevar"
   print "diffvar is $diffvar"
}

samevar=globvalue
diffvar=globvalue
print "samevar is $samevar"
print "diffvar is $diffvar"
afunc
print "samevar is $samevar"
print "diffvar is $diffvar"
```

This code will print the following:

```
samevar is globvalue
diffvar is globvalue
samevar is funcvalue
diffvar is funcvalue
samevar is funcvalue
diffvar is globvalue
```

Figure 6.1 shows this graphically.

Figure 6.1: Local variables in functions

[Mar 13, 2017] Leaning the Korn shell: Chapter 6 Integer Variables and Arithmetic

Mar 13, 2017 | docstore.mik.ua

6.2 Integer Variables and Arithmetic

The expression **$(($OPTIND - 1))** in the last example gives a clue as to how the shell can do integer arithmetic. As you might guess, the shell interprets words surrounded by **$((** and **))** as arithmetic expressions. Variables in arithmetic expressions do *not* need to be preceded by dollar signs, though it is not wrong to do so.

Arithmetic expressions are evaluated inside double quotes, like tildes, variables, and command substitutions. We're *finally* in a position to state the definitive rule about quoting strings: When in doubt, enclose a string in single quotes, unless it contains tildes or any expression involving a dollar sign, in which case you should use double quotes.

date (1) command on System V-derived versions of UNIX accepts arguments that tell it how to format its output. The argument **+%j** tells it to print the day of the year, i.e., the number of days since December 31st of the previous year.

We can use **+%j** to print a little holiday anticipation message:

```
print "Only $(( (365-$(date +%j)) / 7 )) weeks until the New Year!"
```

We'll show where this fits in the overall scheme of command-line processing in Chapter 7, Input/Output and Command-line Processing .

The arithmetic expression feature is built in to the Korn shell's syntax, and was available in the Bourne shell (most versions) only through the external command *expr* (1). Thus it is yet another example of a desirable feature provided by an external command (i.e., a syntactic kludge) being better integrated into the shell. **[[/]]** and **getopts** are also examples of this design trend.

Korn shell arithmetic expressions are equivalent to their counterparts in the C language. [5] Precedence and associativity are the same as in C. Table 6.2 shows the arithmetic operators that are supported. Although some of these are (or contain) special characters, there is no need to backslash-escape them, because they are within the **$((...))** syntax.

[5] The assignment forms of these operators are also permitted. For example, **$((x += 2))** adds 2 to **x** and stores the result back in **x** .

Table 6.2: Arithmetic Operators

Operator	Meaning
+	Plus
-	Minus
*	Times
/	Division (with truncation)
%	Remainder
<<	Bit-shift left
>>	Bit-shift right
&	Bitwise and
|	Bitwise or
~	Bitwise not
^	Bitwise exclusive or

Parentheses can be used to group subexpressions. The arithmetic expression syntax also (like C) supports relational operators as "truth values" of 1 for true and 0 for false. Table 6.3 shows the relational operators and the logical operators that can be used to combine relational expressions.

Table 6.3: Relational Operators

Operator	Meaning
<	Less than
>	Greater than

Operator	Meaning
<=	Less than or equal
>=	Greater than or equal
==	Equal
!=	Not equal
&&	Logical and
||	Logical or

For example, **$((3 > 2))** has the value 1; **$(((3 > 2) || (4 <= 1)))** also has the value 1, since at least one of the two subexpressions is true.

The shell also supports base *N* numbers, where *N* can be up to 36. The notation *B* **#** *N* means " *N* base *B* ". Of course, if you omit the *B* **#** , the base defaults to 10.

6.2.1 Arithmetic Conditionals

Another construct, closely related to **$((...))** , is **((...))** (without the leading dollar sign). We use this for evaluating arithmetic condition tests, just as **[[...]]** is used for string, file attribute, and other types of tests.

((...)) evaluates relational operators differently from **$((...))** so that you can use it in **if** and **while** constructs. Instead of producing a textual result, it just sets its exit status according to the truth of the expression: 0 if true, 1 otherwise. So, for example, **((3 > 2))** produces exit status 0, as does **(((3 > 2) || (4 <= 1)))** , but **(((3 > 2) && (4 <= 1)))** has exit status 1 since the second subexpression isn't true.

You can also use numerical values for truth values within this construct. It's like the analogous concept in C, which means that it's somewhat counterintuitive to non-C programmers: a value of 0 means *false* (i.e., returns exit status 1), and a non-0 value means *true* (returns exit status 0), e.g., **((14))** is true. See the code for the *kshdb* debugger in Chapter 9 for two more examples of this.

6.2.2 Arithmetic Variables and Assignment

The **((** ... **))** construct can also be used to define integer variables and assign values to them. The statement:

```
(( intvar=expression ))
```

creates the integer variable *intvar* (if it doesn't already exist) and assigns to it the result of *expression* .

That syntax isn't intuitive, so the shell provides a better equivalent: the built-in command **let** . The syntax is:

```
let intvar=expression
```

It is not necessary (because it's actually redundant) to surround the expression with **$((** and **))** in a **let** statement. As with any variable assignment, there must not be any space on either side of the equal sign (**=**). It is good practice to surround expressions with quotes, since many characters are treated as special by the shell (e.g., * , **#** , and parentheses); furthermore, you must quote expressions that include whitespace (spaces or TABs). See Table 6.4 for examples.

Table 6.4: Sample Integer Expression Assignments

Assignment	Value
let x=	**$x**
1+4	5
' 1 + 4 '	5
' (2+3) * 5 '	25
' 2 + 3 * 5 '	17
' 17 / 3 '	5
' 17 % 3 '	2
' 1<<4 '	16
' 48>>3 '	6
' 17 & 3 '	1
' 17 | 3 '	19
' 17 ^ 3 '	18

Here is a small task that makes use of integer arithmetic.

Task 6.1

> Write a script called *pages* that, given the name of a text file, tells how many pages of output it contains. Assume that there are 66 lines to a page but provide an option allowing the user to override that.

We'll make our option **-** *N* , a la *head* . The syntax for this single option is so simple that we need not bother with **getopts** . Here is the code:

```
if [[ $1 = -+([0-9]) ]]; then
   let page_lines=${1#-}
   shift
else
   let page_lines=66
fi
let file_lines="$(wc -l < $1)"

let pages=file_lines/page_lines
if (( file_lines % page_lines > 0 )); then
   let pages=pages+1
fi

print "$1 has $pages pages of text."
```

Notice that we use the integer conditional **((file_lines % page_lines > 0))** rather than the **[[** ... **]]** form.

At the heart of this code is the UNIX utility *wc(1)* , which counts the number of lines, words, and characters (bytes) in its input. By default, its output looks something like this:

```
8     34     161  bob
```

wc 's output means that the file *bob* has 8 lines, 34 words, and 161 characters. *wc* recognizes the options **-l** , **-w** , and **-c** , which tell it to print only the number of lines, words, or characters, respectively.

wc normally prints the name of its input file (given as argument). Since we want only the number of lines, we have to do two things. First, we give it input from file redirection instead, as in **wc -l < bob** instead of **wc -l bob** . This produces the number of lines preceded by a single space (which would normally separate the

filename from the number).

Unfortunately, that space complicates matters: the statement **let file_lines=$(wc -l < $1)** becomes "let file_lines= *N* " after command substitution; the space after the equal sign is an error. That leads to the second modification, the quotes around the command substitution expression. The statement **let file_lines="** *N* **"** is perfectly legal, and **let** knows how to remove the leading space.

The first **if** clause in the *pages* script checks for an option and, if it was given, strips the dash (**-**) off and assigns it to the variable **page_lines** . *wc* in the command substitution expression returns the number of lines in the file whose name is given as argument.

The next group of lines calculates the number of pages and, if there is a remainder after the division, adds 1. Finally, the appropriate message is printed.

As a bigger example of integer arithmetic, we will complete our emulation of the C shell's *pushd* and *popd* functions (Task 4-8). Remember that these functions operate on **DIRSTACK** , a stack of directories represented as a string with the directory names separated by spaces. The C shell's *pushd* and *popd* take additional types of arguments, which are:

- **pushd +n** takes the *n* th directory in the stack (starting with 0), rotates it to the top, and **cd** s to it.
- **pushd** without arguments, instead of complaining, swaps the two top directories on the stack and **cd** s to the new top.
- **popd +n** takes the *n* th directory in the stack and just deletes it.

The most useful of these features is the ability to get at the *n* th directory in the stack. Here are the latest versions of both functions:

```
function pushd { # push current directory onto stack
   dirname=$1
   if [[ -d $dirname && -x $dirname ]]; then
        cd $dirname
      DIRSTACK="$dirname ${DIRSTACK:-$PWD}"
      print "$DIRSTACK"
   else
      print "still in $PWD."
   fi
}

function popd {  # pop directory off the stack, cd to new top
   if [[ -n $DIRSTACK ]]; then
      DIRSTACK=${DIRSTACK#* }
      cd ${DIRSTACK%% *}
      print "$PWD"
   else
      print "stack empty, still in $PWD."
   fi
}
```

To get at the *n* th directory, we use a **while** loop that transfers the top directory to a temporary copy of the stack *n* times. We'll put the loop into a function called *getNdirs* that looks like this:

```
function getNdirs{
   stackfront=''
   let count=0
   while (( count < $1 )); do
      stackfront="$stackfront ${DIRSTACK%% *}"
      DIRSTACK=${DIRSTACK#* }
      let count=count+1
   done
}
```

The argument passed to *getNdirs* is the *n* in question. The variable **stackfront** is the temporary copy that will contain the first *n* directories when the loop is done. **stackfront** starts as null; **count** , which counts the number of loop iterations, starts as 0.

The first line of the loop body appends the top of the stack (**${DIRSTACK%% * }**) to **stackfront** ; the second line deletes the top from the stack. The last line increments the counter for the next iteration. The entire loop executes *N* times, for values of **count** from 0 to *N* -1.

When the loop finishes, the last directory in **$stackfront** is the *N* th directory. The expression **${stackfront## * }** extracts this directory. Furthermore, **DIRSTACK** now contains the "back" of the stack, i.e., the stack *without* the first *n* directories. With this in mind, we can now write the code for the improved versions of *pushd* and *popd* :

```
function pushd {
   if [[ $1 = ++([0-9]) ]]; then
      # case of pushd +n: rotate n-th directory to top
      let num=${1#+}
      getNdirs $num

      newtop=${stackfront##* }
      stackfront=${stackfront%$newtop}

      DIRSTACK="$newtop $stackfront $DIRSTACK"
      cd $newtop

   elif [[ -z $1 ]]; then
      # case of pushd without args; swap top two directories
      firstdir=${DIRSTACK%% *}
      DIRSTACK=${DIRSTACK#* }
      seconddir=${DIRSTACK%% *}
      DIRSTACK=${DIRSTACK#* }
      DIRSTACK="$seconddir $firstdir $DIRSTACK"
      cd $seconddir

   else
        cd $dirname
      # normal case of pushd dirname
      dirname=$1
      if [[ -d $dirname && -x $dirname ]]; then
```

```
            DIRSTACK="$dirname ${DIRSTACK:-$PWD}"
            print "$DIRSTACK"
         else
            print still in "$PWD."
         fi
      fi
   }

   function popd {      # pop directory off the stack, cd to new top
      if [[ $1 = ++([0-9]) ]]; then
         # case of popd +n: delete n-th directory from stack
         let num={$1#+}
         getNdirs $num
         stackfront=${stackfront% *}
         DIRSTACK="$stackfront $DIRSTACK"

      else
         # normal case of popd without argument
         if [[ -n $DIRSTACK ]]; then
            DIRSTACK=${DIRSTACK#* }
            cd ${DIRSTACK%% *}
            print "$PWD"
         else
            print "stack empty, still in $PWD."
         fi
      fi
   }
```

These functions have grown rather large; let's look at them in turn. The **if** at the beginning of *pushd* checks if the first argument is an option of the form **+** *N* . If so, the first body of code is run. The first **let** simply strips the plus sign (+) from the argument and assigns the result - as an integer - to the variable **num** . This, in turn, is passed to the *getNdirs* function.

The next two assignment statements set **newtop** to the *N* th directory - i.e., the last directory in **$stackfront** - and delete that directory from **stackfront** . The final two lines in this part of *pushd* put the stack back together again in the appropriate order and **cd** to the new top directory.

The **elif** clause tests for no argument, in which case *pushd* should swap the top two directories on the stack. The first four lines of this clause assign the top two directories to **firstdir** and **seconddir** , and delete these from the stack. Then, as above, the code puts the stack back together in the new order and **cd** s to the new top directory.

The **else** clause corresponds to the usual case, where the user supplies a directory name as argument.

popd works similarly. The **if** clause checks for the **+** *N* option, which in this case means delete the *N* th directory. A **let** extracts the *N* as an integer; the *getNdirs* function puts the first *n* directories into **stackfront** . Then the line **stackfront=${stackfront% *}** deletes the last directory (the *N* th directory) from **stackfront** . Finally, the stack is put back together with the *N* th directory missing.

The **else** clause covers the usual case, where the user doesn't supply an argument.

Before we leave this subject, here are a few exercises that should test your understanding of this code:

1. Add code to *pushd* that exits with an error message if the user supplies no argument and the stack contains fewer than two directories.
2. Verify that when the user specifies **+** *N* and *N* exceeds the number of directories in the stack, both *pushd* and *popd* use the last directory as the *N* th directory.
3. Modify the *getNdirs* function so that it checks for the above condition and exits with an appropriate error message if true.
4. Change *getNdirs* so that it uses *cut* (with command substitution), instead of the **while** loop, to extract the first *N* directories. This uses less code but runs more slowly because of the extra processes generated.

[Feb 14, 2017] Ms Dos style aliases for linux

I think alias ipconfig = 'ifconfig' is really useful for people who work with Linus from Windows POc desktop/laptop.

Feb 14, 2017 | bash.cyberciti.biz

```
# MS-DOS / XP cmd like stuff
  alias edit = $VISUAL
  alias copy = 'cp'
  alias cls = 'clear'
  alias del = 'rm'
  alias dir = 'ls'
  alias md = 'mkdir'
  alias move = 'mv'
  alias rd = 'rmdir'
  alias ren = 'mv'
  alias ipconfig = 'ifconfig'
```

[Feb 04, 2017] Quickly find differences between two directories

You will be surprised, but GNU diff use in Linux understands the situation when two arguments are directories and behaves accordingly

Feb 04, 2017 | www.cyberciti.biz

The diff command compare files line by line. It can also compare two directories:

```
# Compare two folders using diff ##
diff /etc /tmp/etc_old
```

Rafal Matczak September 29, 2015, 7:36 am

§ Quickly find differences between two directories
And quicker:

```
diff -y <(ls -l ${DIR1}) <(ls -l ${DIR2})
```

[Feb 04, 2017] Restoring deleted /tmp folder

Jan 13, 2015 | cyberciti.biz

As my journey continues with Linux and Unix shell, I made a few mistakes. I accidentally deleted /tmp folder. To restore it all you have to do is:

```
mkdir /tmp
chmod 1777 /tmp
chown root:root /tmp
ls -ld /tmp
```

```
mkdir /tmp chmod 1777 /tmp chown root:root /tmp ls -ld /tmp
```

[Feb 04, 2017] Use CDPATH to access frequent directories in bash - Mac OS X Hints

Feb 04, 2017 | hints.macworld.com

The variable CDPATH defines the search path for the directory containing directories. So it served much like "directories home". The dangers are in creating too complex CDPATH. Often a single directory works best. For example export CDPATH = /srv/www/public_html . Now, instead of typing cd /srv/www/public_html/CSS I can simply type: cd CSS

Use CDPATH to access frequent directories in bash
Mar 21, '05 10:01:00AM • Contributed by: jonbauman

I often find myself wanting to cd to the various directories beneath my home directory (i.e. ~/Library, ~/Music, etc.), but being lazy, I find it painful to have to type the ~/ if I'm not in my home directory already. Enter CDPATH , as desribed in man bash):

> The search path for the cd command. This is a colon-separated list of directories in which the shell looks for destination directories specified by the cd command. A sample value is ".:~:/usr".

Personally, I use the following command (either on the command line for use in just that session, or in .bash_profile for permanent use):

```
CDPATH=".:~:~/Library"
```

This way, no matter where I am in the directory tree, I can just cd dirname , and it will take me to the directory that is a subdirectory of any of the ones in the list. For example:

```
$ cd
$ cd Documents
/Users/baumanj/Documents
$ cd Pictures
/Users/username/Pictures
$ cd Preferences
/Users/username/Library/Preferences
etc...
```

[**robg adds:** No, this isn't some deeply buried treasure of OS X, but I'd never heard of the CDPATH variable, so I'm assuming it will be of interest to some other readers as well.]

cdable_vars is also nice
Authored by: clh on Mar 21, '05 08:16:26PM

Check out the bash command **shopt -s cdable_vars**

From the man bash page:

> cdable_vars
>
> If set, an argument to the cd builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.

With this set, if I give the following bash command:

export d="/Users/chap/Desktop"

I can then simply type

cd d

to change to my Desktop directory.

I put the **shopt** command and the various **export** commands in my .bashrc file.

[Feb 04, 2017] Copy file into multiple directories

Feb 04, 2017 | www.cyberciti.biz

Instead of running:

```
cp /path/to/file /usr/dir1
cp /path/to/file /var/dir2
cp /path/to/file /nas/dir3
```

Run the following command to copy file into multiple dirs:

```
echo /usr/dir1 /var/dir2 /nas/dir3 | xargs -n 1 cp -v /path/to/file
```

[Feb 04, 2017] 20 Unix Command Line Tricks – Part I

Feb 04, 2017 | www.cyberciti.biz

Locking a directory

For privacy of my data I wanted to lock down /downloads on my file server. So I ran:

```
chmod
 0000
/
downloads
```

chmod 0000 /downloads

The root user can still has access and ls and cd commands will not work. To go back:

```
chmod
 0755
/
downloads
```

chmod 0755 /downloads **Clear gibberish all over the screen**

Just type:

```
reset
```

reset **Becoming human**

Pass the -h or -H (and other options) command line option to GNU or BSD utilities to get output of command commands like ls, df, du, in human-understandable formats:

```
ls
-lh
# print sizes in human readable format (e.g., 1K 234M 2G)
df
-h
df
-k
# show output in bytes, KB, MB, or GB
free
-b
free
-k
free
-m
free
-g
# print sizes in human readable format (e.g., 1K 234M 2G)
du
-h
# get file system perms in human readable format
stat
-c
%
A
/
boot
# compare human readable numbers
sort
-h
-a
file
# display the CPU information in human readable format on a Linux

lscpu
lscpu
-e

lscpu
-e
```

```
=cpu,node
# Show the  size of each file but in a more human readable way
tree
-h
tree
-h
/
boot
```

ls -lh # print sizes in human readable format (e.g., 1K 234M 2G) df -h df -k # show output in bytes, KB, MB, or GB free -b free -k free -m free -g # print sizes in human readable format (e.g., 1K 234M 2G) du -h # get file system perms in human readable format stat -c %A /boot # compare human readable numbers sort -h -a file # display the CPU information in human readable format on a Linux lscpu lscpu -e lscpu -e=cpu,node # Show the size of each file but in a more human readable way tree -h tree -h /boot **Show information about known users in the Linux based system**

Just type:

```
## linux version ##

lslogins

## BSD version ##

logins
```

linux version ## lslogins## BSD version ## logins

Sample outputs:

```
UID USER      PWD-LOCK PWD-DENY LAST-LOGIN GECOS
  0 root           0        0   22:37:59 root
  1 bin            0        1            bin
  2 daemon         0        1            daemon
  3 adm            0        1            adm
  4 lp             0        1            lp
  5 sync           0        1            sync
  6 shutdown       0        1 2014-Dec17 shutdown
  7 halt           0        1            halt
  8 mail           0        1            mail
 10 uucp           0        1            uucp
 11 operator       0        1            operator
 12 games          0        1            games
 13 gopher         0        1            gopher
 14 ftp            0        1            FTP User
 27 mysql          0        1            MySQL Server
 38 ntp            0        1
 48 apache         0        1            Apache
 68 haldaemon      0        1            HAL daemon
 69 vcsa           0        1            virtual console memory owner
 72 tcpdump        0        1
 74 sshd           0        1            Privilege-separated SSH
 81 dbus           0        1            System message bus
 89 postfix        0        1
 99 nobody         0        1            Nobody
173 abrt           0        1
497 vnstat         0        1            vnStat user
498 nginx          0        1            nginx user
499 saslauth       0        1            "Saslauthd user"
```

Confused on a top command output?

Seriously, you need to try out htop instead of top:

```
sudo
htop
```

sudo htop **Want to run the same command again?**

Just type !! . For example:

```
/
myhome
/
dir
/
script
/
name arg1 arg2

# To run the same command again
!!


## To run the last command again as root user
sudo
!!
```

/myhome/dir/script/name arg1 arg2# To run the same command again !!## To run the last command again as root user sudo !!

The !! repeats the most recent command. To run the most recent command beginning with "foo":

```
!
foo
# Run the most recent command beginning with "service" as root
sudo
!
service
```

!foo # Run the most recent command beginning with "service" as root sudo !service

The !$ use to run command with the last argument of the most recent command:

```
# Edit nginx.conf
sudo
vi
/
etc
/
nginx
/
nginx.conf

# Test nginx.conf for errors
/
sbin
/
nginx
-t
-c
/
etc
/
nginx
/
nginx.conf

# After testing a file with "/sbin/nginx -t -c /etc/nginx/nginx.conf", you
# can edit file again with vi
sudo
vi
!
$
```

Edit nginx.conf sudo vi /etc/nginx/nginx.conf# Test nginx.conf for errors /sbin/nginx -t -c /etc/nginx/nginx.conf# After testing a file with "/sbin/nginx -t -c /etc/nginx/nginx.conf", you # can edit file again with vi sudo vi !$ **Get a reminder you when you have to leave**

If you need a reminder to leave your terminal, type the following command:

```
leave +hhmm
```

leave +hhmm

Where,

- hhmm – The time of day is in the form hhmm where hh is a time in hours (on a 12 or 24 hour clock), and mm are minutes. All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

Home sweet home

Want to go the directory you were just in? Run:
cd -
Need to quickly return to your home directory? Enter:
cd
The variable CDPATH defines the search path for the directory containing directories:

```
export
CDPATH
=
/
var
/
www:
/
nas10
```

export CDPATH=/var/www:/nas10

Now, instead of typing cd /var/www/html/ I can simply type the following to cd into /var/www/html path:

```
cd
 html
```

cd html **Editing a file being viewed with less pager**

To edit a file being viewed with less pager, press v . You will have the file for edit under $EDITOR:

```
less
*
.c
less
 foo.html
## Press v to edit file ##
## Quit from editor and you would return to the less pager again ##
```

less *.c less foo.html ## Press v to edit file ## ## Quit from editor and you would return to the less pager again ## **List all files or directories on your system**

To see all of the directories on your system, run:

```
find
/
-type
 d
|
less
```

```
# List all directories in your $HOME
find
$HOME
-type
 d
-ls
|
less
```

find / -type d | less# List all directories in your $HOME find $HOME -type d -ls | less

To see all of the files, run:

```
find
/
-type
 f
|
less
```

```
# List all files in your $HOME
find
$HOME
-type
 f
-ls
|
less
```

find / -type f | less# List all files in your $HOME find $HOME -type f -ls | less **Build directory trees in a single command**

You can create directory trees one at a time using mkdir command by passing the -p option:

```
mkdir
-p
/
jail
```

```
/
{
dev,bin,sbin,etc,usr,lib,lib64
}
ls
-l
/
jail
/
```

mkdir -p /jail/{dev,bin,sbin,etc,usr,lib,lib64} ls -l /jail/ **Copy file into multiple directories**

Instead of running:

```
cp
/
path
/
to
/
file
/
usr
/
dir1
cp
/
path
/
to
/
file
/
var
/
dir2
cp
/
path
/
to
/
file
/
nas
/
dir3
```

cp /path/to/file /usr/dir1 cp /path/to/file /var/dir2 cp /path/to/file /nas/dir3

Run the following command to copy file into multiple dirs:

```
echo
/
usr
/
dir1
/
var
/
dir2
/
nas
/
dir3
|
xargs
-n
1
cp
-v
/
path
/
to
/
file
```

echo /usr/dir1 /var/dir2 /nas/dir3 | xargs -n 1 cp -v /path/to/file

Creating a shell function is left as an exercise for the reader

Quickly find differences between two directories

The diff command compare files line by line. It can also compare two directories:

```
ls
-l
/
tmp
/
r
ls
-l
/
tmp
/
s
# Compare two folders using diff ##
diff
/
tmp
/
r
/
/
tmp
/
s
/
```

[Feb 04, 2017] List all files or directories on your system

Feb 04, 2017 | www.cyberciti.biz

List all files or directories on your system

To see all of the directories on your system, run:

```
find
/
-type
 d
|
less


# List all directories in your $HOME
find
$HOME
-type
 d
-ls
|
less
```

find / -type d | less# List all directories in your $HOME find $HOME -type d -ls | less

To see all of the files, run:

```
find
/
-type
 f
|
less


# List all files in your $HOME
find
$HOME
-type
 f
-ls
|
less
```

basic ~-.bashrc ~-.bash_profile tips thread

Arch Linux Forums

I added some comments explaining each piece.

Misc stuff:

```
# My prompt, quite basic, decent coloring, shows the value of $?
# (exit value of last command, useful sometimes):
C_DEFAULT="[33[0m]"
C_BLUE="[33[0;34m]"
export PS1="$C_BLUE($C_DEFAULT$?$C_BLUE)[$C_DEFAULTu$C_BLUE@$C_DEFAULTh$C_BLUE:$C_DEFAULTw$C_BLUE]\$ $C_DEFAU
export PS2="$C_BLUE> $C_DEFAULT"

# If you allow Ctrl+Alt+Backspace to kill the X server but are paranoid,
# then this alias will ensure that there will be no shell open afterwards.
alias startx="exec startx"

# Let grep colorize the search results
alias g="egrep --color=always"
alias gi="egrep -i --color=always"

# Hostname appended to bash history filename
export HISTFILE="$HOME/.bash_history_`hostname -s`"

# Don't save repeated commands in bash history
export HISTCONTROL="ignoredups"

# Confirm before overwriting something
alias cp="cp -i"

# Disable ^S/^Q flow control (does anyone like/use this at all?)
stty -ixon

# If your resolution gets fucked up, use this to reset (requires XRandR)
alias resreset="xrandr --size 1280x1024"
```

And some small but handy functions:

```
# mkmv - creates a new directory and moves the file into it, in 1 step
# Usage: mkmv <file> <directory>
mkmv() {
   mkdir "$2"
   mv "$1" "$2"
}

# sanitize - set file/directory owner and permissions to normal values (644/755)
# Usage: sanitize <file>
sanitize() {
   chmod -R u=rwX,go=rX "$@"
   chown -R ${USER}.users "$@"
}

# nh - run command detached from terminal and without output
# Usage: nh <command>
nh() {
   nohup "$@" &>/dev/null &
}

# run - compile a simple c or cpp file, run the program, afterwards delete it
# Usage: run <file> [params]
run() {
   filename="${1%%.*}"
   extension="${1##*.}"
   file="$1"
   shift
   params="$@"
   command=""

   if [ $extension = "cc" -o $extension = "cpp" -o $extension = "c++" ]; then
      command="g++"
   elif [ $extension = "c" ]; then
      command="gcc"
   else
      echo "Invalid file extension!"
      return 1
   fi

   $command -Wall -o $filename $file
   chmod a+x $filename
   ./$filename $params
   rm -f $filename 2>/dev/null
}
```

Offline

```
... ... ...

function mktar() { tar czf "${1%%/}.tar.gz" "${1%%/}/"; }

function mkmine() { sudo chown -R ${USER} ${1:-.}; }

alias svim='sudo vim'

# mkmv - creates a new directory and moves the file into it, in 1 step
# Usage: mkmv <file> <directory>
```

```
mkmv() {
mkdir "$2"
mv "$1" "$2"
}

# sanitize - set file/directory owner and permissions to normal values (644/755)
# Usage: sanitize <file>
sanitize() {
chmod -R u=rwX,go=rX "$@"
chown -R ${USER}.users "$@"
}

# nh - run command detached from terminal and without output
# Usage: nh <command>
nh() {
nohup "$@" &>/dev/null &
}

alias un='tar -zxvf'
alias mountedinfo='df -hT'
alias ping='ping -c 10'
alias openports='netstat -nape --inet'
alias ns='netstat -alnp --protocol=inet | grep -v CLOSE_WAIT | cut
-c-6,21-94 | tail +2'
alias du1='du -h --max-depth=1'
alias da='date "+%Y-%m-%d %A %T %Z"'
alias ebrc='pico ~/.bashrc'

# Alias to multiple ls commands
alias la='ls -Al' # show hidden files
alias ls='ls -aF --color=always' # add colors and file type extensions
alias lx='ls -lXB' # sort by extension
alias lk='ls -lSr' # sort by size
alias lc='ls -lcr' # sort by change time
alias lu='ls -lur' # sort by access time
alias lr='ls -lR' # recursive ls
alias lt='ls -ltr' # sort by date
alias lm='ls -al |more' # pipe through 'more'

# Alias chmod commands
alias mx='chmod a+x'
alias 000='chmod 000'
alias 644='chmod 644'
alias 755='chmod 755'
```

What are some useful Bash tricks - Quora

Gaurav Gada, Master's Information Management, University of Washington Information School (2018)

Written Dec 19, 2011

sudo !!
For when you forget to add sudo to your commands.

I got to know about this, among others at this beautiful website:
http://www.commandlinefu.com/com...

Mattias Jansson, I like cats

Written Jan 6, 2013

Some things I used to use often... and not so often:

o Comment the line you're currently on (Esc-#).

o Send stuff to a host/port using bash builtins- echo foo > /dev/tcp/host/port. For example, quick and dirty file transfer from a minimal linux install to some place with nc installed:
On destination machine: nc -l 7070 > newfile
On source machine: cat somefile > /dev/tcp/somehostname/7070

o C-x C-e to edit current line in your $EDITOR (all readline-enabled programs have this- I really needed this often when writing an SQL query which ended up being very long)

o I've got this simple shell function to take a config file (which uses the hash as a comment initiator) and dump all the contents which do not start with a comment or whitespace:

```
unc ()
{
grep -vE "^[ ]*\#" $1 | grep .
}
```

o A tiny no-nonsense webserver to share the directory you're standing in:

```
alias webshare='python -c "import SimpleHTTPServer;SimpleHTTPServer.test()"'
```

Fred Cirera, works at Twitter

Written Jul 30, 2014

I wouldn't classify the following as tricks but things that every developer writing a bash script should know.

Every shell script should start with set -o nounset and set -o errexit

nounset means that using a variable that is not set will raise an error. In the following example if the bash script is called without argument all the files in /var/log will be delete.

1. #!/bin/bash

```
2. set -o nounset
3. CONTAINER_ROOT=$1
4. ...
5. rm $CONTAINER_ROOT/var/log/*
```

errexit when this options is set the the bash scrip will exit is a command fail. In the following example if the directory /bigdisk/temp doesn't exist mktemp will fail but the script will continue and call generate_big_data with no $TEMPFILE.

```
1. #!/bin/bash
2. set -o errexit
3. TEMPDIR=/bigdisk/temp
4. TEMPFILE=$(mktemp /$TEMPDIR/app.XXXXXX)
5. generate_big_data -out $TEMPFILE
```

In that previews case you can write TEMPFILE=$(mktemp /$TEMPDIR/app.XXXXXX) || exit 1 but it is always good to exist on error in order to be sure they will be no surprising side effect when a command called in the middle of your script fail.

Variables substitution

When you don string manipulation use variables substitution. This is faster and save from doing useless forks.

Stop writing things like that: FILENAME=`basename $1` instead write FILENAME=${1##*/} or DIRNAME=${1%/*} instead of DIRNAME=`dirname $1`. You'll find more information on variables substitution in the bash manual in the paragraph Manipulating Strings.

Nick Shelly, Stanford CS PhD candid, Apple, Air Force capt, Rhodes scholar

Written Aug 20, 2012

Ctrl+R to reverse search through your Bash history. Ctrl+R again keeps searching, Ctrl+G cancels the search.

Though GNU's Readline package is not unique to Bash (Python's interactive shell has this as well), reverse search is one of the most useful aspects of command line shells over GUIs.

Dan Fango, www.danfango.co.uk

Written Apr 21, 2015

A couple I haven't seen (or missed) in the previous answers:

Alt+. : brings back the last word from the previous line. If your previous line was "ls somefile.txt" then "vi Alt+." will translate to "vi somefile.txt". Hitting Alt+.multiple times will cycle back through your history

Alt+# : translates to adding # (comment) to the start of your current command line and hitting return

Jianing Yang, Linux system administrator

Written Dec 19, 2011

How about C-x C-e to open your favorite editor for editing the current command line.

Steven Lehar

Written Feb 27, 2014

When I cd to some/long/path, then I type
$ here=`pwd`
then I cd back/to/some/other/path, then for example
$ there=`pwd`
Now I can do stuff like...
$ cd $here
$ cp file.txt $there

Chris Rutherford, 20 years of unix admin

Written Aug 15, 2012

Check out this guys stuff, best bash script tricks Ive seen in my 20 years of scripting http://www.catonmat.net/blog/bas...

Michael Rinus, every day basher

Written Jun 5, 2015

I strongly recommend spending some time at http://www.commandlinefu.com/com...

There is some quite awesome and helpful stuff out there :)

What are some useful .bash_profile and .bashrc tips - Quora

```
function cl(){ cd "$@" && la; }

function cdn(){ for i in `seq $1`; do cd ..; done;}

PROMPT_COMMAND="${PROMPT_COMMAND:+$PROMPT_COMMAND ; }"'echo `dt` `pwd` $$ $USER "$(history 1)" >>
~/.bash_eternal_history'

if [ -f /etc/bash_completion ]; then
. /etc/bash_completion
fi
alias 'dus=du -sckx * | sort -nr' #directories sorted by size
alias lsdirs="ls -l | grep '^d'"
```

Gaurav Gada, Master's Information Management, University of Washington Information School (2018)

Written Dec 17, 2011

I have these lines:
1shopt -s histappend
2PROMPT_COMMAND="history -n; history -a"
3unset HISTFILESIZE
4HISTSIZE=2000

The first 2 lines keep the history between multiple bash sessions synced and the last two increase the history size from the default 500.

Yaniv Ng, researcher, atheist, ex-gamer, terminalist vimmer

Written Oct 17, 2014

This is something I found very useful when working with multiple terminals on different directories. Sometimes the new terminal opens in the home directory instead of the current working directory (depending on the terminal program).

Use **gg** in the terminal where you want to go. Then go to the new terminal and use **hh**.

```
1. gg() { pwd > /tmp/last_path; }
2. hh() { cd $(cat /tmp/last_path); }
```

```
1. # Easy extract
2. extract () {
3. if [ -f $1 ] ; then
4. case $1 in
5. *.tar.bz2) tar xvjf $1 ;;
6. *.tar.gz) tar xvzf $1 ;;
7. *.bz2) bunzip2 $1 ;;
8. *.rar) rar x $1 ;;
9. *.gz) gunzip $1 ;;
10. *.tar) tar xvf $1 ;;
11. *.tbz2) tar xvjf $1 ;;
12. *.tgz) tar xvzf $1 ;;
13. *.zip) unzip $1 ;;
14. *.Z) uncompress $1 ;;
15. *.7z) 7z x $1 ;;
16. *) echo "don't know how to extract '$1'..." ;;
17. esac
18. else
19. echo "'$1' is not a valid file!"
20. fi
21. }
```

```
alias top-commands='history | awk "{print $2}" | awk "{print $1}" |sort|uniq
```

Ch Huang

Written Feb 8, 2011

When bash is invoked as an interactive login shell, or as a non-interactive shell with the --login option, it first reads and executes commands from the file /etc/profile, if that file exists. After reading that file, it looks for ~/.bash_profile, ~/.bash_login, and ~/.profile, in that order, and reads and executes commands from the first one that exists and is readable.

```
#in case you rm a file by mistake
alias rm=safe_rm

safe_rm () {
local d t f s

[ -z "$PS1" ] && (/bin/rm "$@"; return)

d="${TRASH_DIR:=$HOME/.__trash}/`date +%W`"
t=`date +%F_%H-%M-%S`
[ -e "$d" ] || mkdir -p "$d" || return

for f do
[ -e "$f" ] || continue
s=`basename "$f"`
/bin/mv "$f" "$d/${t}_$s" || break
done

echo -e "[$? $t `whoami` `pwd`]$@\n" >> "$d/00rmlog.txt"
}
```

Akhil Ravidas

Written Feb 10, 2013

```
1. alias Cd='cd -'
```

My Favorite bash Tips and Tricks Linux Journal

However, you can use spaces if they're enclosed in quotes outside the braces or within an item in the comma-separated list:

```
$ echo {"one ","two ","red ","blue "}fish
one fish two fish red fish blue fish

$ echo {one,two,red,blue}" fish"
one fish two fish red fish blue fish
```

You also can nest braces, but you must use some caution here too:

```
$ echo {{1,2,3},1,2,3}
1 2 3 1 2 3

$ echo {{1,2,3}1,2,3}
11 21 31 2 3
```

[Dec 19, 2016] Unknown Bash Tips and Tricks For Linux Linux.com The source for Linux information

The type command looks a lot like the command builtin, but it does more:

$ type ll
ll is aliased to `ls -alF'

$ type -t grep
alias

Bash Functions

Run declare -F to see a list of Bash's builtin function names. declare -f prints out the complete functions, and declare -f [function-name] prints the named function. type won't find list functions, but once you know a function name it will also print it:

```
$ type quote
quote is a function
quote ()
{
  echo \'${1//\'/\'\\\'\'}\'
}
```

This even works for your own functions that you create, like this simple example testfunc that does one thing: changes to the /etc directory:

```
$ function testfunc
> {
> cd /etc
> }
```

Now you can use declare and type to list and view your new function just like the builtins.

[Dec 19, 2016] Bash Tricks " Linux Magazine

Graham Nicholls • 2 years ago

Oh FGS alias rm="rm -i" what a crock. I have _never_ needed this. Unix/Linux is expert friendly, not fool friendly. Possibly useful if you're root, otherwise just an incredible irritant.

OTOH, I think that history time-stamping should be the default. So useful for auditing, and for "I know I did something the other day" stuff. I use '%c' for my HISTTIMEFORMAT.

marnixava > Graham Nicholls • 2 years ago

I fully agree that the 'rm="rm -i"' alias and similar aliases are irritating. I think it might also lull newcomers into a false sense of security that it's pretty safe to do that command. One day they might be on a system without such an alias. It's good to learn early on that "rm" means it's going to be removed, no ifs or buts. One needs to make a habit of reviewing the command line before hitting enter.

Graham Nicholls > marnixava • 2 years ago

That's a really good point, which I'd not considered.

John Lockard • 2 years ago

The "HISTIGNORE" is interesting for other purposes, but the option for ignoring commands which start with space is actually a setting in bash using "export HISTCONTROL=ignorespace". If you want to eliminate duplicate entries you can use "ignoredups" or "erasedups". "ignoreboth" does both "ignoredups" and "ignorespace".

Ryan • 2 years ago

I like that using chattr -a is mentioned as a possible security fix for .bash_history, when the next talked about item is HISTIGNORE and someone could just export HISTIGNORE="*" and it doesnt matter if .bash_alias is append only. The commands are not logged in the first place to be deleted later.

edit: But good post overall. enjoyed it :)

marnixava > Ryan • 2 years ago

Even if the history file is chattr'ed to append-only mode, wouldn't the user still be able to simply remove that history file? IMHO there are too many workarounds for a determined user to make it worthwhile except perhaps if used only as a gentle reminder that we'd like not to alter the history file.

Linux secrets most users don't know about ITworld

J1r1k: "Alt + . (dot) in bash. Last argument of previous command. It took me few years to discover this."

[Dec 06, 2015] Bash For Loop Examples

A very nice tutorial by Vivek Gite (created October 31, 2008 last updated June 24, 2015). His mistake is putting new for loop too far inside the tutorial. It should emphazied, not hidden.

June 24, 2015 | cyberciti.biz

... ...

Bash v4.0+ has inbuilt support for setting up a step value using {START..END..INCREMENT} syntax:

```
#!/bin/bash
echo "Bash version ${BASH_VERSION}..."
for i in {0..10..2}
  do
    echo "Welcome $i times"
 done
```

Sample outputs:

```
Bash version 4.0.33(0)-release...
Welcome 0 times
Welcome 2 times
Welcome 4 times
Welcome 6 times
Welcome 8 times
Welcome 10 times
```

... ...

Three-expression bash for loops syntax

This type of for loop share a common heritage with the C programming language. It is characterized by a three-parameter loop control expression; consisting of an initializer (EXP1), a loop-test or condition (EXP2), and a counting expression (EXP3).

```
for (( EXP1; EXP2; EXP3 ))
do
      command1
      command2
      command3
done
```

A representative three-expression example in bash as follows:

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
  echo "Welcome $c times"
done
```

... ...

Jadu Saikia, November 2, 2008, 3:37 pm

Nice one. All the examples are explained well, thanks Vivek.

seq 1 2 20
output can also be produced using jot

jot – 1 20 2

The infinite loops as everyone knows have the following alternatives.

while(true)
or
while :

//Jadu

Andi Reinbrech, November 18, 2010, 7:42 pm

I know this is an ancient thread, but thought this trick might be helpful to someone:

For the above example with all the cuts, simply do

```
set `echo $line`
```

This will split line into positional parameters and you can after the set simply say

```
F1=$1; F2=$2; F3=$3
```

I used this a lot many years ago on solaris with "set `date`", it neatly splits the whole date string into variables and saves lots of messy cutting :-)

… no, you can't change the FS, if it's not space, you can't use this method

Peko, July 16, 2009, 6:11 pm

Hi Vivek,
Thanks for this a useful topic.

IMNSHO, there may be something to modify here
==========
Latest bash version 3.0+ has inbuilt support for setting up a step value:

#!/bin/bash
for i in {1..5}
==========
1) The increment feature seems to belong to the version 4 of bash.
Reference: http://bash-hackers.org/wiki/doku.php/syntax/expansion/brace
Accordingly, my bash v3.2 does not include this feature.

BTW, where did you read that it was 3.0+ ?
(I ask because you may know some good website of interest on the subject).

2) The syntax is {from..to..step} where from, to, step are 3 integers.
You code is missing the increment.

Note that GNU Bash documentation may be bugged at this time,
because on GNU Bash manual, you will find the syntax {x..y[incr]}
which may be a typo. (missing the second ".." between y and increment).

see http://www.gnu.org/software/bash/manual/bashref.html#Brace-Expansion

The Bash Hackers page
again, see http://bash-hackers.org/wiki/doku.php/syntax/expansion/brace
seeems to be more accurate,
but who knows ? Anyway, at least one of them may be right… ;-)

Keep on the good work of your own,
Thanks a million.

- Peko

Michal Kaut July 22, 2009, 6:12 am

Hello,

is there a simple way to control the number formatting? I use several computers, some of which have non-US settings with comma as a decimal point. This means that
`for x in $(seq 0 0.1 1)` gives 0 0.1 0.2 … 1 one some machines and 0 0,1 0,2 … 1 on other.
Is there a way to force the first variant, regardless of the language settings? Can I, for example, set the keyboard to US inside the script? Or perhaps some alternative to `$x` that would convert commas to points?
(I am sending these as parameters to another code and it won't accept numbers with commas…)

The best thing I could think of is adding `x=`echo $x | sed s/,/./`` as a first line inside the loop, but there should be a better solution? (Interestingly, the sed command does not seem to be upset by me rewriting its variable.)

Thanks,
Michal

Peko July 22, 2009, 7:27 am

To Michal Kaut:

Hi Michal,

Such output format is configured through LOCALE settings.

I tried :

export LC_CTYPE="en_EN.UTF-8″; seq 0 0.1 1

and it works as desired.

You just have to find the exact value for LC_CTYPE that fits to your systems and your needs.

Peko

Peko July 22, 2009, 2:29 pm

To Michal Kaus [2]

Ooops – ;-)
Instead of LC_CTYPE,
LC_NUMERIC should be more appropriate
(Although LC_CTYPE is actually yielding to the same result – I tested both)

By the way, Vivek has already documented the matter : http://www.cyberciti.biz/tips/linux-find-supportable-character-sets.html

Philippe Petrinko October 30, 2009, 8:35 am

To Vivek:
Regarding your last example, that is : running a loop through arguments given to the script on the command line, there is a simplier way of doing this:
instead of:
FILES="$@"
for f in $FILES

use the following syntax
for arg
do
whatever you need here – try : echo "$arg"
done

Of course, you can use any variable name, not only "arg".

Philippe Petrinko November 11, 2009, 11:25 am

To tdurden:

Why would'nt you use

1) either a [for] loop
for old in * ; do mv ${old} ${old}.new; done

2) Either the [rename] command ?
excerpt form "man rename" :

RENAME(1) Perl Programmers Reference Guide RENAME(1)

NAME
rename – renames multiple files

SYNOPSIS
rename [-v] [-n] [-f] perlexpr [files]

DESCRIPTION
"rename" renames the filenames supplied according to the rule specified as the first argument. The perlexpr argument is a Perl expression which is expected to modify the $_ string in Perl for at least some of the filenames specified. If a given filename is not modified by the expression, it will not be renamed. If no filenames are given on the command line, filenames will be read via standard input.

For example, to rename all files matching "*.bak" to strip the extension, you might say

rename 's/\.bak$//' *.bak

To translate uppercase names to lower, you'd use

rename 'y/A-Z/a-z/' *

- Philippe

Philippe Petrinko November 11, 2009, 9:27 pm

If you set the shell option extglob, Bash understands some more powerful patterns. Here, a is one or more pattern, separated by the pipe-symbol (|).

?() Matches zero or one occurrence of the given patterns
*() Matches zero or more occurrences of the given patterns
+() Matches one or more occurrences of the given patterns
@() Matches one of the given patterns
!() Matches anything except one of the given patterns

source: http://www.bash-hackers.org/wiki/doku.php/syntax/pattern

Philippe Petrinko November 12, 2009, 3:44 pm

To Sean:
Right, the more sharp a knife is, the easier it can cut your fingers…

I mean: There are side-effects to the use of file globbing (like in [for f in *]) , when the globbing expression matches nothing: the globbing expression is not susbtitued.

Then you might want to consider using [nullglob] shell extension,
to prevent this.
see: http://www.bash-hackers.org/wiki/doku.php/syntax/expansion/globs#customization

Devil hides in detail ;-)

Dominic January 14, 2010, 10:04 am

There is an interesting difference between the exit value for two different for looping structures (hope this comes out right):
```
for (( c=1; c<=2; c++ )) do echo -n "inside (( )) loop c is $c, "; done; echo "done (( )) loop c is $c"
for c in {1..2}; do echo -n "inside { } loop c is $c, "; done; echo "done { } loop c is $c"
```
You see that the first structure does a final increment of c, the second does not. The first is more useful IMO because if you have a conditional break in the for loop, then you can subsequently test the value of $c to see if the for loop was broken or not; with the second structure you can't know whether the loop was broken on the last iteration or continued to completion.

Dominic January 14, 2010, 10:09 am

sorry, my previous post would have been clearer if I had shown the output of my code snippet, which is:
```
inside (( )) loop c is 1, inside (( )) loop c is 2, done (( )) loop c is 3
inside { } loop c is 1, inside { } loop c is 2, done { } loop c is 2
```

Philippe Petrinko March 9, 2010, 2:34 pm

@Dmitry

And, again, as stated many times up there, using [seq] is counter productive, because it requires a call to an external program, when you should Keep It Short and Simple, using only bash internals functions:

```
for ((c=1; c<21; c+=2)); do echo "Welcome $c times" ; done
```

(and I wonder why Vivek is sticking to that old solution which should be presented only for historical reasons when there was no way of using bash internals.
By the way, this historical recall should be placed only at topic end, and not on top of the topic, which makes newbies sticking to the not-up-to-date technique ;-))

Sean March 9, 2010, 11:15 pm

I have a comment to add about using the builtin **for ((…))** syntax. I would agree the builtin method is cleaner, but from what I've noticed with other builtin functionality, I had to check the speed advantage for myself. I wrote the following files:

builtin_count.sh:

```
#!/bin/bash
for ((i=1;i<=1000000;i++))
do
echo "Output $i"
done
```

seq_count.sh:

```
#!/bin/bash
for i in $(seq 1 1000000)
do
echo "Output $i"
done
```

And here were the results that I got:
time ./builtin_count.sh
real 0m22.122s
user 0m18.329s
sys 0m3.166s

time ./seq_count.sh
real 0m19.590s
user 0m15.326s
sys 0m2.503s

The performance increase isn't too significant, especially when you are probably going to be doing something a little more interesting inside of the for loop, but it does show that builtin commands are not necessarily faster.

Andi Reinbrech November 18, 2010, 8:35 pm

The reason why the external seq is faster, is because it is executed only once, and returns a huge splurb of space separated integers which need no further processing, apart from the for loop advancing to the next one for the variable substitution.

The internal loop is a nice and clean/readable construct, but it has a lot of overhead. The check expression is re-evaluated on every iteration, and a variable on the interpreter's heap gets incremented, possibly checked for overflow etc. etc.

Note that the check expression cannot be simplified or internally optimised by the interpreter because the value may change inside the loop's body (yes, there are cases where you'd want to do this, however rare and stupid they may seem), hence the variables are volatile and get re-evaluted.

I.e. botom line, the internal one has more overhead, the "seq" version is equivalent to either having 1000000 integers inside the script (hard coded), or reading once from a text file with 1000000 integers with a cat. Point being that it gets executed only once and becomes static.

OK, blah blah fishpaste, past my bed time :-)

Cheers,
Andi

Anthony Thyssen June 4, 2010, 6:53 am

The {1..10} syntax is pretty useful as you can use a variable with it!

```
limit=10
echo {1..${limit}}
{1..10}
```

You need to eval it to get it to work!

```
limit=10
eval "echo {1..${limit}}"
1 2 3 4 5 6 7 8 9 10
```

'seq' is not avilable on ALL system (MacOSX for example)
and BASH is not available on all systems either.

You are better off either using the old while-expr method for computer compatiblity!

```
limit=10; n=1;
while [ $n -le 10 ]; do
  echo $n;
  n=`expr $n + 1`;
done
```

Alternatively use a seq() function replacement…

```
# seq_count 10
seq_count() {
  i=1; while [ $i -le $1 ]; do echo $i; i=`expr $i + 1`; done
}
# simple_seq 1 2 10
simple_seq() {
  i=$1; while [ $i -le $3 ]; do echo $i; i=`expr $i + $2`; done
}
seq_integer() {
    if [ "X$1" = "X-f" ]
    then format="$2"; shift; shift
    else format="%d"
    fi
    case $# in
    1) i=1 inc=1 end=$1 ;;
    2) i=$1 inc=1 end=$2 ;;
    *) i=$1 inc=$2 end=$3 ;;
    esac
    while [ $i -le $end ]; do
      printf "$format\n" $i;
      i=`expr $i + $inc`;
    done
  }
```

Edited: by Admin – added code tags.

TheBonsai June 4, 2010, 9:57 am

The Bash C-style for loop was taken from KSH93, thus I guess it's at least portable towards Korn and Z.

The seq-function above could use i=$((i + inc)), if only POSIX matters. expr is obsolete for those things, even in POSIX.

Philippe Petrinko June 4, 2010, 10:15 am

Right Bonsai,
(http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html#tag_02_06_04)

But FOR C-style does not seem to be POSIXLY-correct…

Read on-line reference issue 6/2004,
Top is here, http://www.opengroup.org/onlinepubs/009695399/mindex.html

and the Shell and Utilities volume (XCU) T.OC. is here
http://www.opengroup.org/onlinepubs/009695399/utilities/toc.html

doc is:
http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap01.html

and FOR command:
http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html#tag_02_09_04_03

Anthony Thyssen June 6, 2010, 7:18 am

TheBonsai wrote…. "The seq-function above could use i=$((i + inc)), if only POSIX matters. expr is obsolete for those things, even in POSIX."

I am not certain it is in Posix. It was NOT part of the original Bourne Shell, and on some machines, I deal with Bourne Shell. Not Ksh, Bash, or anything else.

Bourne Shell syntax works everywhere! But as 'expr' is a builtin in more modern shells, then it is not a big loss or slow down.

This is especially important if writing a replacement command, such as for "seq" where you want your "just-paste-it-in" function to work as widely as possible.

I have been shell programming pretty well all the time since 1988, so I know what I am talking about! Believe me.

MacOSX has in this regard been the worse, and a very big backward step in UNIX compatibility. 2 year after it came out, its shell still did not even understand most of the normal 'test' functions. A major pain to write shells scripts that need to also work on this system.

TheBonsai June 6, 2010, 12:35 pm

Yea, the question was if it's POSIX, not if it's 100% portable (which is a difference). The POSIX base more or less is a subset of the Korn features (88, 93), pure Bourne is something "else", I know. Real portability, which means a program can go wherever UNIX went, only in C ;)

Philippe Petrinko November 22, 2010, 8:23 am

And if you want to get rid of double-quotes, use:

one-liner code:

```
while read; do record=${REPLY}; echo ${record}|while read -d ","; do field="${REPLY#\"}"; field="${field%\"}"; echo ${field}; done;
done<data
```

script code, added of some text to better see record and field breakdown:

```
#!/bin/bash
while read
do
echo "New record"
record=${REPLY}
echo ${record}|while read -d ,
do
field="${REPLY#\"}"
field="${field%\"}"
echo "Field is :${field}:"
done
done<data
```

Does it work with your data?

- PP

Philippe Petrinko November 22, 2010, 9:01 am

Of course, all the above code was assuming that your CSV file is named "data".

If you want to use anyname with the script, replace:

```
done<data
```

With:

```
done
```

And then use your script file (named for instance "myScript") with standard input redirection:

```
myScript < anyFileNameYouWant
```

Enjoy!

Philippe Petrinko November 22, 2010, 11:28 am

well no there is a bug, last field of each record is not read – it needs a workout and may be IFS modification ! After all that's what it was built for… :O)

Anthony Thyssen November 22, 2010, 11:31 pm

Another bug is the inner loop is a pipeline, so you can't assign variables for use later in the script. but you can use '<<<' to break the pipeline and avoid the echo.

But this does not help when you have commas within the quotes! Which is why you needed quotes in the first place.

In any case It is a little off topic. Perhaps a new thread for reading CVS files in shell should be created.

Philippe Petrinko November 24, 2010, 6:29 pm

Anthony,
Would you try this one-liner script on your CSV file?

This one-liner assumes that CSV file named [data] has __every__ field double-quoted.

```
while read; do r="${REPLY#\"}";echo "${r/\",\"/\"}"|while read -d \";do echo "Field is :${REPLY}:";done;done<data
```

Here is the same code, but for a script file, not a one-liner tweak.

```
#!/bin/bash
# script csv01.sh
#
```

```
# 1) Usage
# This script reads from standard input
# any CSV with double-quoted data fields
# and breaks down each field on standard output
#
# 2) Within each record (line), _every_ field MUST:
# - Be surrounded by double quotes,
# - and be separated from preceeding field by a comma
# (not the first field of course, no comma before the first field)
#
while read
do
echo "New record" # this is not mandatory-just for explanation
#
#
# store REPLY and remove opening double quote
record="${REPLY#\"}"
#
#
# replace every "," by a single double quote
record=${record/\",\"/\"}
#
#
echo ${record}|while read -d \"
do
# store REPLY into variable "field"
field="${REPLY}"
#
#
echo "Field is :${field}:" # just for explanation
done
done
```

This script named here [cvs01.sh] must be used so:

```
cvs01.sh < my-cvs-file-with-doublequotes
```

Philippe Petrinko November 24, 2010, 6:35 pm

@Anthony,

By the way, using [REPLY] in the outer loop _and_ the inner loop is not a bug.
As long as you know what you do, this is not problem, you just have to store [REPLY] value conveniently, as this script shows.

TheBonsai March 8, 2011, 6:26 am

```
for ((i=1; i<=20; i++)); do printf "%02d\n" "$i"; done
```

nixCraft March 8, 2011, 6:37 am

+1 for printf due to portability, but you can use bashy .. syntax too

```
for i in {01..20}; do echo "$i"; done
```

TheBonsai March 8, 2011, 6:48 am

Well, it isn't portable per se, it makes it portable to pre-4 Bash versions.

I think a more or less "portable" (in terms of POSIX, at least) code would be

```
i=0
while [ "$((i >= 20))" -eq 0 ]; do
  printf "%02d\n" "$i"
  i=$((i+1))
done
```

Philip Ratzsch April 20, 2011, 5:53 am

I didn't see this in the article or any of the comments so I thought I'd share. While this is a contrived example, I find that nesting two groups can help squeeze a two-liner (once for each range) into a one-liner:

for num in {{1..10},{15..20}};do echo $num;done

Great reference article!

Philippe Petrinko April 20, 2011, 8:23 am

@Philip
Nice thing to think of, using brace nesting, thanks for sharing.

Philippe Petrinko May 6, 2011, 10:13 am

Hello Sanya,

That would be because brace expansion does not support variables. I have to check this.
Anyway, Keep It Short and Simple: (KISS) here is a simple solution I already gave above:

xstart=1;xend=10;xstep=1
for ((x = $xstart; x <= $xend; x += $xstep)); do echo $x;done

Actually, POSIX compliance allows to forget $ in for quotes, as said before, you could also write:

xstart=1;xend=10;xstep=1
for ((x = xstart; x <= xend; x += xstep)); do echo $x;done

Philippe Petrinko May 6, 2011, 10:48 am

Sanya,

Actually brace expansion happens __before__ $ parameter exapansion, so you cannot use it this way.

Nevertheless, you could overcome this this way:

max=10; for i in $(eval echo {1..$max}); do echo $i; done

Sanya May 6, 2011, 11:42 am

Hello, Philippe

Thanks for your suggestions
You basically confirmed my findings, that bash constructions are not as simple as zsh ones.
But since I don't care about POSIX compliance, and want to keep my scripts "readable" for less experienced people, I would prefer to stick to zsh where my simple for-loop works

Cheers, Sanya

Philippe Petrinko May 6, 2011, 12:07 pm

Sanya,

First, you got it wrong: solutions I gave are not related to POSIX, I just pointed out that POSIX allows not to use $ in for (( )), which is just a little bit more readable – sort of.

Second, why do you see this less readable than your [zsh] [for loop]?

for ((x = start; x <= end; x += step)) do
echo "Loop number ${x}"
done

It is clear that it is a loop, loop increments and limits are clear.

IMNSHO, if anyone cannot read this right, he should not be allowed to code. :-D

BFN

Anthony Thyssen May 8, 2011, 11:30 pm

If you are going to do… $(eval echo {1..$max});
You may as well use "seq" or one of the many other forms.
See all the other comments on doing for loops.

Tom P May 19, 2011, 12:16 pm

I am trying to use the variable I set in the for line on to set another variable with a different extension. Couldn't get this to work and couldnt find it anywhere on the web… Can someone help.

Example:

```
FILE_TOKEN=`cat /tmp/All_Tokens.txt`
for token in $FILE_TOKEN
do
A1_$token=`grep $A1_token /file/path/file.txt | cut -d ":" -f2`
```

my goal is to take the values from the ALL Tokens file and set a new variable with A1_ infront of it… This tells be that A1_ is not a command…

[Nov 08, 2015] Get timestamps on Bash's History

nickgeoghegan.net

One of the annoyances of Bash, is that searching through your history has no context. When did I last run that command? What commands were run at 3am, while on the lock?

The following, single line, run in the shell, will provide date and time stamping for your Bash History the next time you login, or run bash.

```
echo  'export HISTTIMEFORMAT="%h/%d - %H:%M:%S "' >>  ~/.bashrc
```

[May 08, 2014] 25 Even More – Sick Linux Commands UrFix's Blog

6) Display a cool clock on your terminal

```
watch -t -n1 "date +%T|figlet"
```

This command displays a clock on your terminal which updates the time every second. Press Ctrl-C to exit.

A couple of variants:

A little bit bigger text:

watch -t -n1 "date +%T|figlet -f big"You can try other figlet fonts, too.

Big sideways characters:

watch -n 1 -t '/usr/games/banner -w 30 $(date +%M:%S)'This requires a particular version of banner and a 40-line terminal or you can adjust the width ("30" here).

7) intercept stdout/stderr of another process

```
strace -ff -e trace=write -e write=1,2 -p SOME_PID
```

8) Remove duplicate entries in a file without sorting.

```
awk '!x[$0]++' <file>
```

Using awk, find duplicates in a file without sorting, which reorders the contents. awk will not reorder them, and still find and remove duplicates which you can then redirect into another file.

9) Record a screencast and convert it to an mpeg

```
ffmpeg -f x11grab -r 25 -s 800x600 -i :0.0 /tmp/outputFile.mpg
```

Grab X11 input and create an MPEG at 25 fps with the resolution 800×600

10) Mount a .iso file in UNIX/Linux

```
mount /path/to/file.iso /mnt/cdrom -oloop
```

"-o loop" lets you use a file as a block device

11) Insert the last command without the last argument (bash)

```
!:-
```

/usr/sbin/ab2 -f TLS1 -S -n 1000 -c 100 -t 2 http://www.google.com/then

!:- http://www.urfix.com/is the same as

/usr/sbin/ab2 -f TLS1 -S -n 1000 -c 100 -t 2 http://www.urfix.com/

12) Convert seconds to human-readable format

```
date -d@1234567890
```

This example, for example, produces the output, "Fri Feb 13 15:26:30 EST 2009″

13) Job Control

```
^Z $bg $disown
```

You're running a script, command, whatever.. You don't expect it to take long, now 5pm has rolled around and you're ready to go home… Wait, it's still running… You forgot to nohup it before running it… Suspend it, send it to the background, then disown it… The ouput wont go anywhere, but at least the command will still run…

14) Edit a file on a remote host using vim

```
vim scp://username@host//path/to/somefile
```

15) Monitor the queries being run by MySQL

```
watch -n 1 mysqladmin --user=<user> --password=<password> processlist
```

Watch is a very useful command for periodically running another command – in this using mysqladmin to display the processlist. This is useful for monitoring which queries are causing your server to clog up.

More info here: http://codeinthehole.com/archives/2-Monitoring-MySQL-processes.html

16) escape any command aliases

```
\[command]
```

e.g. if rm is aliased for 'rm -i', you can escape the alias by prepending a backslash:

rm [file] # WILL prompt for confirmation per the alias

\rm [file] # will NOT prompt for confirmation per the default behavior of the command

17) Show apps that use internet connection at the moment. (Multi-Language)

```
ss -p
```

for one line per process:

ss -p | catfor established sockets only:

ss -p | grep STAfor just process names:

ss -p | cut -f2 -sd\"or

ss -p | grep STA | cut -f2 -d\"

18) Send pop-up notifications on Gnome

```
notify-send ["<title>"] "<body>"
```

The title is optional.

Options:

-t: expire time in milliseconds.

-u: urgency (low, normal, critical).

-i: icon path.

On Debian-based systems you may need to install the 'libnotify-bin' package.

Useful to advise when a wget download or a simulation ends. Example:

```
wget URL ; notify-send "Done"
```

19) quickly rename a file

```
mv filename.{old,new}
```

20) Remove all but one specific file

```
rm -f !(survivior.txt)
```

21) Generate a random password 30 characters long

```
strings /dev/urandom | grep -o '[[:alnum:]]' | head -n 30 | tr -d '\n'; echo
```

Find random strings within /dev/urandom. Using grep filter to just Alphanumeric characters, and then print the first 30 and remove all the line feeds.

22) Run a command only when load average is below a certain threshold

```
echo "rm -rf /unwanted-but-large/folder" | batch
```

Good for one off jobs that you want to run at a quiet time. The default threshold is a load average of 0.8 but this can be set using atrun.

23) Binary Clock

```
watch -n 1 'echo "obase=2;`date +%s`" | bc'
```

Create a binary clock.

24) Processor / memory bandwidthd? in GB/s

```
dd if=/dev/zero of=/dev/null bs=1M count=32768
```

Read 32GB zero's and throw them away.

How fast is your system?

25) Backup all MySQL Databases to individual files

```
for I in $(mysql -e 'show databases' -s --skip-column-names);
do mysqldump $I | gzp > "$I.sql.gz"; done
```

[May 08, 2014] 25 Best Linux Commands UrFix's Blog

25) sshfs name@server:/path/to/folder /path/to/mount/point
Mount folder/filesystem through SSH
Install SSHFS from http://fuse.sourceforge.net/sshfs.html
Will allow you to mount a folder security over a network.

24) !!:gs/foo/bar
Runs previous command replacing foo by bar every time that foo appears
Very useful for rerunning a long command changing some arguments globally.
As opposed to ^foo^bar, which only replaces the first occurrence of foo, this one changes every occurrence.

23) mount | column -t
currently mounted filesystems in nice layout
Particularly useful if you're mounting different drives, using the following command will allow you to see all the filesystems currently mounted on your computer and their respective specs with the added benefit of nice formatting.

22) <space>command
Execute a command without saving it in the history
Prepending one or more spaces to your command won't be saved in history.
Useful for pr0n or passwords on the commandline.

21) ssh user@host cat /path/to/remotefile | diff /path/to/localfile -
Compare a remote file with a local file
Useful for checking if there are differences between local and remote files.

20) mount -t tmpfs tmpfs /mnt -o size=1024m
Mount a temporary ram partition
Makes a partition in ram which is useful if you need a temporary working space as read/write access is fast.
Be aware that anything saved in this partition will be gone after your computer is turned off.

19) dig +short txt <keyword>.wp.dg.cx
Query Wikipedia via console over DNS
Query Wikipedia by issuing a DNS query for a TXT record. The TXT record will also include a short URL to the complete corresponding Wikipedia entry.

18) netstat -tlnp
Lists all listening ports together with the PID of the associated process
The PID will only be printed if you're holding a root equivalent ID.

17) dd if=/dev/dsp | ssh -c arcfour -C username@host dd of=/dev/dsp
output your microphone to a remote computer's speaker
This will output the sound from your microphone port to the ssh target computer's speaker port. The sound quality is very bad, so you will hear a lot of hissing.

16) echo "ls -l" | at midnight
Execute a command at a given time
This is an alternative to cron which allows a one-off task to be scheduled for a certain time.

15) curl -u user:pass -d status="Tweeting from the shell" http://twitter.com/statuses/update.xml
Update twitter via curl

14) ssh -N -L2001:localhost:80 somemachine
start a tunnel from some machine's port 80 to your local post 2001
now you can acces the website by going to http://localhost:2001/

13) reset
Salvage a borked terminal
If you bork your terminal by sending binary data to STDOUT or similar, you can get your terminal back using this command rather than killing and restarting the session. Note that you often won't be able to see the characters as you type them.

12) ffmpeg -f x11grab -s wxga -r 25 -i :0.0 -sameq /tmp/out.mpg
Capture video of a linux desktop

11) > file.txt
Empty a file
For when you want to flush all content from a file without removing it (hat-tip to Marc Kilgus).

10) $ssh-copy-id user@host
Copy ssh keys to user@host to enable password-less ssh logins.
To generate the keys use the command ssh-keygen

9) ctrl-x e
Rapidly invoke an editor to write a long, complex, or tricky command
Next time you are using your shell, try typing ctrl-x e (that is holding control key press x and then e). The shell will take what you've written on the command line thus far and paste it into the editor specified by $EDITOR. Then you can edit at leisure using all the powerful macros and commands of vi, emacs, nano, or whatever.

8) !whatever:p
Check command history, but avoid running it
!whatever will search your command history and execute the first command that matches 'whatever'. If you don't feel safe doing this put :p on the end to print without executing. Recommended when running as superuser.

7) mtr google.com
mtr, better than traceroute and ping combined
mtr combines the functionality of the traceroute and ping programs in a single network diagnostic tool.
As mtr starts, it investigates the network connection between the host mtr runs on and HOSTNAME. by sending packets with purposly low TTLs. It continues to send packets with low TTL, noting the response time of the intervening routers. This allows mtr to print the response percentage and response times of the internet route to HOSTNAME. A sudden increase in packetloss or response time is often an indication of a bad (or simply over-loaded) link.

6) cp filename{,.bak}
quickly backup or copy a file with bash

5) ^foo^bar
Runs previous command but replacing
Really useful for when you have a typo in a previous command. Also, arguments default to empty so if you accidentally run: echo "no typozs"
you can correct it with ^z

4) cd -
change to the previous working directory

3):w !sudo tee %
Save a file you edited in vim without the needed permissions
I often forget to sudo before editing a file I don't have write permissions on. When you come to save that file and get the infamous "E212: Can't open file for writing", just issue that vim command in order to save the file without the need to save it to a temp file and then copy it back again.

2) python -m SimpleHTTPServer
Serve current directory tree at http://$HOSTNAME:8000/

1) sudo !!
Run the last command as root
Useful when you forget to use sudo for a command. "!!" grabs the last run command.

[Dec 16, 2012] bash - how do I list the functions defined in my shell - Stack Overflow

Function names and definitions may be listed with the -f option to the declare or typeset builtin commands (see Bash Builtins). The -F option to declare or typeset will list the function names only (and optionally the source file and line number

Unknown Bash Tips and Tricks For Linux Linux.com

Bash Builtins

Bash has a bunch of built-in commands, and some of them are stripped-down versions of their external GNU coreutils cousins. So why use them? You probably already do, because of the order of command execution in Bash:

1. Bash aliases
2. Bash keywords
3. Bash functions
4. Bash builtins
5. Scripts and executable programs that are in your PATH

So when you run echo, kill, printf, pwd, or test most likely you're using the Bash builtins rather than the GNU coreutils commands. How do you know? By using one of the Bash builtins to tell you, the command command:

```
$ command -V echo
echo is a shell builtin

$ command -V ping
ping is /bin/ping
```

The Bash builtins do not have man pages, but they do have a backwards help builtin command that displays syntax and options:

```
$ help echo
echo: echo [-neE] [arg ...]
    Write arguments to the standard output.

    Display the ARGs on the standard output followed by a newline.

    Options:
      -n       do not append a newline
      -e       enable interpretation of the following backslash escapes
[...]
```

I call it backwards because most Linux commands use a syntax of commandname --help, where help is a command option instead of a command.

The type command looks a lot like the command builtin, but it does more:

```
$ type -a cat
cat is /bin/cat

$ type -t cat
file

$ type ll
```

```
ll is aliased to `ls -alF'

$ type -a echo
echo is a shell builtin
echo is /bin/echo

$ type -t grep
alias
```

The type utility identifies builtin commands, functions, aliases, keywords (also called reserved words), and also binary executables and scripts, which it calls ***file***. At this point, if you are like me, you are grumbling "How about showing me a LIST of the darned things." I hear and obey, for you can find these delightfully documented in the The GNU Bash Reference Manual indexes. Don't be afraid, because unlike most software documention this isn't a scary mythical creature like Sasquatch, but a real live complete command reference.

The point of this little exercise is so you know what you're really using when you type a command into the Bash shell, and so you know how it looks to Bash. There is one more overlapping Bash builtin, and that is the time keyword:

```
$ type -t time
keyword
```

So why would you want to use Bash builtins instead of their GNU cousins? Builtins may execute a little faster than the external commands, because external commands have to fork an extra process. I doubt this is much of an issue on modern computers because we have horsepower to burn, unlike the olden days when all we had were tiny little nanohertzes, but when you're tweaking performance it's one thing to look at. When you want to use the GNU command instead of the Bash builtin use its whole path, which you can find with command, type, or the good old not-Bash command which:

```
$ which echo
/bin/echo

$ which which
/usr/bin/which
```

Bash Functions

Run declare -F to see a list of Bash's builtin function names. declare -f prints out the complete functions, and declare -f [function-name] prints the named function. type won't find list functions, but once you know a function name it will also print it:

```
$ type quote
quote is a function
quote ()
{
    echo \'${1//\'/\'\\\'\'}\'
}
```

This even works for your own functions that you create, like this simple example testfunc that does one thing: changes to the /etc directory:

```
$ function testfunc
> {
> cd /etc
> }
```

Now you can use declare and type to list and view your new function just like the builtins.

Bash's Violent Side

Don't be fooled by Bash's calm, obedient exterior, because it is capable of killing. There have been a lot of changes to how Linux manages processes, in some cases making them more difficult to stop, so knowing how to kill runaway processes is still an important bit of knowledge. Fortunately, despite all this newfangled "progress" the reliable old killers still work.

I've had some troubles with bleeding-edge releases of KMail; it hangs and doesn't want to close by normal means. It spawns a single process, which we can see with the ps command:

```
ps axf|grep kmail
 2489 ?     Sl  1:44 /usr/bin/kmail -caption KMail
```

You can start out gently and try this:

```
$ kill 2489
```

This sends the default SIGTERM (signal terminate) signal, which is similar to the SIGINT (signal interrupt) sent from the keyboard with Ctrl+c. So what if this doesn't work? Then you amp up your stopping power and use SIGKILL, like this:

```
$ kill -9 2489
```

This is the nuclear option and it will work. As the relevant section of the GNU C manual says: "The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal." This is different from SIGTERM and SIGINT and other signals that politely ask processes to terminate. They can be trapped and handled in different ways, and even blocked, so the response you get to a SIGTERM depends on how the program you're trying to kill has been programmed to handle signals. In an ideal world a program responds to SIGTERM by tidying up before exiting, like finishing disk writes and deleting temporary files. SIGKILL knocks it out and doesn't give it a chance to do any cleanup. (See man 7 signal for a complete description of all signals.)

So what's special about Bash kill over GNU /bin/kill? My favorite is how it looks when you invoke the online help summary:

```
$ help kill
```

Another advantage is it can use job control numbers in addition to PIDs. In this modern era of tabbed terminal emulators job control isn't the big deal it used to be, but the option is there if you want it. The biggest advantage is you can kill processes even if they have gone berserk and maxed out your system's process number limit, which would prevent you from launching /bin/kill. Yes, there is a limit, and you can see what it is by querying /proc:

```
$ cat /proc/sys/kernel/threads-max
61985
```

With Bash kill there are several ways to specify which signal you want to use. These are all the same:

```
$ kill 2489
$ kill -s TERM 2489
$ kill -s SIGTERM 2489
$ kill -n 15 2489
```

kill -l lists all supported signals.

If you spend a little quality time with man bash and the GNU Bash Manual I daresay you will learn more valuable tasks that Bash can do for you.

My Favorite Bash Substitution Tricks Drastic Code

My Favorite Bash Substitution Tricks

August 01, 2009

Here's a few tricks that I often use on the command line to save time. They take advantage of some variables that the bash shell uses to store various aspects of your history.

Repeating the last command with !!

Sometimes I run a command that requires sudo access, but forget the sudo. This is a great opportunity to use !! which holds the last command you ran.

```
$ tail /var/log/mail.log
tail: cannot open `/var/log/mail.log' for reading: Permission denied
$ sudo !!
sudo tail /var/log/mail.log
# output of command
```

The last argument of the last command using !$

Sometimes it's handy to be able to reference the last argument of your last command. This can make certain operations safer, by preventing a fat fingered typo from deleting important files.

```
$ ls *.log
a.log b.log
$ rm -v !$
removed `a.log'
removed `b.log'
```

Similarly you can use !* to reference all of the last commands' arguments.

```
$ touch a.log b.log
$ rm -v !*
rm -v a.log b.log
removed `a.log'
removed `b.log'
```

Correcting mistakes with ^^

This is a nifty trick that performs a substitution on your last command. It's great for correcting typos, or running similar commands back to back. It looks for a match with whatever is after the first carrot, and replaces it with whatever is after the second.

```
$ cmhod a+x my_script.sh
-bash: cmhod: command not found
$ ^mh^hm
chmod a+x my_script.sh
```

I use this one all the time doing rails development if I make a mistake on a script/generate command.

```
$ script/generate model Animal species:string sex:string birthday:date
      exists app/models/
      exists test/unit/
      exists test/fixtures/
      create app/models/animal.rb
      create test/unit/animal_test.rb
      create test/fixtures/animals.yml
      create db/migrate
      create db/migrate/20090801180754_create_animals.rb
```

```
$ ^generate^destroy
script/destroy model Animal species:string sex:string birthday:date
notempty db/migrate
notempty db
rm db/migrate/20090801180754_create_animals.rb
rm test/fixtures/animals.yml
rm test/unit/animal_test.rb
rm app/models/animal.rb
```

```
rmdir test/fixtures
notempty test
rmdir test/unit
notempty test
rmdir app/models
notempty app
```

```
$ ^destroy ^generate rspec_
script/generate rspec_model Animal species:string sex:string birthday:date
create app/models/
create spec/models/
create spec/fixtures/
create app/models/animal.rb
create spec/models/animal_spec.rb
create spec/fixtures/animals.yml
create db/migrate
create db/migrate/20090801180937_create_animals.rb
```

Hope someone else finds these as handy as I do.

Tagged with: bash command-line tips |

Comments

1. ***Sam*** August 02, 2009 @ 11:20 AM

   One more tip:

   You can also echo a specific number of arguments off the end of the last command using !:n*, where n is the number of the first argument to echo. For example:

   ```
   $ touch 1.log 2.log 3.log 4.log 5.log

   $ rm -v !:3*
   rm -v 3.log 4.log 5.log
   3.log
   4.log
   5.log
   ```

   I don't use this one too much in practice but it could come in handy in certain situations.

2. ***Kirsten*** August 03, 2009 @ 05:15 PM

   Thanks Sam, I didn't know about !* and the ^^ substitution, those will be useful!

Re New line in bash variables pain

Maxim Vexler
Tue, 14 Nov 2006 12:40:28 -0800

On 11/14/06, Oded Arbel <> wrote:
[snip]

> (IFS="$(echo)"; \
> for pair in `awk '/^[^[].+[^\n]$/ {print $1,$3}' passwd.fake`; do
> echo "$pair"; done)
>
> In the second example, I force the record separator to be only the new
> line character (the output from 'echo'. I can probably use \n, but I
> wanted to play it safe). Do mind the wrapping of the second form in
> parenthesis, otherwise you clobber your global IFS, which is something
> you want to avoid.
>
> --
> Oded
> ::..
> We make a living by what we get, but we make a life by what we give.
> -- Winston Churchill

Thanks to everyone for the help, all solution worked.
To sum up the tips:

By Oded Arbel:
a. Use a subshell to avoid mistakenly over riding your shell variables.
b. Use "$(echo)" as portable(?) newline variable scripting style.

By Ehud Karni:
a. Pipeing into bash subshell can be accepted inside the shell with read.
b. using a "while read VAR1 VAR2 VAR3..." is a convenient method to
accepting stdin data.
c. awk has system() !!

By Amos Shapira:
a. General work around is to construct the whole command as text, then
use either piping to sh or bash buildin "expr".

By Omer Shapira:
a. xargs -n switch can be used to "collect" variables separated by either of [\n\t].

By Valery Reznic:
a. set -- "space delimited word list" can be used as a quick method for assigning value to number variables ($1..$9). [question: Really? this does not seem to work for me].
b. bash while loop can get stdin from file IO redirection.

Ariel Biener doesn't understand the need for voodoo in modern life... ;)

Thanks guys for an educational thread.

[Mar 17, 2010] Power Shell Usage Bash Tips & Tricks

Searching the Past

- There are several bad ways of finding previous lines from history
  - Many people go for pressing Up lots (and lots)
    - A tad inefficient, perhaps
- Ctrl+R **searches previous lines**
  - But Ctrl+R zip Esc doesn't find the last zip command - it also matches any line that copied, deleted, unzipped, or did anything else with a zip file
- Those of a gambling bent can chance ! and a command name
  - Irritating when !gv opens gvim instead of gv
- **Sane Incremental Searching**
  - Bash can cycle through lines starting in a particular way
  - Just type in a few characters then press Up
    - Don't need to press Up so many times
    - Don't see lines that merely contain those letters
    - Don't have to chance executing the wrong line
  - Incremental searching with Up and Down is configured in .inputrc

```
"\e[A": history-search-backward
"\e[B": history-search-forward
```

  - Old behavior still available with Ctrl+P and Ctrl+N
  - If that prevents Left and Right from working, fix them like this:

```
"\e[C": forward-char
"\e[D": backward-char
```

- **Repeating Command Arguments**
  - Commonly want to repeat just bits of commands
  - Very often the previous command's last argument
  - Meta+. retrieves the last argument. Press repeatedly to cycle through the final argument from earlier commands
- **Magic Space**
  - A magic space inserts a space character as normal
  - And also performs history expansion in the line
  - See what you type before you commit to it
    - Press Space before Enter if necessary
  - Magic Space Set-Up
    - Magic space is configured in .inputrc
    - Redefine what Space does
    - There are other readline-based programs without this feature, so make it only apply in Bash:

```
$if Bash
  Space: magic-space
$endif
```

- **Forgetting Options**
  - Common to forget an option from a command
  - Want to rerun the command with the option
  - Go to the previous history line, then move just after the command name to type the option
  - Can set up a keyboard macro to do this
  - Insert-Option Macro
    - Meta+O can be made to load the previous command and position the cursor for typing an option
    - Defined in .inputrc:

```
"\M-o": "\C-p\C-a\M-f "
```

      - Ctrl+P: previous line
      - Ctrl+A: start of line
      - Meta+F: forward a word, past the command
      - _: insert a space
    - 17 unused keystrokes with just Ctrl or Meta modifiers

[Aug 9, 2009] My Favorite bash Tips and Tricks

One last tip I'd like to offer is using loops from the command line. The command line is not the place to write complicated scripts that include multiple loops or branching. For small loops, though, it can be a great time saver. Unfortunately, I don't see many people taking advantage of this. Instead, I frequently see people use the up arrow key to go back in the command history and modify the previous command for each iteration.

If you are not familiar with creating for loops or other types of loops, many good books on shell scripting discuss this topic. A discussion on for loops in general is an article in itself.

You can write loops interactively in two ways. The first way, and the method I prefer, is to separate each line with a semicolon. A simple loop to make a backup copy of all the files in a directory would look like this:

```
$ for file in * ; do cp $file $file.bak; done
```

Another way to write loops is to press Enter after each line instead of inserting a semicolon. bash recognizes that you are creating a loop from the use of the for keyword, and it prompts you for the next line with a secondary prompt. It knows you are done when you enter the keyword done, signifying that your loop is complete:

```
$ for file in *
> do cp $file $file.bak
> done
```

[Aug 4, 2009] Tech Tip View Config Files Without Comments Linux Journal

I've been using this grep invocation for years to trim comments out of config files. Comments are great but can get in your way if you just want to see the currently running configuration. I've found files hundreds of lines long which had fewer than ten active configuration lines, it's really hard to get an overview of what's going on when you have to wade through hundreds of lines of comments.

```
$ grep ^[^#] /etc/ntp.conf
```

The regex ^[^#] matches the first character of any line, as long as that character that is not a #. Because blank lines don't have a first character they're not matched either, resulting in a nice compact output of just the active configuration lines.

The Various bash Prompts by Juliet Kemp

PS4 is the prompt shown when you set the debug mode on a shell script using set -x at the top of the script. This echoes each line of the script to STDOUT before executing it. The default prompt is ++. More usefully, you can set it to display the line number, with:

export PS4='$LINENO+ '

It's fairly likely that you already have a personalized setting for PS1, the default bash interaction prompt. But what about the others available: PS2, PS3, and PS4?

PS1 is the default interaction prompt. To set it to give you

```
username@host:directory$
```

use

```
export PS1="u@h w$ "
```

in your ~/.bash_rc. u is the current username, h the current host, and w the working directory. There's a list of escape codes you can use in the bash man page, or in the Bash Prompt HOWTO.

PS2 is the prompt you get when you extend a command over multiple lines by putting at the end of a line and hitting return. By default it's just >, but you can make this a little more obvious with:

```
export PS2="more -> "
```

so it looks like:

```
juliet@glade:~ $ very-long-command-here
more -> -with -lots -of -options
```

PS3 governs the prompt that shows up if you use the select statement in a shell script. The default is #?, so if you do nothing to change that, the select statement will print out the options and then just leave that prompt. Alternatively, use this:

```
PS3="Choose an option: "
select i in yes maybe no
do
      # code to handle reply
done
```

which will output:

```
1) yes
2) maybe
3) no
Choose an option:
```

Far more readable for the user!

Finally, PS4 is the prompt shown when you set the debug mode on a shell script using set -x at the top of the script. This echoes each line of the script to STDOUT before executing it. The default prompt is ++. More usefully, you can set it to display the line number, with:

```
export PS4='$LINENO+ '
```

All of these can be made to be permanent changes by setting them in your ~/.bash_profile or ~/.bashrc file. (Note that this probably makes little sense to do for PS3, which is better to set per-script.)

Recovering Deleted Files With lsof By Juliet Kemp

One of the more neat things you can do with the versatile utility lsof is use it to recover a file you've just accidentally deleted.

A file in Linux is a pointer to an inode, which contains the file data (permissions, owner and where its actual content lives on the disk). Deleting the file removes the link, but not the inode itself – if another process has it open, the inode isn't released for writing until that process is done with it.

To try this out, create a test text file, save it and then type less test.txt. Open another terminal window, and type rm testing.txt. If you try ls testing.txt you'll get an error message. But! less still has a reference to the file. So:

```
> lsof | grep testing.txt
less  4607       juliet  4r  REG 254,4   21
        8880214 /home/juliet/testing.txt (deleted)
```

The important columns are the second one, which gives you the PID of the process that has the file open (4607), and the fourth one, which gives you the file descriptor (4). Now, we go look in /proc, where there will still be a reference to the inode, from which you can copy the file back out:

```
> ls -l /proc/4607/fd/4
lr-x------ 1 juliet juliet 64 Apr  7 03:19
          /proc/4607/fd/4 -> /home/juliet/testing.txt (deleted)
> cp /proc/4607/fd/4 testing.txt.bk
```

Note: **don't** use the -a flag with cp, as this will copy the (broken) symbolic link, rather than the actual file contents.

[Jul 7, 2009] xclip Command-Line Clipboard

xclip (available as a package for Debian and Ubuntu) enables you to interact with the X clipboard directly from the command-line - without having to use the mouse to cut and paste.

This is particularly useful if you're trying to get command-line output over to an e-mail or web page. Instead of scrolling around in the terminal to cut and paste with the mouse, screen by screen, you can use this:

```
command --arg | xclip
```

Then go to whichever graphical program you want to paste the input into, and paste with the middle mouse button or the appropriate menu item.

You can also enter the contents of a file straight into xclip:

```
xclip /path/to/file
```

and again, can then paste that directly wherever you want it.

The -o option enables you to operate it the other way around: output the contents of the clipboard straight onto the command line. So, you could, for example, copy a command line from a web page, then use

```
xclip -o
```

to output it. To output to a file, use

```
xclip -o /path/to/file
```

Use the -selection switch to use the buffer-cut or one of the other selection options, rather than the clipboard default. You can also hook it up to an X display other than the default one (e.g., if you're logged on as a different user on :!) with

```
xclip -d localhost:1
```

[Jun 29, 2009] !! provides the ability to rerun long commands which cannot be executed on your current account without prefixing them with sudo.

```
$ whoami
$ sudo !!
```

[Mar 14, 2009] How to Be Faster at the Linux Command Line

02/05/2009 | hacktux.com

Want to be faster at the Linux command line interface? Since most Linux distributions provide Bash as the default CLI, here are some Bash tricks that will help cut down the amount of typing needed to execute commands. Feel free to comment and share your own speed tricks.

- Control-R Through Your History
- Use History Expansion
- Use History Quick Substitution
- Use Vi or Emacs Editing Mode
- Use Aliases and Functions

Control-R Through Your History

This is my most used shortcut. Hit ***Control-R*** and begin to type a string. You immediately get the last command in your Bash history with that string. Hit ***Control-R*** again to cycle further backwards in your history.

For instance, type the following and hit ***Enter***.

```
grep root /etc/passwd
```

Then hit ***Control-R*** and begin to type 'grep'.

```
Control-R
(reverse-i-search)`gre': grep root /etc/passwd
```

When you see the original command listed, hit ***Enter*** to execute it. Alternatively, you can also hit the ***Right-Arrow*** to edit the command before running it.

Use History Expansion

Bash's command history can be referenced using the exclamation mark. For instance, typing two exclamation marks (!!) will re-execute the last command. The next example executes ***date*** twice:

```
date
!!
```

If you are interested in more than just the last command executed, type ***history*** to see a numbered listing of your Bash's history.

```
history
39 grep root /etc/passwd
40 date
41 date
42 history
```

Since ***grep root /etc/passwd*** is command number 39, you can re-execute it like so:

```
!39
```

You can also reference Bash's history using a search string. For instance, the following will run the last command that started with 'grep'.

```
!grep
```

Note, you can set the number of commands stored in your history by setting HISTSIZE.

```
export HISTSIZE=1000
```

You can also wipe your history clear with the -c switch.

```
history -c
```

Use History Quick Substitution

Historical commands can be edited and reused with quick substitution. Let's say you ***grep*** for 'root' in /etc/passwd:

```
grep root /etc/passwd
```

Now, you need to ***grep*** for 'root' in /etc/group. Substitute 'passwd' for 'group' in the last command using the caret (^).

```
^passwd^group
```

The above command will run:

```
grep root /etc/group
```

Comments

Sun, 02/08/2009 - 2:25pm - Anonymous (not verified)

For my backup function, I

For my backup function, I use pass the %F-%R to my date command. This would allow me to make multiple backup copies of a file in one day and have them ordered by date/time.

Keith

- reply

Thu, 02/05/2009 - 2:58pm - Anonymous (not verified)

Thankyou for ctrl R I have

Thankyou for ctrl R
I have been using command line for two years and one of
my biggest grips was this issue. I and now flying around the command line
thanks

- reply

Wed, 02/04/2009 - 5:59pm - Max (not verified)

Nice set of tricks. I knew

Nice set of tricks. I knew most of them already but it refreshed my memory. Thanks.

I find even more handy to have this in ~/.inputrc :
-------- Bind page up/down wih history search ---------
"\e[5~": history-search-backward
"\e[6~": history-search-forward

I'll take the same example : on the bash prompt, type "gre" and Page up, this will give you "grep root /etc/passwd", the last command that started with "gre". Enter Page up again and it'll show you the previous one. Page down is obvioulsy used to show the next one.

I just noticed that the "set -o vi" trick is messing with this one ^_^ Can't tell you why.

- reply

Thu, 02/05/2009 - 5:43am - MaximB (not verified)

Nice stuff... There are some

Nice stuff...

There are some GNU/Linux distributions that already use aliases "built-in" .
like rm which is "rm -i" in rhel5 . So if you want to ignore the alias for known commands like rm for example, just type :

```
command rm
```

it will ignore the alias for the command.

[Feb 22, 2009] 10 shortcuts to master bash - Program - Linux - Builder AU By Guest Contributor, TechRepublic | 2007/06/25 18:30:02

If you've ever typed a command at the Linux shell prompt, you've probably already used bash -- after all, it's the default command shell on most modern GNU/Linux distributions.

The bash shell is the primary interface to the Linux operating system -- it accepts, interprets and executes your commands, and provides you with the building blocks for shell scripting and automated task execution.

Bash's unassuming exterior hides some very powerful tools and shortcuts. If you're a heavy user of the command line, these can save you a fair bit of typing. This document outlines 10 of the most useful tools:

1. **Easily recall previous commands**

   Bash keeps track of the commands you execute in a history buffer, and allows you to recall previous commands by cycling through them with the Up and Down cursor keys. For even faster recall, "speed search" previously-executed commands by typing the first few letters of the command followed by the key combination Ctrl-R; bash will then scan the command history for matching commands and display them on the console. Type Ctrl-R repeatedly to cycle through the entire list of matching commands.

2. **Use command aliases**

   If you always run a command with the same set of options, you can have bash create an alias for it. This alias will incorporate the required options, so that you don't need to remember them or manually type them every time. For example, if you always run ls with the -l option to obtain a detailed directory listing, you can use this command:

   ```
   bash> alias ls='ls -l'
   ```

   To create an alias that automatically includes the -l option. Once this alias has been created, typing ls at the bash prompt will invoke the alias and produce the ls -l output.

   You can obtain a list of available aliases by invoking alias without any arguments, and you can delete an alias with unalias.

3. **Use filename auto-completion**

   Bash supports filename auto-completion at the command prompt. To use this feature, type the first few letters of the file name, followed by Tab. bash will scan the current directory, as well as all other directories in the search path, for matches to that name. If a single match is found, bash will automatically complete the filename for you. If multiple matches are found, you will be prompted to choose one.

4. **Use key shortcuts to efficiently edit the command line**

   Bash supports a number of keyboard shortcuts for command-line navigation and editing. The Ctrl-A key shortcut moves the cursor to the beginning of the command line, while the Ctrl-E shortcut moves the cursor to the end of the command line. The Ctrl-W shortcut deletes the word immediately before the cursor, while the Ctrl-K shortcut deletes everything immediately after the cursor. You can undo a deletion with Ctrl-Y.

5. **Get automatic notification of new mail**

   You can configure bash to automatically notify you of new mail, by setting the $MAILPATH variable to point to your local mail spool. For example, the command:

   ```
   bash> MAILPATH='/var/spool/mail/john'
   bash> export MAILPATH
   ```

   Causes bash to print a notification on john's console every time a new message is appended to John's mail spool.

6. **Run tasks in the background**

   Bash lets you run one or more tasks in the background, and selectively suspend or resume any of the current tasks (or "jobs"). To run a task in the background, add an ampersand (&) to the end of its command line. Here's an example:

   ```
   bash> tail -f /var/log/messages &
   [1] 614
   ```

   Each task backgrounded in this manner is assigned a job ID, which is printed to the console. A task can be brought back to the foreground with the command fg ***jobnumber***, where ***jobnumber*** is the job ID of the task you wish to bring to the foreground. Here's an example:

   ```
   bash> fg 1
   ```

   A list of active jobs can be obtained at any time by typing jobs at the bash prompt.

7. **Quickly jump to frequently-used directories**

   You probably already know that the $PATH variable lists bash's "search path" -- the directories it will search when it can't find the requested file in the current directory. However, bash also supports the $CDPATH variable, which lists the directories the cd command will look in when attempting to change directories. To use this feature, assign a directory list to the $CDPATH variable, as shown in the example below:

   ```
   bash> CDPATH='.:~:/usr/local/apache/htdocs:/disk1/backups'
   bash> export CDPATH
   ```

   Now, whenever you use the cd command, bash will check all the directories in the $CDPATH list for matches to the directory name.

8. **Perform calculations**

   Bash can perform simple arithmetic operations at the command prompt. To use this feature, simply type in the arithmetic expression you wish to evaluate at the prompt within double parentheses, as illustrated below. Bash will attempt to perform the calculation and return the answer.

   ```
   bash> echo $((16/2))
   8
   ```

9. **Customise the shell prompt**

   You can customise the bash shell prompt to display -- among other things -- the current username and host name, the current time, the load average and/or the current working directory. To do this, alter the $PS1 variable, as below:

   ```
   bash> PS1='\u@\h:\w \@> '

   bash> export PS1
   root@medusa:/tmp 03:01 PM>
   ```

   This will display the name of the currently logged-in user, the host name, the current working directory and the current time at the shell prompt. You can obtain a list of symbols understood by bash from its manual page.

10. **Get context-specific help**

    Bash comes with help for all built-in commands. To see a list of all built-in commands, type help. To obtain help on a specific command, type help ***command***, where ***command*** is the command you need help on. Here's an example:

    ```
    bash> help alias
    ...some help text...
    ```

    Obviously, you can obtain detailed help on the bash shell by typing man bash at your command prompt at any time.

How to Be Faster at the Linux Command Line

Want to be faster at the Linux command line interface? Since most Linux distributions provide Bash as the default CLI, here are some Bash tricks that will help cut down the amount of typing needed to execute commands. Feel free to comment and share your own speed tricks.

Control-R Through Your History

This is my most used shortcut. Hit ***Control-R*** and begin to type a string. You immediately get the last command in your Bash history with that string. Hit ***Control-R*** again to cycle further backwards in your history.

For instance, type the following and hit ***Enter***.

```
grep root /etc/passwd
```

Then hit ***Control-R*** and begin to type 'grep'.

```
Control-R
(reverse-i-search)`gre': grep root /etc/passwd
```

When you see the original command listed, hit ***Enter*** to execute it. Alternatively, you can also hit the ***Right-Arrow*** to edit the command before running it.

Use History Expansion

Bash's command history can be referenced using the exclamation mark. For instance, typing two exclamation marks (!!) will re-execute the last command. The next example executes ***date*** twice:

```
date
!!
```

If you are interested in more than just the last command executed, type ***history*** to see a numbered listing of your Bash's history.

```
history
39 grep root /etc/passwd
40 date
41 date
42 history
```

Since ***grep root /etc/passwd*** is command number 39, you can re-execute it like so:

```
!39
```

You can also reference Bash's history using a search string. For instance, the following will run the last command that started with 'grep'.

```
!grep
```

Note, you can set the number of commands stored in your history by setting HISTSIZE.

```
export HISTSIZE=1000
```

You can also wipe your history clear with the -c switch.

```
history -c
```

Use History Quick Substitution

Historical commands can be edited and reused with quick substitution. Let's say you ***grep*** for 'root' in /etc/passwd:

```
grep root /etc/passwd
```

Now, you need to ***grep*** for 'root' in /etc/group. Substitute 'passwd' for 'group' in the last command using the caret (^).

```
^passwd^group
```

The above command will run:

```
grep root /etc/group
```

Use Vi or Emacs Editing Mode

You can further enhance your abilities to edit previous commands using Vi or Emacs keystrokes. For example, the following sets Vi style command line editing:

```
set -o vi
```

After setting Vi mode, try it out by typing a command and hitting ***Enter***.

```
grep root /etc/passwd
```

Then, ***Up-Arrow*** once to the same command:

```
Up-Arrow
grep root /etc/passwd
```

Now, move the cursor to the 'p' in 'passwd' and hit ***Esc***.

```
grep root /etc/passwd
^
```

Now, use the Vi ***cw*** command to change the word 'passwd' to 'group'.

```
grep root /etc/group
```

For more Vi mode options, see this list of commands available in Vi mode. Alternatively, If you prefer Emacs, use Bash's Emacs mode:

```
set -o emacs
```

Emacs mode provides shortcuts that are available through the ***Control*** and ***Alt*** key. For example, ***Control-A*** takes you to the beginning of the line and ***Control-E*** takes you to the end of the line. Here is a list of commands available in Bash's Emacs mode.

Use Aliases and Functions

Bash allows for commands, or sets of commands, to be aliased into a single instruction. Your interactive Bash shell should already load some useful aliases from /etc/profile.d/. For one, you probably have ***ll*** aliased to ***ls -l***.

If you want to see all aliases loaded, run the ***alias*** Bash builtin.

```
alias
```

To create an alias, use the ***alias*** command:

```
alias ll='ls -l'
```

Here are some other common aliases:

```
alias ls='ls --color=tty'
alias l.='ls -d .* --color=auto'
alias cp='cp -i'
alias mv='mv -i'
```

Note that you can also string together commands. The follow will alias ***gohome*** as ***cd*** , then run ***ls***. Note that running ***cd*** without any arguments will change directory to your $HOME directory.

```
alias gohome='cd; ls'
```

Better yet, only run ***ls*** if the ***cd*** is successful:

```
alias gohome='cd && ls || echo "error($?) with cd to $HOME"'
```

More complex commands can be written into a Bash function. Functions will allow you to provide input parameters for a block of code. For instance, let's say you want to create a backup function that puts a user inputted file into ~/backups.

```
backup() {
file=${1:?"error: I need a file to backup"}

timestamp=$(date '+%m%d%y')
backupdir=~/backups

[ -d ${backupdir} ] || mkdir -p ${backupdir}
cp -a ${file} ${backupdir}/$(basename ${file}).${timestamp}
return $?
}
```

Like the example above, use functions to automate small, daily tasks. Here is one I use to set my xterm title.

```
xtitle() {
unset PROMPT_COMMAND
echo -ne "\033]0;${@}\007"
}
```

Of course, you can use functions together with aliases. Here is one I use to set my xterm title to 'MAIL' and then run Mutt.

```
alias mutt='xtitle "MAIL" && /usr/bin/mutt'
```

Finally, to ensure that your custom aliases and functions are available each login, add them to your .bashrc.

```
vim ~/.bashrc
```

[Apr 2, 2008] 10 shortcuts to master bash - Program - Linux - Builder AU

2007/06/25 | Guest Contributor, TechRepublic

1. Easily recall previous commands

Bash keeps track of the commands you execute in a history buffer, and allows you to recall previous commands by cycling through them with the Up and Down cursor keys. For even faster recall, "speed search" previously-executed commands by typing the first few letters of the command followed by the key combination Ctrl-R; bash will then scan the command history for matching commands and display them on the console. Type Ctrl-R repeatedly to cycle through the entire list of matching commands.

... ...

5. Get automatic notification of new mail

You can configure bash to automatically notify you of new mail, by setting the $MAILPATH variable to point to your local mail spool. For example, the command:

```
bash> MAILPATH='/var/spool/mail/john'
bash> export MAILPATH
```

Causes bash to print a notification on john's console every time a new message is appended to John's mail spool.

6. Run tasks in the background

Bash lets you run one or more tasks in the background, and selectively suspend or resume any of the current tasks (or "jobs"). To run a task in the background, add an ampersand (&) to the end of its command line. Here's an example:

```
bash> tail -f /var/log/messages &
[1] 614
```

Each task backgrounded in this manner is assigned a job ID, which is printed to the console. A task can be brought back to the foreground with the command fg ***jobnumber***, where ***jobnumber*** is the job ID of the task you wish to bring to the foreground. Here's an example:

```
bash> fg 1
```

A list of active jobs can be obtained at any time by typing jobs at the bash prompt.

7. Quickly jump to frequently-used directories

You probably already know that the $PATH variable lists bash's "search path" -- the directories it will search when it can't find the requested file in the current directory. However, bash also supports the $CDPATH variable, which lists the directories the cd command will look in when attempting to change directories. To use this feature, assign a directory list to the $CDPATH variable, as shown in the example below:

```
bash> CDPATH='.:~:/usr/local/apache/htdocs:/disk1/backups'
bash> export CDPATH
```

Now, whenever you use the cd command, bash will check all the directories in the $CDPATH list for matches to the directory name.

8. Perform calculations

Bash can perform simple arithmetic operations at the command prompt. To use this feature, simply type in the arithmetic expression you wish to evaluate at the prompt within double parentheses, as illustrated below. Bash will attempt to perform the calculation and return the answer.

```
bash> echo $((16/2))
8
```

... ...

10. Get context-specific help

Bash comes with help for all built-in commands. To see a list of all built-in commands, type help. To obtain help on a specific command, type help ***command***, where ***command*** is the command you need help on. Here's an example:

```
bash> help alias
...some help text...
```

Obviously, you can obtain detailed help on the bash shell by typing man bash at your command prompt at any time.

[Mar 30, 2008] Bash tips and tricks " Richard's linux, web design and e-learning collection

```
# Bash tips and tricks for History related preferences
# see http://richbradshaw.wordpress.com/2007/11/25/bash-tips-and-tricks/

# == 1 Lost bash history ==
# the bash history is only saved when you close the terminal, not after each command. fix it..
shopt -s histappend
PROMPT_COMMAND='history -a'

# == 2. Stupid spelling mistakes ==
# This will make sure that spelling mistakes such as ect instead of etc are ignored.
shopt -s cdspell

# == 3. Duplicate entries in bash history ==
# This will ignore duplicates, as well as ls, bg, fg and exit as well, making for a cleaner bash history.
export HISTIGNORE="&:ls:[bf]g:exit"

# == 4 Multiple line commands split up in history ==
# this will change multiple line commands into single lines for easy editing.
shopt -s cmdhist
```

My Favorite bash Tips and Tricks

One thing you can do is redirect your output to a file. Basic output redirection should be nothing new to anyone who has spent a reasonable amount of time using any UNIX or Linux shell, so I won't go into detail regarding the basics of output redirection. To save the useful output from the find command, you can redirect the output to a file:

```
$ find /  -name foo > output.txt
```

You still see the error messages on the screen but not the path of the file you're looking for. Instead, that is placed in the file output.txt. When the find command completes, you can cat the file output.txt to get the location(s) of the file(s) you want.

That's an acceptable solution, but there's a better way. Instead of redirecting the standard output to a file, you can redirect the error messages to a file. This can be done by placing a 2 directly in front of the redirection angle bracket. If you are not interested in the error messages, you simply can send them to /dev/null:

This shows you the location of file foo, if it exists, without those pesky permission denied error messages. I almost always invoke the find command in this way.

The number 2 represents the standard error output stream. Standard error is where most commands send their error messages. Normal (non-error) output is sent to standard output, which can be represented by the number 1. Because most redirected output is the standard output, output redirection works only on the standard output stream by default. This makes the following two commands equivalent:

```
$ find / -name foo > output.txt
$ find / -name foo 1> output.txt
```

Sometimes you might want to save both the error messages and the standard output to file. This often is done with cron jobs, when you want to save all the output to a log file. This also can be done by directing both output streams to the same file:

```
$ find / -name foo > output.txt 2> output.txt
```

This works, but again, there's a better way to do it. You can tie the standard error stream to the standard output stream using an ampersand. Once you do this, the error messages goes to wherever you redirect the standard output:

```
$ find / -name foo > output.txt 2>&1
```

One caveat about doing this is that the tying operation goes at the end of the command generating the output. This is important if piping the output to another command. This line works as expected:

```
find -name test.sh 2>&1 | tee /tmp/output2.txt
```

but this line doesn't:

```
find -name test.sh | tee /tmp/output2.txt 2>&1
```

and neither does this one:

```
find -name test.sh 2>&1 > /tmp/output.txt
```

I started this discussion on output redirection using the find command as an example, and all the examples used the find command. This discussion isn't limited to the output of find, however. Many other commands can generate enough error messages to obscure the one or two lines of output you need.

Output redirection isn't limited to bash, either. All UNIX/Linux shells support output redirection using the same syntax.

Bash Tip #2 subprocess

Bash bang commands can be used for shortcuts too.

- !! = last line in history
- !* = all args from last line in history
- !$ = last arg from last line in history
- !^ = first arg from last line in history

I really only use !$ with the cd command. Here's some examples, although some not really useful. Just to give you an idea of what it does:

1. which php (maybe it outputs /usr/local/bin/php)
2. `!!` /path/to/php_script.php (executes php on the script)

Bash Tips and Tricks 'cd' with style

Something you may have seen before in other systems (the much maligned SCO OSes, for example) is this handy option:

shopt -s cdspell

"This will correct minor spelling errors in a '**cd**' command, so that instances of transposed characters, missing characters and extra characters are corrected without the need for retyping."

[Mar 20, 2008] bash Tricks From the Developers of the O'Reilly Network - O'Reilly ONLamp Blog

No more worrying about cases

The best bash tip I can share is very helpful when working on systems that don't allow filenames to differ only in case (like OSX and Windows):

create a file called .inputrc in your home directory and put this line in it:

```
set completion-ignore-case on
```

Now bash tab-completion won't worry about case in filenames. Thus 'cd sit[tab]' would complete to 'cd Sites/'

Last argument

You can also use Esc-period and get the last parm of the previous line. You can repeatedly use Esc-period to scroll back through time with them. That turns out to be even better than $! because you can edit it once it shows up on your command line.

should be !$
Instead of $!, use !$, it works much better. :)

```
$ echo asdf
asdf
$ echo !$
echo asdf
asdf
$ echo $!

$
```

So $! is an empty variable, while !$ brings back the last argument from the last command.

Command substitution
$ for s in `cat server.list`; do ssh $s uptime; done;

Command substution is also done using $(*command*) notation, which I prefer to the backquotes. It allows commands to be nested (backquotes allow that too, but the inner quotes must be escaped using backslashes, which gets messy.

For example:

```
$ for s in $(cat server.list); do echo "$s: $(ssh $s uptime)"; done;
```

or:

```
# get the uptime for just the first server
$ echo "$(date): $(ssh $(head -1 server.list) uptime)"
```

=====

More key bindings and tricks
Bash will keep a history of the directories you visit, you just have to ask.

You can also always go back to the previous directory you were in by typing `cd -` without the need to `pushd` the current directory. Using it more than once cycles between the current and previous directory.

CTRL-A takes you to the beginning of the line and CTRL-E takes you to the end of the line. This is probably basic shell knowledge,

I think it's actually common readline/emacs knowledge, and it works in much more programs than just Bash or a terminal. For instance, you can enable them in Gnome applications by adding the line
`gtk-key-theme-name = "Emacs"` to the `~/.gtkrc-2.0` file.

Other handy key bindings you can use are:

- `ctrl-u` : Cut everything on the current line before the cursor.
- `ctrl-y` : 'Yank' (paste) text that was cut using `ctrl-u`.
- `ctrl-w` : Delete the word on the left of the cursor

There's so much usefull knowledge hidden in Bash that, if you spend any time at the command line, you should really get yourself aquinted with. It saves incredible ammounts of time.

Take for example something I wanted to do yesterday. I wanted to now the number of hits on a certain website. I could have installed a tool to parse the Apache access.log, but this was much easier:

```
$ cat access.log | cut -d"[" -f2 | cut -d"]" -f1 | cut -d"/" -f2 | uniq -c
28905 Mar
16554 Apr
```

Takes no more than a couple of seconds to write, but saves so much time.

Try reading through the Bash man page. It's huge, but think of all the stuff you'll learn! Or read some online Bash scripting tutorials. Everything from gathering statistics from files to creating thumbnails of images (From the top of my head: `for A in *; do convert $A -resize 140x140 th_$A; done`) becomes a cinch.

BASH Help - A Bash Tutorial

Flip the Last Two Characters

If you type like me your fingers spit characters out in the wrong order on occasion. *ctrl-t* swaps the order that the last two character appear in.

Searching Bash History

As you enter commands at the CLI they are saved in a file ~./.bash_history. From the bash prompt you can browse the most recently used commands through the least recently used commands by pressing the up arrow. Pressing the down arrow does the opposite.

If you have entered a command a long time ago and need to execute it again you can search for it. Type the command 'ctrl-r' and enter the text you want to search for.

[Dec 9, 2007] Cool Solutions Bash - Making use of your .bashrc file

Good sample bashrc file

This Is Your Open Enterprise™
Skip to Content United States - EnglishNovell Home LoginDownload
Products & Solutions
Services & Support
Partners & Communities
Search
Advanced Search
SolutionsIdentity, Security and Systems ManagementLinux Operating SystemsWorkgroup CollaborationProducts forIndustriesSmall BusinessProducts A-ZServi
> cool solutions home > cool tools home
Bash - Making use of your .bashrc file
Novell Cool Solutions: Cool ToolRate This Page
Reader Rating from 4 ratings

Printer Friendlytell a friendDigg This - Slashdot This

In Brief
A sample .bashrc file.

Vitals
Product Categories:
Open Enterprise Server
SUSE Linux
SUSE Linux Enterprise Desktop
SUSE Linux Enterprise Server
Functional Categories:
BASH
Shortcuts
Workgroup
Updated: 23 Oct 2006
File Size: 6.9KB
License: GPL
Download: /coolsolutions/tools/downloads/bashrc.txt
Publisher: David Crouse

Disclaimer
Please read the note from our friends in legal before using this file.

Details

I was playing with my .bashrc file again, and was once again impressed by how you can tweak Linux to do what YOU want it to do so easily. I am sure there are

For those that don't know what a .bashrc file does: "The ~/.bashrc file determines the behavior of interactive shells." Quoted From: The Advanced Bash Scripting

Basically , it allows you to create shortcuts (alias's) and interactive programs (functions) that run on the startup of the bash shell or that are used when running an

I have my .bashrc file setup in sections. The following is the breakdown by section of how I keep my list of alias's and functions separated. This is just how I do

Header (So I know when i modified it last and what i was running it on)
Exports (So I can set history size, paths , editors, define colors, etc,)
Sourced Alias's (So I can find those hidden alias's faster)
Workstation Alias's (so i can ssh to local machines quickly)
Remote Server Alias's (so i can ssh to remote servers easily)
Script Alias's (quick links to some of my bashscripts)
Hardware control alias's (so I can control cd/dvd/scanners/audio/etc)
Modified commands (Alias's to normal linux commands with special flags)
Chmod Alias's (makes changing permissions faster)

Alias's for GUI programs (start firefox, etc from command line)
Alias's for xterm and others (open xterm with special settings)
Alias's for Lynx (open lynx with urls - kind of a bash bookmark ;))
UNused Alias's (Alias's that aren't in use on the system, but that i might use later)
Special functions (more of a function than just an alias..it goes here)
Notes (that should be self explanatory ;))
Welcome Screen (code to make my bash shell display some stuff as it starts up)

That's how I lay out my .bashrc files. It may not be perfect, but it works well for me. I like making changes in just my .bashrc file and not the global files. I like t

The following is my .bashrc file (with some things obviously commented out for security... but most of it should be self explanatory). Anyone with comments/su

Want to know what alias's your bash shell has? Simply type the word alias at the command line. The shell will then print out the list of active alias's to the standa

```
#########################################################
# Dave Crouse's .bashrc file
# www.bashscripts.org
# www.usalug.org
#
# Last Modified 04-08-2006
# Running on OpenSUSE 10
#########################################################


# EXPORTS
#########################################################

PATH=$PATH:/usr/lib/festival/ ;export PATH
export PS1="[\[\033[1;34m\w\[\033[0m]\n[\t \u]$ "
export EDITOR=/usr/bin/pico
export HISTFILESIZE=3000 # the bash history should save 3000 commands
export HISTCONTROL=ignoredups #don't put duplicate lines in the history.
alias hist='history | grep $1' #Requires one input

# Define a few Color's
BLACK='\e[0;30m'
BLUE='\e[0;34m'
GREEN='\e[0;32m'
CYAN='\e[0;36m'
RED='\e[0;31m'
PURPLE='\e[0;35m'
BROWN='\e[0;33m'
LIGHTGRAY='\e[0;37m'
DARKGRAY='\e[1;30m'
LIGHTBLUE='\e[1;34m'
LIGHTGREEN='\e[1;32m'
LIGHTCYAN='\e[1;36m'
LIGHTRED='\e[1;31m'
LIGHTPURPLE='\e[1;35m'
YELLOW='\e[1;33m'
WHITE='\e[1;37m'
NC='\e[0m'              # No Color
# Sample Command using color: echo -e "${CYAN}This is BASH
${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on ${RED}$DISPLAY${NC}\n"


# SOURCED ALIAS'S AND SCRIPTS
#########################################################

### Begin insertion of bbips alias's ###
source ~/.bbips/commandline/bbipsbashrc
### END bbips alias's ###

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# enable programmable completion features
if [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi


# ALIAS'S OF ALL TYPES SHAPES AND FORMS ;)
#########################################################

# Alias's to local workstations
alias tom='ssh 192.168.2.102 -l root'
alias jason='ssh 192.168.2.103 -l root'
alias randy='ssh 192.168.2.104 -l root'
alias bob='ssh 192.168.2.105 -l root'
alias don='ssh 192.168.2.106 -l root'
alias counter='ssh 192.168.2.107 -l root'

# ALIAS TO REMOTE SERVERS
alias ANYNAMEHERE='ssh YOURWEBSITE.com -l USERNAME -p PORTNUMBERHERE'
```

```
# My server info removed from above for obvious reasons ;)

# Alias's to TN5250 programs. AS400 access commands.
alias d1='xt5250 env.TERM = IBM-3477-FC env.DEVNAME=D1 192.168.2.5 &'
alias d2='xt5250 env.TERM = IBM-3477-FC env.DEVNAME=D2 192.168.2.5 &'
alias tn5250j='nohup java -jar /home/crouse/tn5250j/lib/tn5250j.jar
2>>error.log &'

# Alias's to some of my BashScripts
alias bics='sh /home/crouse/scripts/bics/bics.sh'
alias backup='sh /home/crouse/scripts/usalugbackup.sh'
alias calc='sh /home/crouse/scripts/bashcalc.sh'
alias makepdf='sh /home/crouse/scripts/makepdf.sh'
alias phonebook='sh /home/crouse/scripts/PHONEBOOK/baps.sh'
alias pb='sh /home/crouse/scripts/PHONEBOOK/baps.sh'
alias ppe='/home/crouse/scripts/passphraseencryption.sh'
alias scripts='cd /home/crouse/scripts'

# Alias's to control hardware
alias cdo='eject /dev/cdrecorder'
alias cdc='eject -t /dev/cdrecorder'
alias dvdo='eject /dev/dvd'
alias dvdc='eject -t /dev/dvd'
alias scan='scanimage -L'
alias playw='for i in *.wav; do play $i; done'
alias playo='for i in *.ogg; do play $i; done'
alias playm='for i in *.mp3; do play $i; done'
alias copydisk='dd if=/dev/dvd of=/dev/cdrecorder' # Copies bit by bit
from dvd to cdrecorder drives.
alias dvdrip='vobcopy -i /dev/dvd/ -o ~/DVDs/ -l'

# Alias's to modified commands
alias ps='ps auxf'
alias home='cd ~'
alias pg='ps aux | grep'  #requires an argument
alias un='tar -zxvf'
alias mountedinfo='df -hT'
alias ping='ping -c 10'
alias openports='netstat -nape --inet'
alias ns='netstat -alnp --protocol=inet | grep -v CLOSE_WAIT | cut
-c-6,21-94 | tail +2'
alias du1='du -h --max-depth=1'
alias da='date "+%Y-%m-%d %A    %T %Z"'
alias ebrc='pico ~/.bashrc'

# Alias to multiple ls commands
alias la='ls -Al'               # show hidden files
alias ls='ls -aF --color=always' # add colors and file type extensions
alias lx='ls -lXB'              # sort by extension
alias lk='ls -lSr'              # sort by size
alias lc='ls -lcr'      # sort by change time
alias lu='ls -lur'      # sort by access time
alias lr='ls -lR'               # recursive ls
alias lt='ls -ltr'              # sort by date
alias lm='ls -al |more'         # pipe through 'more'

# Alias chmod commands
alias mx='chmod a+x'
alias 000='chmod 000'
alias 644='chmod 644'
alias 755='chmod 755'

# Alias Shortcuts to graphical programs.
alias kwrite='kwrite 2>/dev/null &'
alias firefox='firefox 2>/dev/null &'
alias gaim='gaim 2>/dev/null &'
alias kate='kate 2>/dev/null &'
alias suk='kdesu konqueror 2>/dev/null &'

# Alias xterm and aterm
alias term='xterm -bg AntiqueWhite -fg Black &'
alias termb='xterm -bg AntiqueWhite -fg NavyBlue &'
alias termg='xterm -bg AntiqueWhite -fg OliveDrab &'
alias termr='xterm -bg AntiqueWhite -fg DarkRed &'
alias aterm='aterm -ls -fg gray -bg black'
alias xtop='xterm -fn 6x13 -bg LightSlateGray -fg black -e top &'
alias xsu='xterm -fn 7x14 -bg DarkOrange4 -fg white -e su &'

# Alias for lynx web browser
alias bbc='lynx -term=vt100 http://news.bbc.co.uk/text_only.stm'
alias nytimes='lynx -term=vt100 http://nytimes.com'
alias dmregister='lynx -term=vt100 http://desmoinesregister.com'


# SOME OF MY UNUSED ALIAS's
##########################################################
```

```
# alias d=`echo "Good Morning Dave. today's date is" | festival --tts;
date +'%A %B %e' | festival --tts`
# alias shrink84='/home/crouse/shrink84/shrink84.sh'
# alias tl='tail -f /var/log/apache/access.log'
# alias te='tail -f /var/log/apache/error.log'


# SPECIAL FUNCTIONS
#########################################################

netinfo ()
{
echo "--------------- Network Information ---------------"
/sbin/ifconfig | awk /'inet addr/ {print $2}'
echo ""
/sbin/ifconfig | awk /'Bcast/ {print $3}'
echo ""
/sbin/ifconfig | awk /'inet addr/ {print $4}'

# /sbin/ifconfig | awk /'HWaddr/ {print $4,$5}'
echo "-------------------------------------------------------"
}

spin ()
{
echo -ne "${RED}-"
echo -ne "${WHITE}\b|"
echo -ne "${BLUE}\bx"
sleep .02
echo -ne "${RED}\b+${NC}"
}

scpsend ()
{
scp -P PORTNUMBERHERE "$@"
USERNAME@YOURWEBSITE.com:/var/www/html/pathtodirectoryonremoteserver/;
}


# NOTES
#########################################################

# To temporarily bypass an alias, we preceed the command with a \
# EG:  the ls command is aliased, but to use the normal ls command you would
# type \ls

# mount -o loop /home/crouse/NAMEOFISO.iso /home/crouse/ISOMOUNTDIR/
# umount /home/crouse/NAMEOFISO.iso
# Both commands done as root only.


# WELCOME SCREEN
#########################################################

clear
for i in `seq 1 15` ; do spin; done ;echo -ne "${WHITE} USA Linux Users
Group ${NC}"; for i in `seq 1 15` ; do spin; done ;echo "";
echo -e ${LIGHTBLUE}`cat /etc/SUSE-release` ;
echo -e "Kernel Information: " `uname -smr`;
echo -e ${LIGHTBLUE}`bash --version`;echo ""
echo -ne "Hello $USER today is "; date
echo -e "${WHITE}"; cal ; echo "";
echo -ne "${CYAN}";netinfo;
mountedinfo ; echo ""
echo -ne "${LIGHTBLUE}Uptime for this computer is ";uptime | awk /'up/
{print $3,$4}'
for i in `seq 1 15` ; do spin; done ;echo -ne "${WHITE} http://usalug.org
${NC}"; for i in `seq 1 15` ; do spin; done ;echo "";
echo ""; echo ""The following belong under the "function" section in my .bashrc. Useable as seperate programs, I've integrated them simply as functions for my

Requires: zenity, gpg

################### Begin gpg functions ##################
encrypt ()
{
# Use ascii armor
gpg -ac --no-options "$1"
}

bencrypt ()
{
# No ascii armor
# Encrypt binary data. jpegs/gifs/vobs/etc.
gpg -c --no-options "$1"
}

decrypt ()
```

```
{
gpg --no-options "$1"
}

pe ()
{
# Passphrase encryption program
# Created by Dave Crouse 01-13-2006
# Reads input from text editor and encrypts to screen.
clear
echo "         Passphrase Encryption Program";
echo "---------------------------------------------------"; echo "";
which $EDITOR &>/dev/null
 if [ $? != "0" ];
    then
    echo "It appears that you do not have a text editor set in your
.bashrc file.";
    echo "What editor would you like to use ? " ;
    read EDITOR ; echo "";
 fi
echo "Enter the name/comment for this message :"
read comment
$EDITOR passphraseencryption
gpg --armor --comment "$comment" --no-options --output
passphraseencryption.gpg --symmetric passphraseencryption
shred -u passphraseencryption ; clear
echo "Outputting passphrase encrypted message"; echo "" ; echo "" ;
cat passphraseencryption.gpg ; echo "" ; echo "" ;
shred -u passphraseencryption.gpg ;
read -p "Hit enter to exit" temp; clear
}

keys ()
{
# Opens up kgpg keymanager
kgpg -k
}

encryptfile ()
{
zenity --title="zcrypt: Select a file to encrypt" --file-selection > zcrypt
encryptthisfile=`cat zcrypt`;rm zcrypt
# Use ascii armor
#  --no-options (for NO gui usage)
gpg -acq --yes ${encryptthisfile}
zenity --info --title "File Encrypted" --text "$encryptthisfile has been
encrypted"
}

decryptfile ()
{
zenity --title="zcrypt: Select a file to decrypt" --file-selection > zcrypt
decryptthisfile=`cat zcrypt`;rm zcrypt
# NOTE: This will OVERWRITE existing files with the same name !!!
gpg --yes -q ${decryptthisfile}
zenity --info --title "File Decrypted" --text "$encryptthisfile has been
decrypted"
}

##################### End gpg functions ##################
Novell Cool Solutions (corporate web communities) are produced by WebWise Solutions. www.webwiseone.com

Reader Comments
cool man, really cool. i love such stuffs you know. working in the command line makes you feel like a real linux geek
it's really cool. good job.

AuthorsDocumentationGlossaryKnowledgebaseNovell ConnectionPartner Product GuideSupport ForumsTrainingAppNotes by DateAppNotes by TitleSubmit a
1.800.529.3400 local numbers
Request Call
Corporate Governance | Legal & Export | Privacy | Subscribe | Feedback | Glossary | RSS | Contact | Printer Friendly
© 2007 Novell, Inc. All Rights Reserved.
```

xargs, find and several useful shortcuts

See also Unix Xargs and Unix Find Command pages.

Re:pushd and popd (and other tricks) (Score:2)
by Ramses0 (63476) on Wednesday March 10, @07:39PM (#8527252) My favorite "Nifty" was when I spent the time to learn about "xargs" (I pronounce it zargs), and brush up on "for" syntax.

```
ls | xargs -n 1 echo "ZZZ> "
```

Basically indents (prefixes) everything with a "ZZZ" string. Not really useful, right? But since it invokes the echo command (or whatever command you specify) $n times (where $n is the number of lines passed to it) this saves me from having to write a lot of crappy little shell scripts sometimes.

A more serious example is:

```
find -name \*.jsp | sed 's/^/http:\/\/127.0.0.1/server/g' | xargs -n 1 wget
```

...will find all your jsp's, map them to your localhost webserver, and invoke a wget (fetch) on them. Viola, precompiled JSP's.

Another:

```
for f in `find -name \*.jsp` ; do echo "==> $f" >> out.txt ; grep "TODO" $f >> out.txt ; done
```

...this searches JSP's for "TODO" lines and appends them all to a file with a header showing what file they came from (yeah, I know grep can do this, but it's an example. What if grep couldn't?) ...and finally...

```
( echo "These were the command line params"
echo "---------"
for f in $@ ; do
echo "Param: $f"
done ) | mail -s "List" you@you.com
```

...the parenthesis let your build up lists of things (like interestingly formatted text) and it gets returned as a chunk, ready to be passed on to some other shell processing function.

Shell scripting has saved me a lot of time in my life, which I am grateful for. :^)

[May 7, 2007] To strip file extensions in bash, like this.rbl --> this

```
fname=${file%.rbl}
```

Last argument reuse

```
tail -f /tmp/foo
rm !$  # !$ is the last argument to the previous command.
```

Correction sed style

```
grep 'wibble' afile | lwss  #typo: meant to type less
!!:s/lw/le #!! is last command string, :s does sed-style  modification. :gs does a global replace

# or for simpler corrections
# n.b. textile screws this up. replace the sup elements with circumflexes.
cat .bash_profilx #typo - meant the x to be an e
<sup>x</sup>e # 'repeat last command, subsituting x for e

touch a{1,2,3,4}b # brace gets expanded to a1b a2b a3b a4b so 4 files get touched

cp file{,.old} # brace gets expanded to file file.old , thus creating a backup.
```

- shell variables

$CDPATH
This is a little known and very underrated shell variable. *CDPATH* does for the **cd** built-in what *PATH* does for executables. By setting this wisely, you can cut down on the number of key-strokes you enter per day.

Try this:

```
$ export CDPATH=.:~:~/docs:~/src:~/src/ops/docs:/mnt:/usr/src/redhat:/usr/src/redhat/RPMS:/usr/src:/usr/lib:/usr/local:/software:/software/redhat
```

Using this, cd i386 would likely take you to /usr/src/redhat/RPMS/i386 on a Red Hat Linux system.

$HISTIGNORE
Set this to to avoid having consecutive duplicate commands and other not so useful information appended to the history list. This will cut down on hitting the up arrow endlessly to get to the command before the one you just entered twenty times. It will also avoid filling a large percentage of your history list with useless commands.

Try this:

```
$ export HISTIGNORE="&:ls:ls *:mutt:[bf]g:exit"
```

Using this, consecutive duplicate commands, invocations of **ls**, executions of the mutt mail client without any additional parameters, plus calls to the **bg**, **fg** and **exit** built-ins will not be appended to the history list.

$MAILPATH
bash will warn you of new mail in any folder appended to *MAILPATH*. This is very handy if you use a tool like procmail to presort your e-mail into folders.

Try adding the following to your ~/.bash_profile to be notified when any new mail is deposited in any mailbox under ~/Mail.

```
MAILPATH=/var/spool/mail/$USER
for i in `echo ~/Mail/[^.]*`
do
      MAILPATH=$MAILPATH:$i
done
export MAILPATH
unset i
```

If you use mutt and many of those folders don't receive automatically filtered mail, you may prefer to have bash alert you only when new e-mail arrives in a folder that you also track in **mutt**.

In that case, try something like the following in your ~/.bash_profile:

```
export `perl -ne 's/^mailboxes /MAILPATH=/ && tr/ /:/ && print && exit' < ~/.muttrc`
```

$TMOUT

If you set this to a value greater than zero, bash will terminate after this number of seconds have elapsed if no input arrives.

This setting is useful in root's environment to reduce the potential security risk of someone forgetting to log out as the superuser.

- **set** options

ignoreeof
Ordinarily, issuing Ctrl-D at the prompt will log you out of an interactive shell. This can be annoying if you regularly need to type Ctrl-D in other situations, for example, when trying to disconnect from a Telnet session. In such a situation, hitting Ctrl-D once too often will close your shell, which can be very frustrating. This option disables the use of Ctrl-D to exit the shell.

- **shopt** options

You can set each of the options below with shopt -s <option>.

cdspell
This will correct minor spelling errors in a **cd** command, so that instances of transposed characters, missing characters and extra characters are corrected without the need for retyping.
cmdhist
This is very much a matter of taste. Defining this will cause multi-line commands to be appended to your bash history as a single line command. This makes for easy command editing.
dotglob
This one allows files beginning with a dot ('.') to be returned in the results of path-name expansion.
extglob

This will give you ksh-88 egrep-style extended pattern matching or, in other words, turbo-charged pattern matching within bash. The available operators are:

?(*pattern-list*)
Matches zero or one occurrence of the given patterns
***(*pattern-list*)**
Matches zero or more occurrences of the given patterns
+(*pattern-list*)
Matches one or more occurrences of the given patterns
@(*pattern-list*)
Matches exactly one of the given patterns
!(*pattern-list*)
Matches anything except one of the given patterns

Here's an example. Say, you wanted to install all RPMs in a given directory, except those built for the noarch architecture. You might use something like this:

rpm -Uvh /usr/src/RPMS/!(*noarch*)

These expressions can be nested, too, so if you wanted a directory listing of all non PDF and PostScript files in the current directory, you might do this:

ls -lad !(*.p?(df|s))

readline Tips and Tricks

The readline library is used by bash and many other programs to read a line from the terminal, allowing the user to edit the line with standard Emacs editing keys.

- set show-all-if-ambiguous on

If you have this in your /etc/inputrc or ~/.inputrc, you will no longer have to hit the <Tab> key twice to produce a list of all possible completions. A single <Tab> will suffice. This setting is highly recommended.

- set visible-stats on

Adding this to your /etc/inputrc or ~/.inputrc will result in a character being appended to any file-names returned by completion, in much the same way as ls -F works.

- If you're a fan of vi as opposed to Emacs, you might prefer to operate bash in vi editing mode. Being a GNU program, bash uses Emacs bindings unless you specify otherwise.

Set the following in your /etc/inputrc or ~/.inputrc:

set editing-mode vi
set keymap vi

and this in your /etc/bashrc or ~/.bashrc:

set -o vi

Set Vi Mode in Bash

```
# set -o vi
```

Vi mode allows for the use of vi like commands when at the bash prompt. When set to this mode initially you will be in insert mode (be able to type at the prompt unlike when you enter vi). Hitting the *escape* key takes you into command mode.

Commands to take advantage of bash's Vi Mode:

h	Move cursor left
l	Move cursor right
A	Move cursor to end of line and put in insert mode

0	(zero) Move cursor to beginning of line (doesn't put in insert mode)
i	Put into insert mode at current position
a	Put into insert mode after current position
dd	Delete line (saved for pasting)
D	Delete text after current cursor position (saved for pasting)
p	Paste text that was deleted
j	Move up through history commands
k	Move down through history commands
u	Undo

Useful Commands and Features

The commands in this section are non-mode specific, unlike the ones listed above.

Flip the Last Two Characters

If you type like me your fingers spit characters out in the wrong order on occasion. *ctrl-t* swaps the order that the last two character appear in.

Searching Bash History

As you enter commands at the CLI they are saved in a file ~./.bash_history. From the bash prompt you can browse the most recently used commands through the least recently used commands by pressing the up arrow. Pressing the down arrow does the opposite.

If you have entered a command a long time ago and need to execute it again you can search for it. Type the command 'ctrl-r' and enter the text you want to search for.

Dealing with Spaces

First, I will mention a few ways to deal with spaces in directory names, file names, and everywhere else.

Using the Backslash Escape Sequence

One option is to use bash's escape character \. Any space following the backslash is treated as being part of the same string. These commands create a directory called "foo bar" and then remove it.

```
# mkdir foo\ bar
# rm foo\ bar
```

The backslash escape sequence can also be used to decode commands embedded in strings which can be very useful for scripting or modifying the command prompt as discussed later.

Using Single/Double Quotes with Spaces and Variables

Single and double quotes can also be used for dealing with spaces.

```
# touch 'dog poo'
# rm "dog poo"
```

The difference between single and double quotes being that in double quotes the **$**, \, and **'** characters still preserve their special meanings. Single quotes will take the **$** and \ literally and regard the **'** as the end of the string. Here's an example:

```
# MY_VAR='This is my text'
# echo $MY_VAR
This is my text
# echo "$MY_VAR"
This is my text
# echo '$MY_VAR'
$MY_VAR
```

The string following the **$** character is interpreted as being a variable except when enclosed in single quotes as shown above.

Lists Using { and }

The characters **{** and **}** allow for list creation. In other words you can have a command be executed on each item in the list. This is perhaps best explained with examples:

```
# touch {temp1,temp2,temp3,temp4}
```

This will create/modify the files temp1, temp2, temp3, and temp4 and as in the example above when the files share common parts of the name you can do:

```
# mv temp{1,2,3,4} ./foo\ bar/
```

This will move all four of the files into a directory 'foo bar'.

Executing Multiple Commands in Sequence

This is a hefty title for a simple task. Consider that you want to run three commands, one right after the other, and you do not want to wait for each to finish before typing the next. You can type all three commands on a line and then start the process:

```
# ./configure; make; make install
OR
# ./configure && make && make install
```

I often use these formats in crontab files for commands that need to be executed in sequence if I choose not to make a script.

Piping Output from One Command to Another

Piping allows the user to do several fantastic thing by combining utilities. I will cover only very basic uses for piping. I most commonly use the pipe command, |, to pipe text that is outputted from one command through the **grep** command to search for text.

Examples:

See if a program, centericq, is running:

```
# ps ax | grep centericq
25824 pts/2 S 0:18 centericq
```

Count the number of files in a directory (nl counts things):

```
# ls | nl
1 #.emacs#
2 BitchX
3 Outcast double cd.lst
4 bm.shader
5 bmtexturesbase.pk3
```

If my memory serves using RPM to check if a package is installed:

```
# rpm -qa | grep package_name
```

A more advance example:

```
# cat /etc/passwd | awk -F: '{print $1 "\t" $6}' | sort > ./users
```

This sequence takes the information if the file passwd, pipes it to awk, which takes the first and sixth fields (the user name and home directory respectively), pipes these fields separated by a tab ("\t") to sort, which sorts the list alphabetically, and puts it into a file called users.

Aliasing Commands

Once again I like how this topic is covered on freeunix.dyndns.org:8088 in "Customizing your Bash environment" I will quote the section entitled "Aliasses":

If you have used UNIX for a while, you will know that there are many commands available and that some of them have very cryptic names and/or can be invoked with a truckload of options and arguments. So, it would be nice to have a feature allowing you to rename these commands or type something simple instead of a list of options. Bash provides such a feature : the ***alias*** .
Aliasses can be defined on the command line, in *.bash_profile*, or in *.bashrc*, using this form :

```
alias name=command
```

This means that **name** is an alias for **command**. Whenever name is typed as a command, Bash will substitute command in its place. Note that there are no spaces on either side of the equal sign. Quotes around command are necessary if the string being aliassed consists of more than one word. A few examples :

```
alias ls=&#39;ls -aF --color=always&#39; alias ll=&#39;ls -l&#39; alias search=grep alias mcd=&#39;mount /mnt/cdrom&#39; alias ucd=&#39;umount
            /mnt/cdrom&#39; alias mc=&#39;mc -c&#39; alias ..=&#39;cd ..&#39; alias ...=&#39;cd ../..&#39;
```

The first example ensures that ls always uses color if availabe, that dotfiles are listed as well,that directories are marked with a / and executables with a *. To make ls do the same on FreeBSD, the alias would become :

```
alias ls=&#39;/bin/ls -aFG&#39;
```

To see what aliasses are currently active, simply type **alias** at the command prompt and all active aliasses will be listed. To "disable" an alias type **unalias** followed by the alias name.

Altering the Command Prompt Look and Information

Bash has the ability to change how the command prompt is displayed in information as well as colour. This is done by setting the PS1 variable. There is also a PS2 variable. It controls what is displayed after a second line of prompt is added and is usually by default '> '. The PS1 variable is usually set to show some useful information by the Linux distribution you are running but you may want to earn style points by doing your own modifications.

Here are the backslash-escape special characters that have meaning to bash:

```
\a    an ASCII bell character (07)
\d    the date in "Weekday Month Date" format
      (e.g., "Tue May 26")
\e    an ASCII escape character (033)
\h    the hostname up to the first `.'
\H    the hostname
\j    the number of jobs currently managed by the shell
\l    the basename of the shell's terminal device name
\n    newline
\r    carriage return
\s    the name of the shell, the basename of $0
      (the portion following the final slash)
\t    the current time in 24-hour HH:MM:SS format
\T    the current time in 12-hour HH:MM:SS format
\@    the current time in 12-hour am/pm format
\u    the username of the current user
\v    the version of bash (e.g., 2.00)
\V    the release of bash, version + patchlevel
      (e.g., 2.00.0)
\w    the current working directory
\W    the basename of the current working direcory
```

```
\!    the history number of this command
\#    the command number of this command
\$    if the effective UID is 0, a #, otherwise a $
\nnn  the character corresponding to the octal number nnn
\\    a backslash
\[    begin a sequence of non-printing characters,
      which could be used to embed a terminal control
      sequence into the prompt
\]    end a sequence of non-printing characters
```

Colours In Bash:

Black	0;30	Dark Gray	1;30
Blue	0;34	Light Blue	1;34
Green	0;32	Light Green	1;32
Cyan	0;36	Light Cyan	1;36
Red	0;31	Light Red	1;31
Purple	0;35	Light Purple	1;35
Brown	0;33	Yellow	1;33
Light Gray	0;37	White	1;37

Here is an example borrowed from the Bash-Prompt-HOWTO:

```
PS1="\[\033[1;34m\][\$(date +%H%M)][\u@\h:\w]$\[\033[0m\] "
```

This turns the text blue, displays the time in brackets (very useful for not losing track of time while working), and displays the user name, host, and current directory enclosed in brackets. The "\[\033[0m\]" following the $ returns the colour to the previous foreground colour.

How about command prompt modification thats a bit more "pretty":

```
PS1="\[\033[1;30m\][\[\033[1;34m\]\u\[\033[1;30m\]@\[\033[0;35m\]\h\[\033[1;30m\]] \[\033[0;37m\]\W \[\033[1;30m\]\$\
[\033[0m\] "
```

This one sets up a prompt like this: [user@host] directory $

Break down:

```
\[\033[1;30m\] - Sets the color for the characters that follow it. Here 1;30 will set them to Dark Gray.
\u \h \W \$ - Look to the table above
\[033[0m\] - Sets the colours back to how they were originally.
```

Each user on a system can have their own customized prompt by setting the PS1 variable in either the .bashrc or .profile files located in their home directories.

FUN STUFF!

A quick note about bashish. It allows for adding themes to a terminal running under a GUI. Check out the site for some screen-shots of what it can do.

Also, the program **fortune** is a must [At least I have considered it so every since my Slackware days (since Slackware included it by default)]. It doesn't have anything to do with bash and is a program that outputs a quote to the screen. Several add-ons are available to make it say stuff about programming, the xfiles, futurama, starwars, and more. Just add a line in your /etc/profile like this to brighten your day when you log into your computer:

```
echo;fortune;echo
```

Basic and Extended Bash Completion

Basic Bash Completion will work in any bash shell. It allows for completion of:

1. File Names
2. Directory Names
3. Executable Names
4. User Names (when they are prefixed with a ~)
5. Host Names (when they are prefixed with a @)
6. Variable Names (when they are prefixed with a $)

This is done simply by pressing the tab key after enough of the word you are trying to complete has been typed in. If when hitting tab the word is not completed there are probably multiple possibilities for the completion. Press tab again and it will list the possibilities. Sometimes on my machine I have to hit it a third time.

Extended Programmable Bash Completion is a program that you can install to complete much more than the names of the things listed above. With extended bash completion you can, for example, complete the name of a computer you are trying to connect to with **ssh** or **scp**. It achieves this by looking through the known_hosts file and using the hosts listed there for the completion. This is greatly customizable and the package and more information can be found here.

Configuration of Programmable Bash Completion is done in **/etc/bash_completion**. Here is a list of completions that are in my **bash_completion** file by default.

- completes on signal names
- completes on network interfaces
- expands tildes in pathnames
- completes on process IDs
- completes on process group IDs
- completes on user IDs
- mysqladmin(1) completion
- gzip(1) completion
- bzip2(1) completion
- openssl(1) completion
- screen(1) completion
- lftp(1) bookmark completion

- completes on group IDs
- ifconfig(8) and iwconfig(8) helper function
- bash alias completion
- bash export completion
- bash shell function completion
- bash complete completion
- service completion
- chown(1) completion
- chgrp(1) completion
- umount(8) completion
- mount(8) completion
- Linux rmmod(8) completion
- Linux insmod(8), modprobe(8) and modinfo(8) completion
- man(1) completion
- renice(8) completion
- kill(1) completion
- Linux and FreeBSD killall(1) completion
- GNU find(1) completion
- Linux ifconfig(8) completion
- Linux iwconfig(8) completion
- RedHat & Debian GNU/Linux if{up,down} completion
- Linux ipsec(8) completion (for FreeS/WAN)
- Postfix completion
- cvs(1) completion
- rpm completion
- apt-get(8) completion
- chsh(1) completion
- chkconfig(8) completion
- user@host completion
- host completion based on ssh's known_hosts
- ssh(1) completion
- scp(1) completion
- rsync(1) completion
- Linux route(8) completion
- GNU make(1) completion
- GNU tar(1) completion
- jar(1) completion
- Linux iptables(8) completion
- tcpdump(8) completion
- autorpm(8) completion
- ant(1) completion
- ncftp(1) bookmark completion
- gdb(1) completion
- Postgresql completion
- psql(1) completion
- createdb(1) completion
- dropdb(1) completion
- gcc(1) completion
- Linux cardctl(8) completion
- Debian dpkg(8) completion
- Debian GNU dpkg-reconfigure(8) completion
- Debian Linux dselect(8) completion
- Java completion
- PINE address-book completion
- mutt completion
- Debian reportbug(1) completion
- Debian querybts(1) completion
- update-alternatives completion
- Python completion
- Perl completion
- rcs(1) completion
- lilo(8) completion
- links completion
- FreeBSD package management tool completion
- FreeBSD kernel module commands
- FreeBSD portupgrade completion
- FreeBSD portinstall completion
- Slackware Linux removepkg completion
- look(1) completion
- ypcat(1) and ypmatch(1) completion
- mplayer(1) completion
- KDE dcop completion
- wvdial(1) completion
- gpg(1) completion
- iconv(1) completion
- dict(1) completion
- cdrecord(1) completion
- mkisofs(8) completion
- mc(1) completion
- yum(8) completion
- yum-arch(8) completion
- ImageMagick completion

Links

1. Bash Prompt HOWTO
2. Bash Reference Manual
3. Customizing your Bash environment
4. Working more productively with bash 2.x
5. Advancing in the Bash Shell
6. Bash - Bourne Again SHell
7. What's GNU: Bash - The GNU Shell
8. Bash Tips in Gentoo Forums
9. bash(1) - Linux man page
10. BASHISH

Learn About Bash Scripting:

1. Bash by example, Part 1
2. Bash by example, Part 2
3. Bash by example, Part 3
4. Advanced Bash-Scripting Guide
5. A quick guide to writing scripts using the bash shell

unixtips.org bash tips

bash — Nicolas Lidzborski at 19 February, 23:54:09

If you want your *xterm* or *rxvt* tille bar to show the username, hostname and current directory and if you uses bash, you can set the PROMPT_COMMAND shell variable. Personally, I use the following command in my /etc/profile:

```
if [ $TERM = "xterm" ]; then
export PROMPT_COMMAND='echo -ne \
"\033]0;${USER}@${HOSTNAME}: ${PWD}\007"'
fi
```

The test around the export command is done in order to avoid causing problems in text terms.

bash — sRp at 19 February, 05:23:23

You can execute bash command a certain number of times by using something similar to the following:

```
n=0;while test -$n -gt -10; do echo n=$n; n=$[$n+1]; done
```

That code will print "n=0", "n=1", and so on 10 times.

bash — sRp at 30 January, 07:18:30

You can use *CTRL-_* or *CTRL-X, CTRL-U* to make undo's at the bash prompt.

bash — Ian Eure at 29 January, 12:55:02

Bash supports tab-completion. That is, you type the first few characters of a command (or file / directory) and hit tab, and bash automagically completes it for you. For example, if you wanted to run the program *WPrefs* (Window Maker prefrences util), all you have to do is type *WP<tab>* and bash will fill in the rest plus a trailing space.

bash — sRp at 28 January, 01:06:05

Hitting *META-P* in bash will allow you to search through the bash history.

bash — sRp at 27 January, 13:24:42

If you find yourself having to *cd* back and forth between long directory names, bash's pushd is the perfect solution. Start in one of the directories, and the type *pushd directory2* to go to the second directory. Now if you type *dirs* you should see the two directories listed. To switch between these two directories just type *pushd +1*

bash — sRp at 27 January, 13:16:39

While using bash, if you have typed a long command, and then realize you don't want to execute it yet, don't delete it. Simply append a # to the beginning of the line, and then hit enter. Bash will not execute the command, but will store it in history so later you can go back, remove the # from the front, and execute it.

bash — sRp at 27 January, 13:10:05

In the bash shell, *CTRL-U* will delete everything to the left of the cursor.

bash sRp at 27 January, 13:08:22
CTRL-T in bash will transpose two characters; great for typos.

bash sRp at 21 January, 05:39:18
Hitting *CTRL-W* in bash will delete the word just before your cursor. *CTRL-Y* will yank back in the last deleted word (or words if they were delete consecutively). If you deleted words after you deleted what you wanted to yank back in, and already pressed *CTRL-Y*, you can use *ALT-Y* to look through those words.

bash Mike Lowrie at 19 January, 07:17:36
Here's another way to change into long directory names in bash. For example, the directory, *samba-2.0.0beat2*. You can put in *cd samb** and it will change to the directory that matches the wildcard.

bash Sid Boyce at 19 January, 05:00:05
In the bash shell, you can utilize shortcuts. If your last command started with an *l* was *less xxx*, then *!l* will re-execute it. However, if you had been using *lpr* and *ln* as well, and you wanted to run *less* again, then *!le* would execute it.

bash sRp at 18 January, 21:25:17
In bash, hitting ALT-b will move you back a word, and hitting ALT-f will move you forward a word.

bash sRp at 18 January, 21:25:10
Typeing CTRL-l at a bash prompt, will clear the screen, and put the current line at the top of the screen.

bash schvin at 17 January, 12:46:44
Turning on the scrolllock in a console will pause or suspend the current command in progress in bash, such as *ls*, *du* or *mpg123*.

bash mulo at 30 September, 21:43:22
To lowercase files in current $PWD #!/bin/sh for x in * do newx=`echo $x | tr "[:upper:]" "[:lower:]";` mv "$x" "$newx" echo "$x --> $newx" done

bash Jose at 30 September, 21:43:33
For one fast and effective `clear' use echo e='\ec' It does more that `clear'

bash nexz at 30 September, 21:44:50
Finding out all the commands installed on your box? At the prompt, press tab twice and it will ask you if you want to see all the commands. Say y and it will show you all the commands that you installed on your box including shell syntax. Very easy to find out and to familiar yourself with the commands you don't know (btw, this only searches according to path variable set in bash login files). But be careful if you are the root; try --help or man page first before blindly type into it. If all the commands listed are in single column and you can't see the top, edit .bash_profile or .bashrc to include this alias: alias ls="ls -C". Then you should be able to see all. One other alternative might be to increase the buffer for the terminal so that it will hold more characters. Hope this helps!

bash Daniel Giribet at 30 September, 21:46:07
Would you like to list only directories (without a long -l listing)?

```
dirs () {
ls -F $1 | grep \/ | sed -e 's/\/$//g'
}
```

Use 'dirs ' on your bash shell and enjoy!

bash sRp at 31 July, 19:23:44
The readline support in the bash shell defaults to *emacs* editing mode. You can easily switch that to *vi* mode by issuing the following command: set -o vi.

bash Antonio at 8 February, 12:42:20
If you use bash, you can search backwards into its history: hit *CTRL-R* and start typing what you want to search (it works exactly as in Emacs). If there are lots of similar lines in your history, repetedly typing *CTRL-R* will browse through them

bash irfan ahmed at 23 December, 19:36:34
bash allows you to move between the current directory and the previous directory using the hyphen after the cd command. Say you were in /home/john/pies/american. You give the command ***cd /home/jack/steak/grilled*** Now you could back to the ../../american directory using ***cd -***

bash hictio at 18 January, 02:30:17
you can clear the screen when you logout, in bash, by adding this to the *~/.bash_logout* file:

```
setterm -clear
```

if you don´t have a *.bash_logout* file, just make one.

bash johnnycal at 18 January, 02:31:14
I use ***cd bla; ls -l bla*** so much I made a function for it ***see***

```
function see () { cd $1; ls . ; }
```

bash Nate Fox at 27 December, 04:32:07
In bash, if you add this:

```
complete -d cd
```

Into your ***~/.bash_profile*** or ***/etc/profile*** file, then when you cd, it will only search for directories. So if you have a file called "jiggy" and a directory called "joogy" and those are the only things in the directory, and you type ***cd*** and press tab, it will just go into "joogy".

bash sRp at 5 September, 17:02:58
Under *bash* or *zsh*, if you would like to edit a previous command in a text editor instead of on the command line, use the *fc* command.

bash frodo at 10 April, 04:15:04
Aliasing ***dir*** to list just directories can be useful. To do so, do the following:

```
alias dir='ls -l | grep ^d'
```

grep in this case searches for a ***d*** in the first column of each line.

bash HellHound at 14 January, 04:31:20
Another search-in-bash thingy: *CTRL+R*, this is more "realtime"--when you enter a char/string, it gives you a found match directly.

bash Joerg Tretter at
If you want to switch off the "beep" during command line-completion you should add an entry either in your *~/.inputrc* or system wide in your */etc/inputrc*:

```
for visual signal : set bell-style visible
for absolutely no signal: set bell-style none
```

bash Jason P. Stanford at 20 May, 05:21:59
This is a variation for the "colorful directory listing" hint users, that works "better" under bash. Put the following in *$HOME/.bashrc* or *$HOME/.bash_profile*:

```
function v () { ls -l --color=auto $*; }
function d () { ls --color=auto $*; }
```

HINT: Think of 'v' as "verbose" and 'd' as "directory". And they're much quicker to type (only a single char), so this should satisfy most unix junkies.

Recommended Links

Google matched content

Softpanorama Recommended

Top articles

[Jul 07, 2020] **The Missing Readline Primer by Ian Miell Published on Jul 07, 2020 | zwischenzugs.com**

[Jul 05, 2020] **Learn Bash the Hard Way by Ian Miell [Leanpub PDF-iPad-Kindle] Published on Jul 05, 2020 | leanpub.com**

[Jul 04, 2020] **Eleven bash Tips You Might Want to Know by Ian Miell Published on Jul 04, 2020 | zwischenzugs.com**

[Jul 02, 2020] **7 Bash history shortcuts you will actually use by Ian Miell Published on Oct 02, 2019 | opensource.com**

[Aug 14, 2019] **linux - How to get PID of background process - Stack Overflow Published on Aug 14, 2019 | stackoverflow.com**

[Jan 26, 2019] **Ten Things I Wish I'd Known About about bash Published on Jan 06, 2018 | zwischenzugs.com**

Sites

- Blog - Putorius A lot of interesting information and tips about bash and shell programming. Some tips are of very high quality.

- Tips of the Trade
- Bash tips and tricks " Richard's linux, web design and e-learning collection
- My Favorite bash Tips and Tricks
- 10 shortcuts to master bash - Program - Linux - Builder AU
- bash tips
- 10 Essential UNIX-Linux Command Cheat Sheets TECH SOURCE FROM BOHOL
- (1) What are some useful Bash tricks - Quora
- basic ~.bashrc ~.bash_profile tips thread (Page 2) - Community Contributions - Arch Linux Forums

Good advices and "tutorials" for .profile you can find here:

- Speed Up Your Terminal Workflow with Command Aliases and .profile - Tuts+ Computer Skills Tutorial
- Improving My Workflow
- Hacking your terminal
- My Mac OSX Bash Profile

- You should really give sites like commandlinefu.com a check sometime since that's really the best way to browse this kind of information.

  erwanjegouzo/dotfiles

Just in general go to github.com and check out the bash files, there are some really good ones.

Etc

Society

Groupthink : Two Party System as Polyarchy : Corruption of Regulators : Bureaucracies : Understanding Micromanagers and Control Freaks : Toxic Managers : Harvard Mafia : Diplomatic Communication : Surviving a Bad Performance Review : Insufficient Retirement Funds as Immanent Problem of Neoliberal Regime : PseudoScience : Who Rules America : Neoliberalism : The Iron Law of Oligarchy : Libertarian Philosophy

Quotes

War and Peace : Skeptical Finance : John Kenneth Galbraith :Talleyrand : Oscar Wilde : Otto Von Bismarck : Keynes : George Carlin : Skeptics : Propaganda : SE quotes : Language Design and Programming Quotes : Random IT-related quotes : Somerset Maugham : Marcus Aurelius : Kurt Vonnegut : Eric Hoffer : Winston Churchill : Napoleon Bonaparte : Ambrose Bierce : Bernard Shaw : Mark Twain Quotes

Bulletin:

Vol 25, No.12 (December, 2013) Rational Fools vs. Efficient Crooks The efficient markets hypothesis : Political Skeptic Bulletin, 2013 : Unemployment Bulletin, 2010 : Vol 23, No.10 (October, 2011) An observation about corporate security departments : Slightly Skeptical Euromaydan Chronicles, June 2014 : Greenspan legacy bulletin, 2008 : Vol 25, No.10 (October, 2013) Cryptolocker Trojan (Win32/Crilock.A) : Vol 25, No.08 (August, 2013) Cloud providers as intelligence collection hubs : Financial Humor Bulletin, 2010 : Inequality Bulletin, 2009 : Financial Humor Bulletin, 2008 : Copyleft Problems Bulletin, 2004 : Financial Humor Bulletin, 2011 : Energy Bulletin, 2010 : Malware Protection Bulletin, 2010 : Vol 26, No.1 (January, 2013) Object-Oriented Cult : Political Skeptic Bulletin, 2011 : Vol 23, No.11 (November, 2011) Softpanorama classification of sysadmin horror stories : Vol 25, No.05 (May, 2013) Corporate bullshit as a communication method : Vol 25, No.06 (June, 2013) A Note on the Relationship of Brooks Law and Conway Law

History:

Fifty glorious years (1950-2000): the triumph of the US computer engineering : Donald Knuth : TAoCP and its Influence of Computer Science : Richard Stallman : Linus Torvalds : Larry Wall : John K. Ousterhout : CTSS : Multix OS Unix History : Unix shell history : VI editor : History of pipes concept : Solaris : MS DOS : Programming Languages History : PL/1 : Simula 67 : C : History of GCC development : Scripting Languages : Perl history : OS History : Mail : DNS : SSH : CPU Instruction Sets : SPARC systems 1987-2006 : Norton Commander : Norton Utilities : Norton Ghost : Frontpage history : Malware Defense History : GNU Screen : OSS early history

Classic books:

The Peter Principle : Parkinson Law : 1984 : The Mythical Man-Month : How to Solve It by George Polya : The Art of Computer Programming : The Elements of Programming Style : The Unix Hater's Handbook : The Jargon file : The True Believer : Programming Pearls : The Good Soldier Svejk : The Power Elite

Most popular humor pages:

Manifest of the Softpanorama IT Slacker Society : Ten Commandments of the IT Slackers Society : Computer Humor Collection : BSD Logo Story : The Cuckoo's Egg : IT Slang : C++ Humor : ARE YOU A BBS ADDICT? : The Perl Purity Test : Object oriented programmers of all nations : Financial Humor : Financial Humor Bulletin, 2008 : Financial Humor Bulletin, 2010 : The Most Comprehensive Collection of Editor-related Humor : Programming Language

Humor : Goldman Sachs related humor : Greenspan humor : C Humor : Scripting Humor : Real Programmers Humor : Web Humor : GPL-related Humor : OFM Humor : Politically Incorrect Humor : IDS Humor : "Linux Sucks" Humor : Russian Musical Humor : Best Russian Programmer Humor : Microsoft plans to buy Catholic Church : Richard Stallman Related Humor : Admin Humor : Perl-related Humor : Linus Torvalds Related humor : PseudoScience Related Humor : Networking Humor : Shell Humor : Financial Humor Bulletin, 2011 : Financial Humor Bulletin, 2012 : Financial Humor Bulletin, 2013 : Java Humor : Software Engineering Humor : Sun Solaris Related Humor : Education Humor : IBM Humor : Assembler-related Humor : VIM Humor : Computer Viruses Humor : Bright tomorrow is rescheduled to a day after tomorrow : Classic Computer Humor

The Last but not Least *Technology is dominated by two types of people: those who understand what they do not manage and those who manage what they do not understand ~Archibald Putt. Ph.D*

Copyright © 1996-2021 by Softpanorama Society. www.softpanorama.org was initially created as a service to the (now defunct) UN Sustainable Development Networking Programme (SDNP) without any remuneration. This document is an industrial compilation designed and **created exclusively for educational use** and is distributed under the Softpanorama Content License. Original materials copyright belong to respective owners. ***Quotes are made for educational purposes only in compliance with the fair use doctrine.***

FAIR USE NOTICE This site contains copyrighted material the use of which has not always been specifically authorized by the copyright owner. We are making such material available to advance understanding of computer science, IT technology, economic, scientific, and social issues. We believe this constitutes a 'fair use' of any such copyrighted material as provided by section 107 of the US Copyright Law according to which such material can be distributed without profit exclusively for research and educational purposes.

This is a Spartan WHYFF (We Help You For Free) site written by people for whom English is not a native language. Grammar and spelling errors should be expected. **The site contain some broken links as it develops like a living tree...**

Donate

You can use PayPal to to buy a cup of coffee for authors of this site

Disclaimer:

The statements, views and opinions presented on this web page are those of the author (or referenced source) and are not endorsed by, nor do they necessarily reflect, the opinions of the Softpanorama society. We do not warrant the correctness of the information provided or its fitness for any purpose. The site uses AdSense so you need to be aware of Google privacy policy. You you do not want to be tracked by Google please disable Javascript for this site. ***This site is perfectly usable without Javascript.***

Last modified: June 08, 2021