Core

UNIX

Quick Reference to BASH

Table of Contents

1. Commands

1.1. Data Commands

1.1.1. cat

Concatenate files and print on standard output.
Simply displays the contents of a file or files as one continuous stream.

1.1.2. cmp

Compares two files on a, byte by byte basis. Returns the first position (byte number and line) where the files differ. If there are no differences then cmp returns nothing. (See also: diff)

1.1.3. cut

Cuts a portion from each line of a file.
Can cut either, a range of characters or specific fields, using a delimiter.

```
cut -c2 filename                    gets the second character of each line
cut -c-5 filename                   gets the first five characters
cut -c5- filename                   gets the fifth character to the end of the line
cut -c3-7 filename                  gets the third through seventh char
cut -c3,7 filename                  gets the third and seventh char
cut -d ":" -f2 accounts     using ":" as a delimiter, retrieve the second field.
cut -d ":" -f2,4 accounts   using ":" as a delimiter, retrieves the second and fourth
field.
cut -d ":" -f2-4 accounts   using ":" as a delimiter, retrieves the second through
fourth fields.
```

1.1.4. diff

Compares two files on a line by line basis. Shows how the two files differ. That is, shows what changes would have to be made to change file1 to be identical to file2. (Also see cmp)

1.1.5. echo

```
echo hello              writes hello followed by a newline character to the screen.
echo -n hello                   omits the newline character
echo -e "tab \t This lets you use escape characters \n\n"
```

1.1.6. grep

Searches through a file for strings or regular expressions. See Regular Expressions.

```
grep the filename           Retrieves lines that contain "the"
```

```
grep -v the filename        inverse: returns lines that do not contain "the"
grep -i the filename        case insensitive
grep -w the filename        match whole word
grep -c the filename        count the number of lines containing "the"
```

1.1.7. head

```
head -5 myFile              retrieves the first 5 lines
head -c100 myFile                retrieves the first 100 characters
```

1.1.8. less

Less lets you scroll forward and backwards through a file, using the up and down arrow keys.

1.1.9. more

More lets you scroll forwards and backwards through a file, using the [Enter] key.

1.1.10. sort

```
sort file1
sort -r file1               sort in reverse order

sort -t":" -k4 accounts     Using ":" as the delimiter, sort the lines by each key
                            starting with the fourth key to the end of the line.  Note
                            similarities and differences from cut.

sort -k2 accounts           Using any space (or spaces) as the delimiter, sort by the
                            lines by each key starting with the second key to the end
                            of the line.

sort -t":" -k3,5 accounts   Using ":" as the delimiter, sort the lines by the third, fourth
                            and the fifth key.  If there are more keys after the fifth,
                            don't bother sorting by those keys.

sort -n -t":" -k2 accounts  using ":" as the delimiter, sort the lines by the second key
                            and sort in numeric order rather than alphabetic.

sort -u accounts            “Unique” - After sorting, display any line that is repeated
only once.
```

1.1.11. tail

```
tail -5 myFile.txt          returns the last 5 lines from myFile.txt
tail -c20 myFile.txt        returns the last 20 characters from myFile.txt
```

1.1.12. tr

```
echo James:Taylor | tr ":" " "          replaces ":" with a space character
echo chris | tr crh Evl                 turns chris into Elvis
echo -n abcde123 | tr -c aeiou -        outputs a---e---
                                        -c means compliment set.  This example
                                        turns any character not in aeiou to
                                        hyphens.

cat myFile | tr [:lower:] [:upper:]     turn lowercase characters into uppercase
echo hello | tr -d h                    deletes all the 'h' character (ello)
echo hello | tr -d ho                   deletes all instances of the 'h' and 'o' (ell)
cat myFile | tr -d a-h                  deletes all lower case chars between a
                                        and h (inclusive)
cat myFile | tr -d [:upper:]            deletes all capital letters from the output
echo NA1234567D | tr -d [:alpha:]       deletes all letters (1234567)
echo NA1234567D | tr -d [:digit:]       deletes all numbers
echo ABBBBBBD | tr -s "B"               Squeeze Bs down to one B.
```

1.1.13. uniq

```
uniq myFile             displays lines from the file but eliminates repeats if that repeat
occurs on the very next line.  To eliminate all duplicates the input file must be sorted.
uniq -c myFile          displays the number of times each line is repeated as well as the
line from the file.
uniq -d myFile          displays only lines that are repeated.
```

1.1.14. wc

wc is word count. When using wc with a filename, it displays the name of the file.

```
wc myFile               displays the number of lines, words, bytes and the name of the
file
wc -l myFile            number of lines
wc -w myFile            number of words (delimited by white space)
wc -c myFile            number of bytes
wc -m myFile            number of characters
```

When using wc with an input stream, (for example `wc -c < file`) it does not display the name of the file.

```
cat myFile | wc -l      display the number of lines in the input stream.
```

Also when used to count the number of characters,within a pipe it also counts the carriage return,
giving the wrong answer

```
echo name | wc -c
```

Use echo –n to ignore the carriage return character

```
echo -n name | wc -c
```

1.2. File Commands

1.2.1. basename

Removes the path and returns the base name for a file. (It simply manipulates the string: it does not confirm the actual existence of the directories nor the file.)

`basename /abc/def/ghi` returns ghi

1.2.2. cd - Change Directory.

`cd ..` to go to parent directory.

1.2.3. chmod

chmod is used to change the permissions to a file.
Both files and directories have three sets of permissions: the first set is the permissions for the owner, the second set is the permissions for others in the same group as the owner, and the third set is the permissions for everyone else (others). Within each set, there are separate permissions for read, write and execute. For example, if you, the owner, have permission to read, write and execute then the code will be: rwx------
For directories, the r is the permission to read, w is the permission to write, and x is the permission to enter the directory or use this directory name in a command. In order to write to a directory, you must have the permission to enter the directory as well as write to it.

1.2.3.1. Octal Method

The chmod octal method turns permissions on and off using octal numbers (0 through 7). The bits in the numbers 0 to 7 are used to turn the rwx bits on or off.

1.2.3.2. Symbolic Method (= + -)

`u` – means "user/owner"
`o` – means “other”
`g` – means “group”
`a` – means “all”

`+r` – add permission to read
`-r` – revoke permission to read
`+w` – add permission to write
`-w` – revoke permission to write
`+x` – add permission to execute
`-x` – revoke permission to execute
examples:

`chmod go-r file` remove read permission for group and others.
`chmod u=rwx,g+x,o-r file` Set permission for user (owner) to rwx, add execute for group and remove read permission for group and others.

1.2.4. chown

chown changes the owner of the file.

1.2.5. dirname

Removes the file base name, and returns the path. (It simply manipulates the string: it does not confirm the actual existence of the directories nor the file.)

`dirname /abc/def/ghi` returns /abc/def

1.2.6. du

Displays a list of directories and how much disk space, in blocks (512 characters) they occupy.

1.2.7. cp - Copy files and directories

```
cp sourcefile targetfile
```

1.2.8. file - Show the type of files

`file *` Lists the files and file types.

1.2.9. find – Find Files or Directories

`find myFile` finds myFile in the current directory

`find . -name myFile` finds myFile in current directory and its sub-folders
. indicates the current directory, which is the default

`find . -size -10k` finds any file less than 10k in size

`find . -size +10k` finds any file over 10k in size

`find . -type f -size +10k` finds any file over 10k. Type uses f for file, d for directory, l for link

`find /usr /bin -name myFile` finds an item in specific directories (item – file/ directory)

`find . -mtime -2` finds items modified during the past 2 days

`find . -mmin -60` finds items modified in the past 60 minutes

`find . -mmin +60` finds items modified more than 60 minutes ago

`find . -name myFile -print` -print is the default action and not always required

`find . -name "The*" -exec rm {} \;` execute rm on all files starting with “The” in the current directory and every sub-directory.

`find . -name "The*" -exec rm -i {} \;` an interactive version

1.2.10. ln – link files or directories

1.2.10.1. Hard Link

Syntax:

```
ln targetfile linkname
```

When hard linked, the file names actually refer to the same physical data location.
Cannot cross filesystems
Can link files but not directories
Hard links are always linked even when files are moved.

1.2.10.2. Soft Link

Syntax:

```
ln -s targetfile linkname
```

A soft link is a separate file that acts as a “reference pointer“ to another file or directory.
Can cross filesystems
Can link files or directories
If files are moved, soft links are not updated.

1.2.11. ls – List file

`ls` lists files
`ls -l` long listing, providing more information including permissions
`ls -i` inode number is included
`ls -a` lists all files, including hidden files.
`ls -r` reverse. Sorts listing in reverse order.
`ls -R` recursive, including subdirectories

1.2.12. mkdir

Create directory

1.2.13. mv

Move or rename

1.2.14. pwd – print working directory

1.2.15. rm - remove or delete

`rm -r` will recursively delete all files and directories.
`rm -i` interactive delete

1.2.16. readlink

`readlink -m $1` will return the absolute path and filename for a file.

1.2.17. rmdir

Remove directory

1.2.18. stat

stat –c%i filename - retrieves the inode number for the file

1.2.19. touch

Create a new file

1.2.20. tree

Display the directories and files in a tree structure.

1.3. Math and Logic Commands

1.3.1. bc

Performs floating point math. See Performing Calculations.

1.3.2. let

Performs integer math. See Performing Calculations

1.3.3. test

Performs a logical test - Single Square Brackets (or test)

1.4. Process Commands

If a process is running in the foreground then it is the current job. When a process is running in the foreground, the system expects any user input to be handled by the current job.

1.4.1. &

You can also start command running in the background using `&`.

1.4.2. bg

To resume a stopped job so it runs in the background, use `bg`.
background – You can change stopped jobs to running in the background using bg

1.4.3. CTRL-c

CTRL-c will terminate the current job. After CTRL-c, the job will not appear in the jobs list and not appear in ps.

1.4.4. CTRL-z

To suspend the current job, use `CTRL-z`. After CTRL-z, the job will appear as "stopped" in the jobs list, and the process will appear in ps.

1.4.5. fg

`fg` will resume a stopped job so it runs in the foreground (as the current job).
`fg` can also bring a job that is running in the background to the foreground.

1.4.6. jobs

To list all jobs, use the `jobs` command. It will list stopped jobs and jobs running in the background.

1.4.7. kill

To kill jobs, use: `kill %jobnumber`
To kill a process use: `kill pid`

1.4.8. ps

List processes.

1.4.9. sleep

Causes the script to pause for a specified number of seconds.

1.5. System Commands

1.5.1. clear

Clears the screen and places the cursor at the top of the screen.

1.5.2. date

Retrieves the system date and time

1.5.3. env

Lists the environment variables.
See Environment Variables

1.5.4. eval

Using eval forces an extra pass of variable substitution.

An assignment statement will perform variable substitution.

```
myshell=$SHELL
echo $myshell                    shows "/bin/bash"
```

But when reading from a file, if the file contains "$SHELL", the variable would be set to "$SHELL".

```
myshell=$(cat file)
echo $myshell                    shows "$SHELL"
```

Using eval will force another pass of variable substitution. Therefore the "$SHELL" will be changed to "/bin/bash".

```
eval myshell=$(cat file)
echo $myshell                    shows "/bin/bash"
```

Example2: The following will dynamically create variables: name1, name2 ... nameN

```
for ((i=1;i<=$N;i++)); do
   read -p "Please enter name number $i: " name$i
done
```

However, to dynamically get the values from $name1, $name2 we need the following:

```
for ((i=1;i<=$N;i++)); do
   eval echo "Name $i is \$name$i"
done
```

1.5.5. export

See Environment Variables

1.5.6. finger user

Shows information on the user.

1.5.7. lsof

Lists processes and the files the processes have open.
Example:
`vi myfile` (starts vi which opens a file called myfile)
`CTRL-z` (stops but vi continues to hold the myfile open)
`lsof | grep myfile` (finds the process which has myfile open)

1.5.8. printenv

Lists the environment variables (Same as env)

1.5.9. set

`set`	Lists all the shell and environment variables and functions
`set andy bruce charlie`	Sets $1, $2, and $3 to andy, bruce, and charlie, respectively
`set -x`	Turns on trace mode

1.5.10. sh

Runs a script in the default shell.
`sh myscript`
`sh -x myscript` Runs the script with xtrace (tracing statements).

1.5.11. Uname

Displays the name of the server
`uname -a` Show all information

1.5.12. users

Shows who is logged on.

1.5.13. w

Shows who is logged on and what they are doing.

1.5.14. which cmd

Shows the full path of the command, cmd.

1.5.15. who

Shows who is logged on.

1.5.16. whoami

Shows the userid that is currently logged on to the session.

1.5.17. xargs

xargs takes the standard input from a pipe and uses that input as the command line argument(s) for another command.

```
echo /home | ls
```

Now try the following:

```
$ echo "/home" | xargs ls
```

xargs will take the output of 'echo "/home"' (which of course is just "/home") and pass it as an argument to the ls command.

This time, you'll see that everything returned by 'ls "/home"' has been passed as an argument to echo, resulting in a long, messy string of text appearing on screen. We can improve on this by invoking xargs' "-n" option which limits the number of arguments passed to echo at a time. Try the following:

```
$ ls "/home" | xargs -n2 echo
$ ls "/home" | xargs -n1 echo "Filename ==>"
$ ls "/home" | xargs -n1 echo "Filename ==>" | more
```

2. Environment Variables

`$HOME` - Stores the path to the user's home directory
`~` - Also stores the path to the user's home directory.
`$USER` - Stores the userid that is currently logged on.
`$PATH` - Stores a list of directories the command line interpreter will search when you enter a command.
`$SHELL` - Stores the name of the shell you are using.
`export` - Export command lets you create your own environment variables. Example:

```
MYENV_VAR=abc
export MYENV_VAR
```

3. Performing Calculations

3.1. Integer Math

3.1.1. Math within Text

Math within text requires either, a dollar sign and double parentheses , $(( )), or requires a dollar sign and single square brackets, $[ ].

```
echo This is math $((7-2)) within text.
echo $[7-2]
echo $[3+4*5]
echo $[(3+4)*5]
var1=5
echo division $[23/$var1]  modulo $[12%$var1]
```

3.1.2. Math with let command

The `let` command allows mathematical operations

Example

```
let x=5*3
let x++
echo $x
```

3.2. Floating Point

You can perform floating point math with the “bench calculator” bc.

```
echo "scale=3; 5/2" | bc
echo "scale=3; (5/2)^3" | bc
```

or

```
bc <<< "scale=3; 5/2"
```

4. Glob Matching (Simple Pattern Matching)

*	matches zero or more of any character
?	matches any single character and there must be a character
[…]	matches one character from the list of characters. The list can contain a range of characters such as [a-zA-Z] and can be negated/complemented by using ^ or ! at the start, e.g. [^a-zA-Z] = not a letter. [!0-9] = not a digit

5. Brace Expansion

{ , }	matches a list of options. Each option is separated by commas.
..	.. denotes a range.

6. Regular Expressions (Regex)

Regular Expressions provide a way to perform advanced pattern matching.

Regular Expressions can be used in conditional statements (such as if, while, and until) or using the grep command.

6.1. Operator Description

`.` Match one character
`^` Matches the start of line. Used to anchor the match.
`$` Matches the end of the line. Used to anchor the match.
`[...]` Match any character enclosed within the brackets
`[^...]` Match a character that is not within the brackets.
`[a-z]` Matches any lower case letter
`[A-Z]` Matches any upper case letter
`[a-zA-Z]` Matches any letter
`[0-9]` Matches any digit

For each of the following, X is a literal or a regular expression
`X*` Match zero or more occurrences of X
`X+` Match one or more occurrences of X
`X?` Match one or zero occurrences of X. That is, X is optional.
`X{n}` Match exactly n occurrences of X
`X{n,}` Match n or more occurrences of X
`X{n,m}` Match between n and m (inclusive) occurrences of X
`X | Y` Matches X or Y
`\` Escape a special character
`(XY)` Create a group.

6.2. Regular Expressions with Grep

Grep uses two versions of regular expressions: "basic" and "extended".

6.2.1. Basic - grep

Basic regular expressions use only the **^ $. * [ ]**.
The extended characters are **+? | { } ( )** These characters are treated as literals when using basic regular expressions.
`grep "^a+$" filename` will find `literally a+`
The **+ | { } ()** characters can be used for their extended meaning but they must be prefixed with a backslash \ and the expression must be surrounded by double quotes.
`grep "^a\+$" filename` will find `a, aa, aaa etc`

6.2.2. Extended - grep –E or egrep

The -E flag will cause grep to use extended regular expressions. egrep also uses extended regular expressions. The characters +? | { } () are used for their extended meaning without the need for the \.

`grep -E "^a+$" filename`	will find a, aa, aaa etc.
`egrep "^a+$" filename`	will also find a, aa, aaa etc.
`grep <pattern> *`	matches all files containing the specified pattern

6.3. Regular Expressions in Logic Statements

When using regular expressions in conditional logic statements (such as if, while or until), you must use double square brackets, "[[ ]]" and "=~" (equals tilde).
When using regular expressions in conditional logic statements, you do not prefix the **+ | { } ( )** characters with backslash.

6.4. Characters Needing an Escape

The following special characters need to be escaped using the backslash '\' if you want to use them as literals in regular expressions:

Char	Description
'	single quotation mark
!	exclamation mark
$	dollar sign
;	semicolon
*	asterisk
&	ampersand

7. How BASH Processes Commands

7.1. Single or Double Quotes?

Strings surrounded by single quotation marks will always be treated as literal. For example, variables within the string will not be expanded.

```
var=Hello
echo 'The value of $var is: '$var
```
returns - The value of $var is: Hello

Strings surrounded by double quotes will allow expansion of variables.

```
var=Hello
echo "The value is $var"
```
returns - The value is Hello

7.2. Processing Steps

7.2.1. Parsing

Parsing is the process of analyzing a text, made of a sequence (sentence) of tokens (words), to determine its grammatical structure with respect to a given set of rules.

For example, the command 'echo Hello Alan' would be split into the tokens 'echo', 'Hello' and 'Alan', while the command 'echo "Hello Alan" ' would be split into just two tokens: 'echo' and "Hello Alan".

7.2.2. History expansion and alias substitution

7.2.2.1. $!!

Once a command has been parsed, bash scans it for history commands and aliases. Note that it only does this at the command line – not when executing scripts.

Examples:

```
[alan.mann@unix ~]$ echo hello
hello
[alan.mann@unix ~]$ !!
echo hello
hello
[alan.mann@unix ~]$ echo !!
echo echo hello
echo hello
[alan.mann@unix ~]$
```
As you can see "!!" is simply replaced with the last command run.

7.2.2.2. Alias

An alias is an alternative name given to a particular command. For example, you might want to create an alias for a command that outputs a motivational message on-screen each time you
enter it:

```
alias dir='ls –l'
```
Having run the above command, issuing dir from the command line will execute an ls -l

7.2.3. Brace expansion

The following example illustrates how brace expansion works:

```
$ touch file{1,2,3}
```
This will expand to:

```
$ touch file1 file2 file3
```

7.2.4. Tilde expansion

The next scan will look through tokens for **tildes** (i.e. the '**~**' symbol) to expand. As you'll recall the tilde expands to the address of your home directory. When used in conjunction with other characters, it can also expand to a few other things:

`~user` : the home directory of user
`~+` : the current directory
`~-` : `the` last directory you were in before this one
This last one is especially useful when navigating so take note!

7.2.5. Parameter and variable expansion

On this scan, tokens preceded by '$' are replaced with relevant values. For example '$1' is replaced by the value contained in parameter 1, and $something is replaced by the value contained in the variable 'something'.

Note that variable names can also be contained in curly braces; i.e. '${3}' is equivalent to '$3' and '${something}' equivalent to '$something'. This is useful if you are embedding a variable name with a string.

```
cp $file ${file}.bak
```

7.2.6. Command substitution

Next comes, **command substitution**, when a command is enclosed a inside parentheses preceded by a dollar sign $(command). We refer to the value sent to the standard output by that command. To perform a command substitution, the command within the substitution must itself be parsed, scanned and executed successfully before the value of its **STDOUT** is substituted.

7.2.7. Arithmetic expansion

At this stage, arithmetic expressions enclosed in $(()) are expanded. As with command substitution, the contents of the expression is parsed and scanned before the value of the expression is calculated and substituted. For example, in:

```
$ echo $(($(echo horse | wc -c)-1))
```
the command substitution needs to be evaluated before the result of the entire arithmetic expression can be calculated and substituted.

7.2.8. Word splitting

This is the next stage in the process. On this scan of the command line, all of the results of previous expansions are split into separate words based on tab, space and newline delimiters, unless they are contained in quotation marks.

7.2.9. Globbing

The next scan looks for patterns in filenames and expands them accordingly into an alphabetical list ("globbing" just means "wild card matching"). For example, if there are three files called fileA, fileB and fileC and we run the following command:

```
rm file*                this command will be expanded to:
rm fileA fileB fileC
```

7.2.10. Quote Removal

All non-escaped quotation marks are removed.

7.2.11. Execution

Once all expansions have been performed, the command can be run. Recall that if command substitution occurs, the command within $() will be executed first. Also, if pipes are used then the command to the left of the pipe will be executed before the command to the right is expanded.

The shell will assume that the leftmost word on the command line (or within a command substitution or immediately to the right of a pipe) is a command.

The first thing it does is check if the command is a user-defined function – if so, it executes the corresponding code. If it isn't a function, it checks if it is a built-in command.

If the command is neither of these the shell searches for the executable program that corresponds to it; e.g. if you enter 'ls' at the command line, bash will look for the corresponding 'ls' file. It searches through the directories listed in a special variable called PATH – try echoing $PATH now to see what these are. If it finds the corresponding file, it executes it – if not, it throws a 'command not found' error.

8. Scripts

8.1. Command Line Arguments & Manipulation

`$1, $2, ...`	references the command line arguments
`$0`	name of the script that is running
`$*`	all arguments
`$@`	all arguments
`$#`	number of argument
`shift`	shift arguments so that $1 holds what $2 used to hold etc.
`set FDM UNIX`	sets $1 to FDM and $2 to UNIX
`set -x`	turns on the xtrace from within a script

8.2. Conditional Statements

8.2.1. IF

8.2.1.1. IF Syntax

```
if <logic>
then
   <list of statements>
elif <logic>
then
   <list of statements>
else
   <list of statements>
fi
```

8.2.2. CASE

```
case $word in
   a)       echo "Exact match on the letter a"
            echo "Another statement for this case";;
   [0-9])   echo "Any single digit" ;;
   [0-9][0-9])   echo "Any two digits" ;;
   *[0-9])       echo "something ending with a digit" ;;
   [0-9]*)       echo "something starting with a digit" ;;
   [yY]|YES|yes)      echo "y or Y or YES or yes";;
   *)       echo The word is something else;;
esac
```

8.3. Functions

Bash scripts do not distinguish between functions and procedures. All subprograms are called functions whether they return a value or not.

8.3.1. Parameters

There are only input parameters.
Parameters are accessed within the function as $1 $2 etc. and $*, $#.

8.3.2. Return Codes

$? – variable that holds the exit or return status of the previous command executed.

8.3.3. Call Stack

Every call to a function adds the function to the call stack. The entry on the call stack includes the information on where to return when the function ends.
Every time a function ends, the entry is removed from the call stack.
Beware of recursion. If your functions never end (because they just call other functions) then the call stack will just build up and up.
Recursion is a very powerful tool to use but only use recursion when you need it, which means avoid writing functions that call themselves.

8.4. getopts

To get your program to recognize options, you can use a command called getopts which searches the command line for arguments beginning with a '-'. A typical getopts statement uses a CASE statement embedded in a WHILE loop:

```
while getopts ab OPTION
do
  case $OPTION in
    a) echo "option a selected";;
    b) echo "option b selected";;
    *) echo "invalid option";;
  esac
done
```

Note the syntax: 'getopts', followed by a string containing possible options ('ab' in this case) followed by a variable name in which to store a or b if either is found on the command line. $OPTION is then processed by the CASE statement.

8.4.1. Using options together with arguments

The $OPTIND variable can be used together with the shift command to shift past the options flags.
It stores an index number indicating the position of the next option/argument to be processed.

```
#!/bin/bash
upper=0
while getopts u OPTION; do
  case $OPTION in
    u) upper=1;;
    *) echo Invalid Option
```

```
        exit;;
  esac
done
shift $(($OPTIND-1))
while [ $# -gt 0 ]
do
if [ $upper -eq 1 ]
   then
      echo $1 | tr [:lower:] [:upper:]
   else
      echo $1
   fi
   shift
done
```

Try the preceding script with and without the –u flag and with several command line arguments.

8.5. Looping Statements

8.5.1. For

8.5.1.1. Looping Through List

```
for i in 1 a 4 mike 7 10
do
   echo $i
done
```

8.5.1.2. C Style FOR

```
for ((i=1;i<10;i++)); do
   echo The number is $i
done
```

8.5.2. While

A while statement uses the same logic as if statements.

```
while <logic>
do
   <list of statements>
done
```

8.5.3. Until

An until statement uses, the same logic as an if statement.

```
until <logic>
do
   <list of statements>
done
```

8.5.4. Continue

Continue will cause the code to immediately jump to the top of a loop without executing the commands after the, continue.

8.5.5. Break

Break will exit a loop.

8.6. <Logic>

These logic statements can be used with if, while or until. You can also test them by running them on the command line. To see the result use `echo $?`

8.6.1. No Brackets or No Parentheses

When your logic statement contains no brackets (USA: parentheses) BASH checks the status or return code.

```
#!/bin/bash

if ls
then
  echo ls was successful (return code = 0)
else
  echo ls failed (return code <> 0)
fi

function f() {
   return 0
}

if f
then
   echo f was successful (return code = 0)
fi
```

BASH understands the strings “true” and “false” as Boolean “true” or false, respectively. Also, a variable can be set to “true” or “false”.

```
#!/bin/bash

var=true
if $var
then
  echo This will run because $var is true
fi

while true
do
  echo endless loop.  Press Ctrl-C.
done
```

8.6.2. Single Square Brackets (or test)

```
test expression
or
[ expression ]
```

8.6.2.1. String Comparison

```
[ $var = abc ]
[ $var = 'abc' ]
[ $var = "abc" ]
[ $var != 'abc' ]
test $var = 'abc'
```

8.6.2.2. Numeric Comparison

```
[ $var -eq 5 ]
```
or
```
test $var -eq 5
```

`-eq` - equal
`-gt` - greater than
`-lt` - less than
`-ne` - not equal
`-ge` - greater than or equal
`-le` - less than or equal

8.6.2.3. Flags

```
[ -e name ]
or
test -e name
```

`-e name` - name exists (as a file or directory)
`-d name` - name is a directory
`-f name` - name is a regular file.
`-h name` - name is a symbolic link.
`-r name` - name exists and is readable.
`-w name` - name exists and is writeable.
`-x name` - name exists and is executable.
`-s name` - name exists and has nonzero size.
`-z string` - string has zero length
`-n string` - string has non-zero length

8.6.3. Double Square Brackets

Double square brackets provide extended functionality beyond single square brackets.

8.6.3.1. Multiple Logic Conditions && ||

`&&` – Logical AND Ex: `[[ 5 -eq 5 && 6 -gt 2 ]]`
`||` – Logical OR Ex: `[[ 5 -eq 4 || 6 -gt 2 ]]`

8.6.3.2. Simple Pattern Match (Glob Match)

```
[[ $var = [abc]* ]]
```

8.6.3.3. Regular Expression Match

```
[[ $var =~ ^[ab]*(cd){2}$ ]]
```

8.6.4. Double Round Brackets (Parentheses)

Double parentheses are used for numeric and bitwise options.

8.6.4.1. Bitwise Logic

& - Bitwise AND

```
if (( $1 & $2 )); then
  echo "Some bits match"
else
  echo "No bits match"
fi
```

| -Bitwise OR

```
if (( $1 | $2 )); then
  echo "Some bits are on"
else
  echo "No bits are on"
fi
```

&& Logical AND

```
if (( $1 && $2 )); then
  echo "$1 is true AND $2 is true"
else
  echo "Either $1 is false OR $2 is false"
fi
```

|| Logical OR

```
if (( $1 || $2 )); then
  echo "$1 is true or $2 is true"
else
  echo "Neither $1 nor $2 are true"
fi
```

8.7. Sh-Bang!

```
#!/bin/bash
```

Every script starts with a sh-bang.
If you run a script using "sh script" then the sh-bang doesn't make a difference.
If you make your script executable and you execute it by typing the name of the script on the command line then the operating system will use the sh-bang to choose which shell to use.
Example shells and sh-bangs:

```
#!/bin/bash
#!/bin/ksh
#!/bin/dash
```

8.8. User Interface Commands

8.8.1. Displaying Text

```
echo This line will be displayed on screen
```

or

```
cat << eof
This is a way to display
multiple lines of text
in a single "Here document"
eof
```

The here document is text between the <<eof and the eof at the end.

8.8.2. Reading Input from Users

```
read var
read -p "prompt: " var
```

The read command with -p will display a prompt on standard error (the screen).

```
read -n1 -p "Press Any Key to Continue "
```

-n1 means to read only one character.

The read command will not return until the user presses enter and uses spaces as a separator.

```
read -p "Tell me three things: " var1 var2 var3
echo $var1 $var2 $var3
```

To read a password and keep it invisible.

```
#!/bin/bash
read -p "Username: " uname

stty -echo
read -p "Password: " passw
```

```
stty echo

echo $uname
```

8.8.3. Controlling the Cursor Position

```
clear
```

The clear command will clear the screen and position the cursor at the top of the screen.

```
tput cup 0 0
```

The tput command moves the cursor to the row and column position specified, use clear first.

8.9. Variables

8.9.1. Declaring Variables

You do not need to declare variables. You can just start using them.

```
var=hello
```

All variables in Bash are text.

8.9.2. Referencing Variables

Variables are prefixed by $ when referencing them.

```
var=hello
echo $var     will produce hello
```

If necessary, a variable can be delineated using { }

```
var=abc
echo ${var}d    will produce abcd
echo $vard      will produce a blank (vard is empty)
```

8.9.3. Scope

By default variables have global scope, but you can declare variables to be local using the keyword: local.

8.9.4. Command substitution

var=$(ls –l)

8.9.5. Length of the contents of a variable:

```
var="Hello"
echo –n $var | wc –c
```

or

```
var="Hello"
echo ${#var}
```

8.9.6. Arrays

Examples

```
array=("Andy" "Bruce")
echo 'First Element = ' ${array[0]}
echo 'Second Element = ' ${array[1]}
echo 'All Elements  = ' ${array[*]}

Arr[0]=Hello
Echo ${Arr[0]}
```

To copy an array to another array

```
c=(${array[*]})
echo 'All of c = ' ${c[*]}
```

8.9.7. Constants

In BASH, a constant is merely a variable that has been declared as read only.
It is fair game to use constants globally.
For sample code see: constant

9. Streams & Redirection

| | sends standard output of one command to the input of another command (pipe)
> | sends standard output (default) to a file (replaces the file)
1> | sends standard output to a file
2> | sends standard error to a file
>> | appends standard output (default) to a file
1>> | appends standard output to a file
2>> | appends standard error to a file
`echo Error 1>&2` redirects stdout to stderr
`echo Error 2>&1` redirects stderr to stdout
`echo unwanted stuff > /dev/null` redirects to /dev/null to get rid of things that you don't want.

9.1. STDIN

STDIN in has the number 0.

```
cat file
cat < file
cat 0< file
```
explicitly sends file to standard input of cat.

9.1.1. Here Strings

```
echo hello | tr "l" "c"
```
not using a here string

```
tr "l" "c" <<< hello
```
using a here string

9.1.2. Here Documents

Example:

```
cat > mydoc << eof
This is a "here document".
The text can continue
until eof appears on a line by itself.
eof
```

9.2. STDOUT

STDOUT is number 1 and is the default.

```
date 1> file      -- same
date > file       -- same
```

9.3. STDERR

STDERR is number 2.

```
ls 2> file
ls 2> /dev/null
```

9.4. tee

Read from standard input and write to both standard output and file(s)

`echo hello | tee f1 f2`	Creates files f1 and f2 (overwrites if they exist) and writes "hello" to f1 and f2 as well as standard output.
`echo hello | tee -a f1`	Appends "hello" to f1 as well as writing "hello" to standard output. (Creates file f1 if it does not exist.)

10. vi Editor

When you enter VI, you start in command mode. Pressing esc will always bring you back to command mode. In command mode you can access these commands (and many more):

`:#` move to line #. Example: 8 will move you to line 8
`a` append (insert after current character)
`A` append at the end of the current line
`D` deletes the text from the cursor to the end of the current line
`dd` delete the current line
`dw` deletes current word
`gg` move to top of file
`4gg` move to line 4
`G` `move to end of file`
`i` enter insert mode and insert before the current character
`I` enter insert mode and insert at the end of the current line
`r` replace one character (and return to command mode)
`R` enter replace mode
`u` undo last change
`U` undo all changes to the current line
`x` deletes the current character
`ZZ` save your work and exit
`:wq` save your work and exit
`:q!` exit vi without saving
`/word` highlight a particular `word` throughout your file
`:nohlsearch` turn off highlighted words