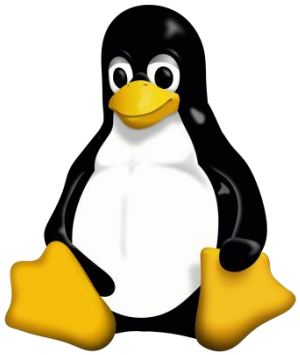
Linux Fundamentals

VERSION 3



Part 3

Eng Ali Mohammad. Bani Bakkar

Control Operators

semicolon

- You can put two or more commands on the same line separated by a semicolon **;** .

$ echo Hello

Hello

$ echo World

World

$ echo Hello ; echo World

Hello

World

ampersand

- When a line ends with an ampersand **&**, the shell will not wait for the command to finish.
- You will get your shell prompt back, and the command is executed in background. You will get a message when this command has finished executing in background.

$ sleep 20 & ls -1

[1] 7925

$

...wait 20 seconds...

$

[1]+ Done sleep 20

$? dollar question mark

- The exit code of the previous command is stored in the shell variable **$?**. Actually **$?** is a shell parameter and not a variable, since you cannot assign a value to **$?**.

```
$ touch file1
$ echo $?
0
$ rm file1
$ echo $?
0
$ rm file1
rm: cannot remove `file1': No such file or directory
$ echo $?
1
$
```

&& double ampersand

- The shell will interpret **&&** as a **logical AND**. When using **&&** the second command is executed only if the first one succeeds (returns a zero exit status).

$ echo first && echo second

first

second

$ zecho first && echo second

-bash: zecho: command not found

- Another example of the same **logical AND** principle. This example starts with a working **cd** followed by **ls**, then a non-working **cd** which is **not** followed by **ls**.

$ cd gen && ls

file1 file3 File55 fileab FileAB fileabc

file2 File4 FileA Fileab fileab2

$ cd gen && ls

-bash: cd: gen: No such file or directory

|| double vertical bar

- The **||** represents a **logical OR**. The second command is executed only when the first command fails (returns a non-zero exit status).

$ echo first || echo second ; echo third

first

third

$ zecho first || echo second ; echo third

-bash: zecho: command not found

second

third

- Another example of the same **logical OR** principle.

$ cd gen || ls

$ cd gen || ls

-bash: cd: gen: No such file or directory

file1 file3 File55 fileab FileAB fileabc

file2 File4 FileA Fileab fileab2

combining && and ||

- You can use this logical AND and logical OR to write an **if-then-else** structure on the command line. This example uses **echo** to display whether the **rm** command was successful.

$ rm file1 && echo It worked! || echo It failed!

It worked!

$ rm file1 && echo It worked! || echo It failed!

rm: cannot remove `file1': No such file or directory

It failed!

pound sign

- Everything written after a **pound sign** (#) is ignored by the shell. This is useful to write a **shell comment**, but has no influence on the command execution or shell expansion.

$ mkdir test # we create a directory

end of line backslash

- Lines ending in a backslash are continued on the next line. The shell does not interpret the newline character and will wait on shell expansion and execution of the command line until a newline without backslash is encountered.

$ echo This command line \

> is split in three \

> parts

This command line is split in three parts

Shell Variables

$ dollar sign

- The shell will look for an **environment variable** named like the string following the **dollar sign** and replace it with the value of the variable (or with nothing if the variable does not exist).
- shell variables are case sensitive!
- These are some examples using $HOSTNAME, $USER, $UID, $SHELL, and $HOME.

$ echo This is the $SHELL shell

This is the /bin/bash shell

$ echo This is $SHELL on computer $HOSTNAME

This is /bin/bash on computer RHELv4u3.localdomain

$ echo The userid of $USER is $UID

The userid of ali is 500

$ echo My homedir is $HOME

My homedir is /home/ali

Environment Variables
$env commandecho

```
Output
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35;...
LESSCLOSE=/usr/bin/lesspipe %s %s
LANG=en_US
S_COLORS=auto
XDG_SESSION_ID=5
USER=linuxize
PWD=/home/linuxize
HOME=/home/linuxize
SSH_CLIENT=192.168.121.1 34422 22
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop
SSH_TTY=/dev/pts/0
MAIL=/var/mail/linuxize
TERM=xterm-256color
SHELL=/bin/bash
SHLVL=1
LANGUAGE=en_US:
LOGNAME=linuxize
XDG_RUNTIME_DIR=/run/user/1000
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
LESSOPEN=| /usr/bin/lesspipe %s
_=/usr/bin/printenv
```


Email : alli_m_alqadri@hotmail.com

- Env
- Echo $PATH
- LS (WHERE DOES IT SEARCH?)
- Make a file (content hello)
- Try to run afile in different location
- Add a location of a file to apath
- Export PATH=location:$PATH
- Now run the file(with out ./)

creating variables

- This example creates the variable **$MyVar** and sets its value. It then uses **echo** to verify the value.

$ MyVar=555

$ echo $MyVar

555

How to add variables and print the sum?

- A=“ali”
- Echo $a
- Bash
- Echo $a
- Exit
- Echo $a
- Now---- exit
- Export A=“ali”
- Echo $a
- Bash
- Echo $a
- Exit
- Echo $a
- Now --- logout
- Echo $a
- env

Set an Environment Variable in Linux Permanently

If you wish a variable to persist after you close the shell session, you need to set it as an environmental variable permanently. You

1. To set permanent environment variables for a single user, edit the **.bashrc** file:

sudo nano ~/.bashrc

2. Write a line for each variable you wish to add using the following syntax:

export [VARIABLE_NAME]=[variable_value]

3. Save and exit the file. The changes are applied after you restart the shell. If you want to apply the changes during the current s

source ~/.bashrc

4. To set permanent environment variables for all users, create an **.sh** file in the **/etc/profile.d** folder:

sudo nano /etc/profile.d/[filename].sh

5. The syntax to add variables to the file is the same as with **.bashrc**:

6. Save and exit the file. The changes are applied at the next logging in

System Variable	Meaning	To View Variable Value Type
BASH_VERSION	Holds the version of this instance of bash.	echo $BASH_VERSION
HOSTNAME	The name of the your computer.	echo $HOSTNAME
CDPATH	The search path for the cd command.	echo $CDPATH
HISTFILE	The name of the file in which command history is saved.	echo $HISTFILE
HISTFILESIZE	The maximum number of lines contained in the history file.	echo $HISTFILESIZE
HISTSIZE	The number of commands to remember in the command history. The default value is 500.	echo $HISTSIZE
HOME	The home directory of the current user.	echo $HOME

IFS	The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the read builtin command. The default value is <space><tab><newline>.	echo $IFS
LANG	Used to determine the locale category for any category not specifically selected with a variable starting with LC_.	echo $LANG
PATH	The search path for commands. It is a colon-separated list of directories in which the shell looks for commands.	echo $PATH
PS1	Your prompt settings.	echo $PS1
TMOUT	The default timeout for the read builtin command. Also in an interactive shell, the value is interpreted as the number of seconds to wait for input after issuing the command. If not input provided it will logout user.	echo $TMOUT
TERM	Your login terminal type.	echo $TERM export TERM=vt100
SHELL	Set path to login shell.	echo $SHELL
DISPLAY	Set X display name	echo $DISPLAY export DISPLAY=:0.1
EDITOR	Set name of default text editor.	export EDITOR=/usr/bin/vim

Eng Ali Mohammad. Bani Bakkar
Email : ali_m_albakkar@hotmail.com

quotes

- Notice that double quotes still allow the parsing of variables, whereas single quotes prevent this.

$ MyVar=555

$ echo $MyVar

555

$ echo "$MyVar"

555

$ echo '$MyVar'

$MyVar

- The bash shell will replace variables with their value in double quoted lines, but not in single quoted lines.

$ city=Burtonville

$ echo "We are in $city today."

We are in Burtonville today.

$ echo 'We are in $city today.'

We are in $city today.

unset

- Use the **unset** command to remove a variable from your shell environment.

$ MyVar=8472

$ echo $MyVar

8472

$ unset MyVar

$ echo $MyVar

$PATH

- The **$PATH** variable is determines where the shell is looking for commands to execute (unless the command is builtin or aliased).

$ echo $PATH

/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:

Exercise

- Unset PATH

delineate variables

- Until now, we have seen that bash interprets a variable starting from a dollar sign, continuing until the first occurrence of a non-alphanumeric character that is not an underscore. In some situations, this can be a problem. This issue can be resolved with curly braces like in this example.

$ prefix=Super

$ echo Hello $prefixman and $prefixgirl

Hello and

$ echo Hello ${prefix}man and ${prefix}girl

Hello Superman and Supergirl

unbound variables

- The example below tries to display the value of the **$MyVar** variable, but it fails because the variable does not exist. By default the shell will display nothing when a variable is unbound (does not exist).

$ echo $MyVar

- There is, however, the **nounset** shell option that you can use to generate an error when a variable does not exist.

$ set -u

$ echo $Myvar

- bash: Myvar: unbound variable
- $ set +u

$ echo $Myvar

history

- To see older commands, use **history** to display the shell command history
- **History n** to see the last n commands).
- $ **history 10**
- 38 mkdir test
- 39 cd test
- 40 touch file1
- 41 echo hello > file2
- 42 echo It is very cold today > winter.txt
- 43 ls
- 44 ls -l
- 45 cp winter.txt summer.txt
- 46 ls -l
- 47 history 10

History !n

- When typing **!** followed by the number preceding the command you want repeated, then the

shell will echo the command and execute it.

$ **!43**

ls

file1 file2 summer.txt winter.txt

History

- All commands are kept in .bash_history
- Cat .bash_history

$HISTSIZE

- The $HISTSIZE variable determines the number of commands that will be remembered in your current environment. Most distributions default this variable to 500 or 1000.

$**echo $HISTSIZE**

prevent recording a command

- You can prevent a command from being recorded in **history** using a space prefix.

$ **echo abc**

abc

$ **echo def**

def

$ **echo ghi**

ghi

$ **history 3**

9501 echo abc

9502 echo ghi

9503 history 3

File Globbing

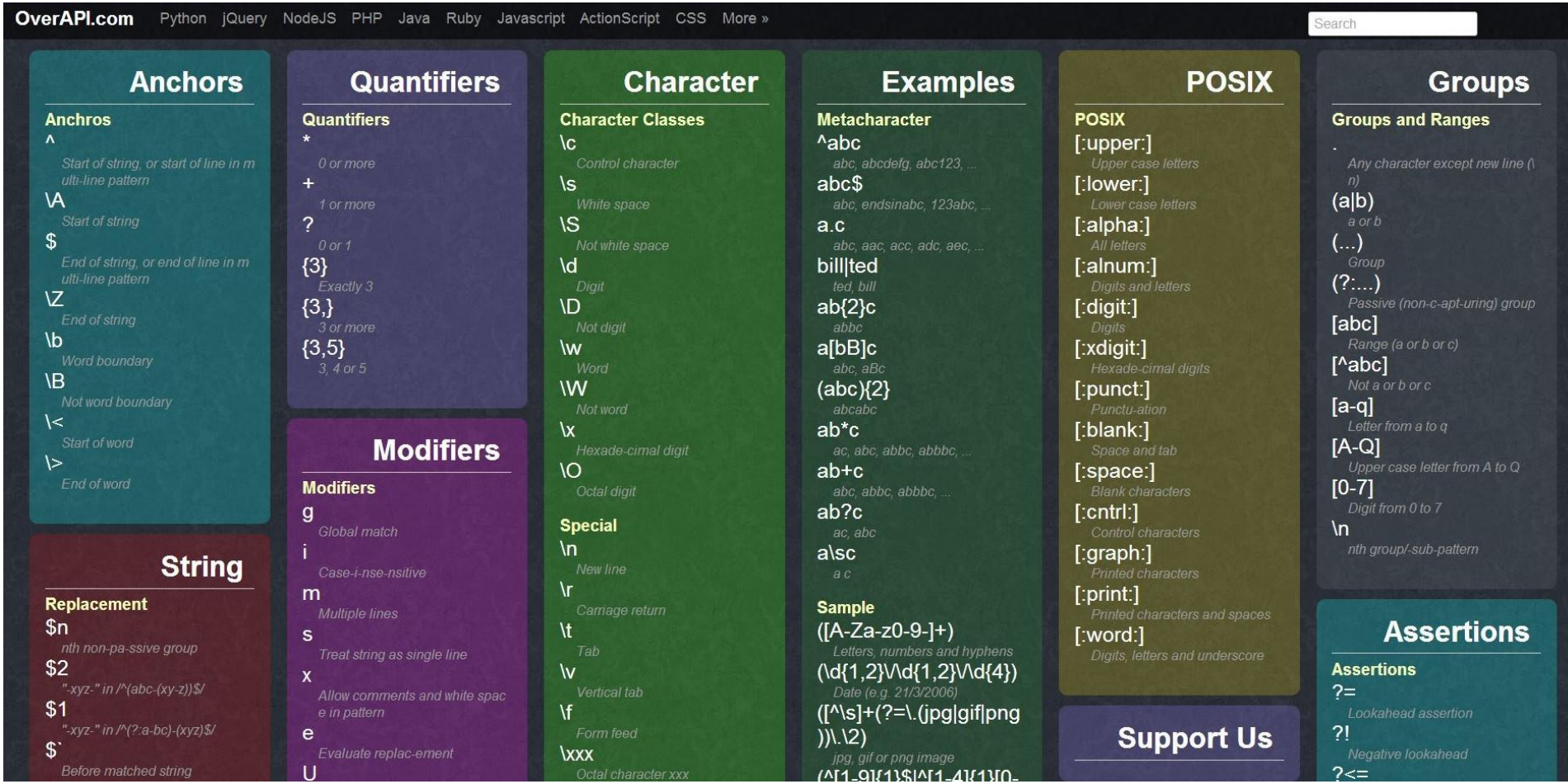
file globbing

- The shell is also responsible for **file globbing** (or dynamic filename generation).

Regular Expressions (RegEx)

Usage

- Regexps are acronyms for regular expressions. Regular expressions are special characters or sets of characters that help us to search for data and match the complex pattern.


OverAPI.com
Python
jQuery
NodeJS
PHP
Java
Ruby
Javascript
ActionScript
CSS
More »
Search
Anchors
Anchros
^
Start of string, or start of line in m ulti-line pattern
\A
Start of string
$
End of string, or end of line in m ulti-line pattern
\Z
End of string
\b
Word boundary
\B
Not word boundary
\<
Start of word
\>
End of word
String
Replacement
$n
nth non-pa-ssive group
$2
"-xyz-" in /^(abc-(xy-z))$/
$1
"-xyz-" in /^(?:a-bc)-(xyz)$/
$`
Before matched string
Quantifiers
Quantifiers
*
0 or more
+
1 or more
?
0 or 1
{3}
Exactly 3
{3,}
3 or more
{3,5}
3, 4 or 5
Modifiers
Modifiers
g
Global match
i
Case-i-nse-nsitive
m
Multiple lines
s
Treat string as single line
x
Allow comments and white spac e in pattern
e
Evaluate replac-ement
U
Character
Character Classes
\c
Control character
\s
White space
\S
Not white space
\d
Digit
\D
Not digit
\w
Word
\W
Not word
\x
Hexade-cimal digit
\O
Octal digit
Special
\n
New line
\r
Carriage return
\t
Tab
\v
Vertical tab
\f
Form feed
\xxx
Octal character xxx
Examples
Metacharacter
^abc
abc, abcdefg, abc123, ...
abc$
abc, endsinabc, 123abc, ...
a.c
abc, aac, acc, adc, aec, ...
bill|ted
ted, bill
ab{2}c
abbc
a[bB]c
abc, aBc
(abc){2}
abcabc
ab*c
ac, abc, abbc, abbbc, ...
ab+c
abc, abbc, abbbc, ...
ab?c
ac, abc
a\sc
a c
Sample
([A-Za-z0-9-]+)
Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})
Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)
jpg, gif or png image
POSIX
POSIX
[:upper:]
Upper case letters
[:lower:]
Lower case letters
[:alpha:]
All letters
[:alnum:]
Digits and letters
[:digit:]
Digits
[:xdigit:]
Hexade-cimal digits
[:punct:]
Punctu-ation
[:blank:]
Space and tab
[:space:]
Blank characters
[:cntrl:]
Control characters
[:graph:]
Printed characters
[:print:]
Printed characters and spaces
[:word:]
Digits, letters and underscore
Support Us
Groups
Groups and Ranges
.
Any character except new line (\ n)
(a|b)
a or b
(...)
Group
(?:...)
Passive (non-c-apt-uring) group
[abc]
Range (a or b or c)
[^abc]
Not a or b or c
[a-q]
Letter from a to q
[A-Q]
Upper case letter from A to Q
[0-7]
Digit from 0 to 7
\n
nth group/-sub-pattern
Assertions
Assertions
?=
Lookahead assertion
?!
Negative lookahead
?<=

.	Single character wildcar	s...r
\	Escape character	\.
^	The line must start with	^c
[ ]	Any single character in the brackets	[ch]
[^]	NOT any of the single characters in the brackets	[^ch]
$	The line ends with	s$
*	The preceding character can occur **zero** or **more** times	colou*r
?	The preceding character can occur **zero** or **one** time[1]	colou?r
+	The preceding character can occur **one** or **more** times[1]	colou+r
|	OR operator[1]	chips|salsa

[1] – Extended regular expression requires the -E option

Using “.” (dot)

Using “.” we can find a string if we do not know the exact string, or we just remember only the start and end of the string, we can use “.” As a missing character, and it will fill that missing character.

```
fruits_file=`cat fruit.txt | grep App.e`
```

Using “^” (caret) to match the beginning of the string

Using “^”, we can find all the strings that start with the given character. Let’s see an example for a better understanding. Here we are trying to find all the fruit names that start with the letter B:

```
fruits_file=`cat fruit.txt | grep ^B`
```

Using “$” (dollar sign) to match the ending of the string

- Using “$” we can find all the strings that end with the given character. Let’s see an example for a better understanding. Here we are trying to find all the fruit’s names that end with the letter e:
- fruits_file=`cat fruit.txt | grep e$`

Using “*” (an asterisk) to find any number of repetitions of a string

- Using “*”, we can match up to zero or more occurrences of the character of the string. Let’s see an example for a better understanding. Here we are trying to find all the fruit’s names that has one or more occurrences of ‘ap’ one after another in it.

- fruits_file=`cat fruit.txt | grep ap*le`

Using “\” (a backslash) to match the special symbol

- Using “\” with special symbols like whitespace (” “), newline(“\n”), we can find strings from the file. Let’s see an example for a better understanding. Here we are trying to find all the fruit’s names that have space in their full names
- fruits_file=`cat fruit.txt | grep “\ “`

Using “()” (braces) to match the group of regexp.

- Using “()”, we can find matched strings with the pattern in the “()”. Let’s see an example for a better understanding. Here we are trying to find all the fruit’s names that have space in their full name
- fruits_file=`cat fruit.txt | grep -E “(fruit)”`

Using “?”(question mark) to find all the matching characters

- Using “?”, we can match 0 or 1 repetitions of the preceding. For example, if we do something like this: ab? It will match either ‘a’ or ‘ab’. Let’s see another example for better understanding. Here we are trying to find all the fruit’s names that have the character ‘Ch’ in them.
- fruits_file=`cat fruit.txt | grep -E Ch?`

* asterisk

The asterisk * is interpreted by the shell as a sign to generate filenames, matching the asterisk

to any combination of characters (even none). When no path is given, the shell will use

filenames in the current directory.

$ ls

file1 file2 file3 File4 File55 FileA fileab Fileab FileAB fileabc

$ ls File*

File4 File55 FileA Fileab FileAB

$ ls file*

file1 file2 file3 fileab fileabc

$ ls *ile55

- File55

$ ls F*ile55

- File55

$ ls F*55

File55

? question mark

Similar to the asterisk, the question mark **?** is interpreted by the shell as a sign to generate

filenames, matching the question mark with exactly one character.

$ ls

file1 file2 file3 File4 File55 FileA fileab Fileab FileAB fileabc

$ ls File?

File4 FileA

$ ls Fil?4

File4

$ ls Fil??

File4 FileA

$ ls File??

File55 Fileab FileAB

[] square brackets (1)

The square bracket **[** is interpreted by the shell as a sign to generate filenames, matching any of the characters between **[** and the first subsequent **]**. The order in this list between the brackets is not important. Each pair of brackets is replaced by exactly one character.

$ ls

file1 file2 file3 File4 File55 FileA fileab Fileab FileAB fileabc

$ ls File[5A]

FileA

$ ls File[A5]

FileA

$ ls File[A5][5b]

File55

$ ls File[a5][5b]

File55 Fileab

$ ls File[a5][5b][abcdefghijklm]

ls: File[a5][5b][abcdefghijklm]: No such file or directory

$ ls file[a5][5b][abcdefghijklm]

fileabc

[] square brackets (2)

- You can also exclude characters from a list between square brackets with the exclamation
- mark **!**. And you are allowed to make combinations of these **wild cards**.

$ ls

file1 file2 file3 File4 File55 FileA fileab Fileab FileAB fileabc

$ ls file[a5][!Z]

fileab

$ ls file[!5]*

file1 file2 file3 fileab fileabc

a-z and 0-9 ranges

- The bash shell will also understand ranges of characters between brackets.

```
$ ls
file1 file3 File55 fileab FileAB fileabc
file2 File4 FileA Fileab fileab2
$ ls file[a-z]*
fileab fileab2 fileabc
$ ls file[0-9]
file1 file2 file3
$ ls file[a-z][a-z][0-9]*
fileab2
```

I/O redirection

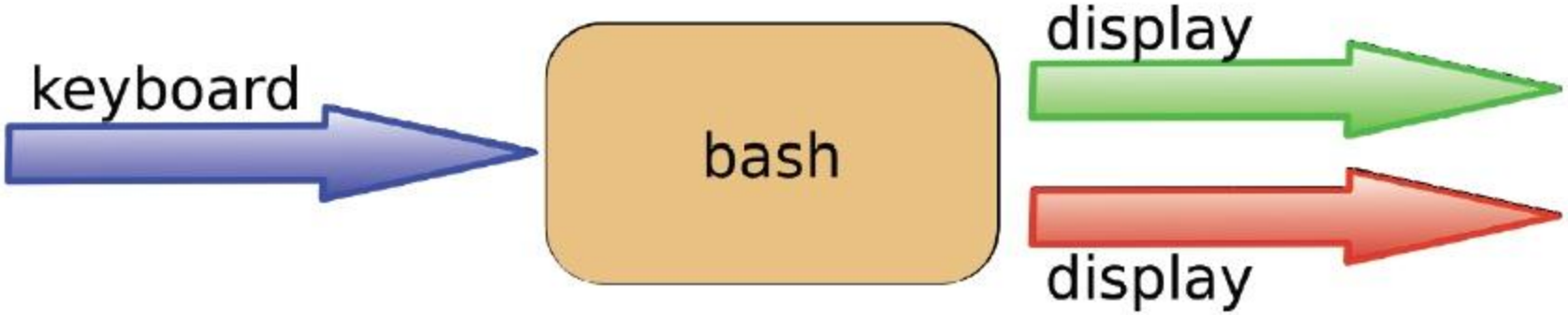
I/O redirection

- input/output redirection
- error streams.
- pipes.

stdin, stdout, and stderr

- The bash shell has three basic streams; it takes input from **stdin** (stream **0**), it sends output
- to **stdout** (stream **1**) and it sends error messages to **stderr** (stream **2**) .
- The drawing below has a graphical interpretation of these three streams.

- The keyboard often serves as **stdin**, whereas **stdout** and **stderr** both go to the display.



stdout

- **stdout** can be redirected with a **greater than** sign. While scanning the line, the shell will see the **>** sign and will clear the file.

$ echo It is cold today!

It is cold today!

$ echo It is cold today! > winter.txt

$ cat winter.txt

It is cold today!

output file is erased

- While scanning the line, the shell will see the > sign and **will clear the file**! Since this happens before resolving **argument 0**, this means that even when the command fails, the file will have been cleared!

$ cat winter.txt

It is cold today!

$ zcho It is cold today! > winter.txt

-bash: zcho: command not found

$ cat winter.txt

noclobber

- Erasing a file while using > can be prevented by setting the **noclobber** option.

$ cat winter.txt

It is cold today!

$ set -o noclobber

$ echo It is cold today! > winter.txt

-bash: winter.txt: cannot overwrite existing file

$ set +o noclobber

overruling noclobber

- The **noclobber** can be overruled with **>|**.

$ set -o noclobber

$ echo It is cold today! > winter.txt

-bash: winter.txt: cannot overwrite existing file

$ echo It is very cold today! >| winter.txt

$ cat winter.txt

It is very cold today!

>> append

- Use **>>** to **append** output to a file.

$ echo It is cold today! > winter.txt

$ cat winter.txt

It is cold today!

$ echo Where is the summer ? >> winter.txt

$ cat winter.txt

It is cold today!

Where is the summer ?

stderr

- Redirecting **stderr** is done with **2>**. This can be very useful to prevent error messages from cluttering your screen.
- The screenshot below shows redirection of **stdout** to a file, and **stderr** to **/dev/null**. Writing **1>** is the same as >.

$ find / > allfiles.txt 2> /dev/null

Example

Echo “jordan”>file1.txt

Ls –l file1.txt>file2.txt 2>fileerror.txt