LINUX BASICS

WHAT IS “THE SHELL”?

Simply put, the shell is an interface that takes your commands from the keyboard and gives them to the operating system to perform. In the old days, it was the only user interface available on a Unix computer. Nowadays, we have graphical user interfaces (GUIs) in addition to command line interfaces (CLIs) such as the shell.
On most Linux systems a program called bash (which stands for Bourne Again SHell, an enhanced version of the original Bourne shell program, sh, written by Steve Bourne) acts as the shell program. There are several additional shell programs available on a typical Linux system. These include: ksh, tcsh and zsh.

What's an xterm, gnome-terminal, konsole, etc.?

These are called "terminal emulators." They are programs that put a window up and let you interact with the shell. There are a bunch of different terminal emulators you can use. Most Linux distributions supply several, such as: xterm, rxvt, konsole, kvt, gnome-terminal, nxterm, and eterm.

STARTING A TERMINAL

Your window manager probably has a way to launch programs from a menu. Look through the list of programs to see if anything looks like a terminal emulator program. In KDE, you can find "konsole" and "terminal" on the Utilities menu. In Gnome, you can find "color xterm," "regular xterm," and "gnome-terminal" on the Utilities menu. You can start up as many of these as you want and play with them. While there are a number of different terminal emulators, they all do the same thing. They give you access to a shell session. You will probably develop a preference for one, based on the different bells and whistles each one provides.

Testing the Keyboard

Ok, let's try some typing. Bring up a terminal window. You should see a shell prompt that contains your user name, the name of the machine and the name of the current directory (~ means that you are in your home) followed by a dollar sign. Something like this:

[user@pc133 ~]$

Excellent! Now type some nonsense characters and press the enter key.

[user@pc133 ~]$ kdkjflajfks

If all went well, you should have gotten an error message complaining that it cannot understand you:

bash: kdkjflajfks: command not found

Wonderful! Now press the up-arrow key. Watch how our previous command "kdkjflajfks" returns. Yes, we have command history. Press the down-arrow and we get the blank line again.

Recall the "kdkjflajfks" command using the up-arrow key if needed. Now, try the left and right-arrow keys. You can position the text cursor anywhere in the command line. This allows you to easily correct mistakes.

Using the Mouse

Even though the shell is a command line interface, you can still use the mouse for several things, if you have a 3-button mouse (and you should have a 3-button mouse if you want to use Linux).
First, you can use the mouse to scroll backwards and forwards through the output of the terminal window. To demonstrate, hold down the enter key until it scrolls off the window. Now, with your mouse, you can use the scroll bar at the side of the terminal window to move the window contents up and down.
Next, you can copy text with the mouse. Drag your mouse over some text (for example, "kdkjflajfks" right here on the browser window) while holding down the left button. The text should highlight. Now move your mouse pointer to the terminal window and press the middle mouse button. The text you highlighted in the browser window should be copied into the command line.

You're not logged in as root, are you?

Don't operate the computer as the superuser. You should only become the superuser when absolutely necessary. Doing otherwise is dangerous, stupid, and in poor taste. Create a user account for yourself now!

FILE SYSTEM ORGANIZATION

Like that legacy operating system, the files on a Linux system are arranged in what is called a hierarchical directory structure. This means that they are organized in a tree-like pattern of directories (called folders in other systems), which may contain files and other directories. The first directory in the file system is called the root directory. The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on.

The tree of the file system starts at the trunk or slash, indicated by a forward slash (/). This directory, containing all underlying directories and files, is also called the root directory or "the root" of the file system.

Directories that are only one level below the root directory are often preceded by a slash, to indicate their position and prevent confusion with other directories that could have the same name.

/ (root directory)

↓

↓ ↓ ↓ ↓ ↓ ↓ ↓

/root **/bin** **/usr** **/boot** **/home** **/var** **...**

↓

↓ ↓ ↓ ↓

/home/user_1 **/home/user_2** **/home/user_3** **/home/user_4**

Content of some subdirectories of the root directory

/bin Common programs, shared by the system, the system administrator and the users.
/boot The startup files and the kernel, vmlinuz.
/dev Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home Home directories of the common users.
/lib Library files, includes files for all kinds of programs needed by the system and the users.
/root The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the root user.
/sbin Programs for use by the system and the system administrator.
/usr Programs, libraries, documentation etc. for all user-related programs.
/var Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet,...

SHELL CONFIGURATION

A BASH shell has general settings for all users, but each one can customize its shell modifying conveniently some files. A BASH shell has five configuration files:

- /etc/profile
- /etc/bashrc
- /home/user/.bash_profile
- /home/user/.bashrc
- /home/user/.bash_logout

The first and the second files contain general settings for all users, in particular for the initialization of enviromental variables and for the execution of startup programs, while the others are specific for each user.
The file .bash_profile contains the login user settings for the shell initialization, such as the home path, and is executed only when the user signs in. In the .bashrc file the user can define aliases and enviromental variables. This is executed at the login and everytime a shell is opened. At last the file .bash_logout contains the logout settings and is executed when a user signs out.

LOOKING AROUND

Now we're going to take a tour of your Linux system and, along the way, learn some things about what makes it tick. But before we begin, I have to teach you some tools that will come in handy during our adventure. These are:

man (manual of a command)
ls (list files and directories)

man calls the manual for a given command. It provides informations about the commands usage and about its possible options. Let't try this:

[user@pc133 ~]$ man ls

As you can see the *ls* command is used to list the contents of a directory. It is probably the most commonly used Linux command. It can be used in a number of different ways. If no arguments are supplied, it prints a list of files/directories contained in the current directory:

[user@pc133 ~]$ ls

(oops! type *q* to close the manual!) otherwise if you specify a directory name it lists its content:

[user@pc133 ~]$ ls /bin

(Note that before the directory bin we need a /, since it is in the root directory. For directories you will create don't put a / before the name!)

You can also specify more than one directory:

[user@pc133 ~]$ ls /etc /bin

With the option *–l* it list the files in long format:

[user@pc133 ~]$ ls -l /bin

This example also point out an important concept about commands. Most commands operate like this:

command -options arguments

where command is the name of the command, -options is one or more adjustments to the command's behavior, and arguments is one or more "things" upon which the command operates.

Now let's have a closer look to long format. Using the –l option, you will get a file listing that contains a wealth of information about the files being listed. Here's an example:

-rw-------	1	bshotts	bshotts	576	Apr 17 1998	weather.txt
drwxr-xr-x	6	bshotts	bshotts	1024	Oct 9 1999	web_page
-rw-rw-r--	1	bshotts	bshotts	276480	Feb 11 20:41	web_site.tar
-rw-------	1	bshotts	bshotts	5743	Dec 16 1998	xmas_file.txt
↓	↓	↓	↓	↓	↓	↓
file permissions	number of files	file's owner	group	file's size (byte)	last modification time	filename

Important facts about file names

- File names that begin with a full stop character are hidden. This only means that *ls* will not list them unless you use *ls -a*. When your account was created, several hidden files were placed in your home directory to configure things for your account. In addition, some applications will place their configuration and settings files in your home directory as hidden files.

- File names in Linux, like Unix, are case sensitive. The file names "File1" and "file1" refer to different files.

- Linux has no concept of a "file extension" like legacy operating systems. You may name files any way you like. The contents/purpose of a file is determined by other means.

- While Linux supports long file names which may contain embedded spaces and punctuation characters, limit the punctuation characters to period, dash, and underscore. Most importantly, do not embed spaces in file names. If you want to represent spaces between words in a file name, use underscore characters instead. You will thank yourself later.

I/O REDIRECTION

In this paragraph, we will explore a powerful feature used by many command line programs called input/output redirection. As we have seen, many commands such as ls print their output on the display. This does not have to be the case, however. By using some special notation we can redirect the output of many commands to files, devices, and even to the input of other commands.

Standard Output

Most command line programs that display their results do so by sending their results to a facility called standard output. By default, standard output directs its contents to the display. To redirect standard output to a file, the ">" character is used like this:

[user@pc133 ~]$ ls > file_list.txt

In this example, the *ls* command is executed and the results are written in a file named file_list.txt. Since the output of *ls* was redirected to the file, no results appear on the display. If the file does not exist when the command is run, it will be created.
Each time the command above is repeated, file_list.txt is overwritten (from the beginning) with the output of the command ls. You can see the content of the file using the *cat* command, which displays the file on screen:

[user@pc133 ~]$ cat file_list.txt

If you want the new results to be appended to the file instead, use ">>" like this:

[user@pc133 ~]$ ls –l >> file_list.txt

When the results are appended, the new results are added to the end of the file, thus making the file longer each time the command is repeated.

Standard Input

Many commands can accept input from a facility called standard input. By default, standard input gets its contents from the keyboard, but like standard output, it can be redirected. To redirect standard input from a file instead of the keyboard, the "<" character is used like this:

[user@pc133 ~]$ sort < file_list.txt

In the above example we used the *sort* command to process the contents of file_list.txt. The *sort* command orders alphabetically the lines of the file.The result is output on the display since the standard output is not redirected in this example. We could redirect standard output to another file like this:

[user@pc133 ~]$ sort < file_list.txt > sorted_file_list.txt

As you can see, a command can have both its input and output redirected. Be aware that the order of the redirection does not matter. The only requirement is that the redirection operators (the "<" and ">") must appear after the other options and arguments in the command.

Pipes

By far, the most useful and powerful thing you can do with I/O redirection is to connect multiple commands together with what are called pipes. With pipes, the standard output of one command is fed into the standard input of another:

[user@pc133 ~]$ ls -l | sort

In this example, the output of the *ls* command is fed into *sort*. Compare this result with that of *ls –l* alone.

FILES & DIRECTORIES

Absolute and relative path

A path, which is the way you need to follow in the tree structure to reach a given file, can be written starting from the trunk of the tree (the / or root directory). In this case, the path starts with a slash and is called absolute path. In the other cases, the path starts from your working directory and is called relative.

Directories

The command to create a directory is *mkdir* (make directory), followed by the direcory name (or names if we want to create more than one directory):

[user@pc133 ~]$ mkdir example workdir

Now in your home you have two directory named “example” and “workdir”. Use *ls* if you don’t believe... Now you can use the command *cd* (change directory) to go in one of these. Notice that the name of the directory appears in the prompt:

[user@pc133 ~]$ cd example
[user@pc133 example]$

If you want to return to your home you can use one of the equivalent commands:

[user@pc133 example]$ cd .. (the double full stop indicates the upper level directory)
[user@pc133 example]$ cd /home/user

The first is an example of relative path, the second is an absolute path. If you want instead to go from the example directory to the workdir directory you can use:

[user@pc133 example]$ cd ../workdir
[user@pc133 example]$ cd /home/user/workdir

If you don’t know in which directory you are, use the *pwd* command:

[user@pc133 workdir]$ pwd
/home/user/workdir

Files

Now let’s suppose you are in the directory workdir you want to create an empty file in the directory example. You can do this without change directory, but simply specifying the path and the name of the file after the command *touch*:

[user@pc133 workdir]$ touch ../example/newfile

You can check it with:

[user@pc133 workdir]$ cat ../example/newfile

which prints on screen the content of file (of course this file is empty...).
Now it’s time to introduce a new command: *echo*, which allows to display a string or a variable on screen:

[user@pc133 workdir]$ echo “Hello World”
Hello World

Now that we have print ‘Hello World’ on screen we can use redirection to write it on the file:

[user@pc133 workdir]$ echo “Hello World” > file.dat
[user@pc133 workdir]$ cat file.dat
Hello World
[user@pc133 workdir]$ echo “Hello World” >> file.dat
[user@pc133 workdir]$ cat file.dat
Hello World
Hello World

Copy, move and remove files and directories

If you want to delete a file use the command *rm*. The file can’t be recovered.

[user@pc133 workdir]$ rm ../example/newfile

To delete a directory you should add the *–r* option, which means that the command *rm* acts recursively on each element in the directory:

[user@pc133 workdir]$ ls ..
[user@pc133 workdir]$ rm –r ../example
[user@pc133 workdir]$ cd ..
[user@pc133 ~]$ rm –r workdir
[user@pc133 ~]$ ls

Now your home should be empty. Let’s restart!

[user@pc133 ~]$ mkdir dir1 dir2 dir3
[user@pc133 ~]$ mkdir dir1/tmp
[user@pc133 ~]$ touch dir1/tmp/file1 dir2/file2 file3

For copy (*cp*) and move (*mv*) we can reduce to four cases:

- *cp (mv) file_1 file2*	copy or move file_1 to file_2. If file_2 exists it is overwritten, otherwise it will be created.
- *cp (mv) file_1 directory*	copy or move file_1 in the directory.
- *cp (mv) file_1 directory/file_2*	copy or move file_1 to file_2 in the directory. If file_2 exists it is overwritten, otherwise it will be created.
- *cp –r (mv) directory_1 directory_2*	copy or move directory_1 to directory_2. directory_2 should not exist.

Let's try!

[user@pc133 ~]$ cp -r dir1 dir4
[user@pc133 ~]$ mv dir4/tmp/file1 .
[user@pc133 ~]$ cp dir2/file2 dir4/tmp/file5
[user@pc133 ~]$ mv file1 file3 dir3
[user@pc133 ~]$ mv dir2 dir4/tmp/file5 dir3
[user@pc133 ~]$ mv dir3/dir4/tmp/file5 .
[user@pc133 ~]$ ls –R

You shold have noticed that:
- the full stop means "the current directory"
- if you specify more than two entries after mv or copy, the last entry must be a directory
- ls –R list recursively all files in all directories.

TAB COMPLETION

Command-line completion (also tab completion) is a common feature of command line interpreters, in which the program automatically fills in partially typed commands or filenames.
Command-line completion allows you to type the first few characters of a command, program, or filename, and press a completion key (normally Tab ⭾) to fill in the rest of the name. Then press *enter* to run the command or open the file.

[user@pc133 ~]$ touch example1
[user@pc133 ~]$ ls ex   Tab ⭾
[user@pc133 ~]$ ls example1

If more entries are possible they are listed on the line below pressing Tab twice. Then you should complete the name until there will be only one possible entry.

[user@pc133 ~]$ touch example1 example2
[user@pc133 ~]$ ls ex   Tab ⭾ Tab ⭾
example1 example2
[user@pc133 ~]$ ls example2

ENVIROMENTAL VARIABLES

Environment variables are a set of dynamic named object that contains data used by one or more applications. In simple terms, it is a variable with a name and a value. For example the value of an environmental variable can be the location of all executable files in the filesystem, the default editor that should be used, or the system locale settings.
We can use *echo* to display the values of enviromentals variables:

[user@pc133 ~]$ echo $USER *# print the username*
[user@pc133 ~]$ echo $SHELL *# print shell type*
[user@pc133 ~]$ echo $PWD *# print the path of the current directory*
[user@pc133 ~]$ echo $OLDPWD *# print the path of the previous directory*
[user@pc133 ~]$ cd $OLDPWD *# return to the previous directory*
[user@pc133 ~]$ env *# print a list of all enviromental variables*

WILDCARDS

The usage of wildcards is a nice feature that the shell provides us. Simple characters such as * or ? can make our bash life much more easier than we have ever imagined. Wildcards are special characters that allow us to select a big number of files/directories matching a specified pattern. These wildcards can be used with most of the common Linux commands.

? (question mark)
This can represent any single character. If you specified something at the command line like "*hd?*" Linux would look for *hda*, *hdb*, *hdc* and every other letter/number between a-z, 0-9.

*** (asterisk)**
This can represent any number of characters (including zero, in other words, zero or more characters). If you specified a *"cd*"*, it would use "*cda*", "*cdrom*", "*cdrecord*" and anything that starts with “*cd*” also including “*cd*” itself. "*m*l*" could by *mill*, *mull*, *ml*, and anything that starts with an m and ends with an l.

[ ] (square brackets)

It specifies a range. If you did m[a,o,u]m it can become: mam, mum, mom. If you did: m[a-d]m it can become anything that starts and ends with m and has a, b, c or d in between. For example, these would work: mam, mbm, mcm, mdm. Spaces are not allowed after the commas (or anywhere else).

{ } (curly brackets)

The terms in brackets are separated by commas and each term must be the name of something or a wildcard. This wildcard will copy anything that matches either wildcard(s), or exact name(s). Spaces are not allowed after the commas (or anywhere else).

[!]
This construct is similar to the [ ] construct, except rather than matching any characters inside the brackets, it'll match any character, as long as it is not listed between the brackets.
For example *rm myfile[!9]* will remove all myfiles* (ie. myfiles1, myfiles2, etc) but won't remove a file with the number 9 anywhere within its name.

Some examples:

[user@pc133 ~]$ mkdir dir
[user@pc133 ~]$ touch dir/{pippo,pluto,pluto.dat,output,input}
[user@pc133 ~]$ touch dir/pippo.{log,dat }
[user@pc133 ~]$ touch dir/ pippo[1-3].dat
[user@pc133 ~]$ cd dir
*[user@pc133 dir]$ ls **
    pippo pippo.log pippo.dat pippo1.dat pippo2.dat pippo3.dat pluto pluto.dat output
*[user@pc133 dir]$ ls pippo**
    pippo pippo.log pippo.dat pippo1.dat pippo2.dat pippo3.dat
*[user@pc133 dir]$ ls *dat*
    pippo.dat pippo1.dat pippo2.dat pippo3.dat pluto.dat
[user@pc133 dir]$ ls pippo?.dat
    pippo1.dat pippo2.dat pippo3.dat
[user@pc133 dir]$ ls p???o
    pippo pluto
[user@pc133 dir]$ ls pippo[1,2].dat
    pippo1.dat pippo2.dat
[user@pc133 dir]$ ls pippo[1-3].dat
    pippo1.dat pippo2.dat pippo3.dat
[user@pc133 dir]$ ls pippo.{log,dat}
    pippo.log pippo.dat
[user@pc133 dir]$ ls {inp,out}put
    input output
[user@pc133 dir]$ mkdir dir{1,2,3}
[user@pc133 dir]$ touch dir{1,2,3}/file{1,2}
*[user@pc133 dir]$ ls **   <u>(NOTE: this lists also the content of directories in the current directory)</u>
    pippo pippo.log pippo.dat pippo1.dat pippo2.dat pippo3.dat pluto pluto.dat output
    dir1:
     file1 file2
    dir2:
     file1 file2
    dir3:
     file1 file2
*[user@pc133 dir]$ rm –r dir**

META-CHARACTERS

A metacharacter is a character that has a special meaning (instead of a literal meaning), then is interpreted by the shell. The main metacharacters are:

< > | ? ! ( ) [ ] { } $ & # ; || && >> << \ ˆ * blank tab

A \ before prevents the shell from interpreting a metacharacter.

[user@pc133 ~]$ echo $PWD
/home/user
[user@pc133 ~]$ echo \$PWD
$PWD

Also quotes prevent the shell from interpreting a metacharacter. Double quotes, instead, allow only $ to be interpreted.

[user@pc133 ~]$ echo '$PWD'
$PWD
[user@pc133 ~]$ echo "$PWD"
/home/user
[user@pc133 ~]$ ls ""*
*ls: *: No such file or directory*

MAIN SHELL SHORTCUTS

!!	re-executes the last command
!(string)	re-executes the last command beginning with the string
CTRL-u	delete the entire line from the cursor position to the begin of line
CTRL-k	delete the entire line from the cursor position to the end of line
←,→	move cursor to the left/right
↑,↓	run the command history backwards and forwards
home key, CTRL-a	move cursor to the begin of line
end key, CTRL-e	move cursor to the end of line
TAB	expand the name of files or commands
CTRL-d	logout
CTRL-c	exit from a command/program
CTRL-z	stop a command/program without kill it

SUMMARY OF THE MAIN SHELL SYMBOLS

~	home (F12)
.	in this directory
..	in the upper level directory
?	match any single character
*****	match any character or string
[ ]	match a range/list of single characters
{ }	match a list of elements
;	command separator in a sequence of commands
(meta-char)	prevent the shell from interpreting the following meta-character, treating it as text
' '	prevent the shell from interpreting the meta-characters in between, treating them as text
" "	prevent the shell from interpreting the meta-characters in between, but allowing variable and command substitutions
(command) &	run the command in background
(command1) | (command2)	concatenate commands
(command) < (file)	redirect standard input to the command
(command) > (file)	redirect standard output of the command to the file
(command) >> (file)	append standard output of the command at the end of the file

SUMMARY OF THE MAIN COMMANDS

man (command)	manual of the command (q to exit)
ls (file/dir)	list files/directories
	ls -a show hidden files
	ls -l list in long format
	ls -lt list in long format sorted by time modified
	ls -R list recursively all files in all directories
echo (string)	print the string
mkdir (dir)	create a directory
cd (dir)	change directory
	cd - return to previous directory
	cd return to home
pwd	write the path of current directory
touch (file)	create an empty file
mv (file1/dir1) (file2/dir2)	move a file/directory. If more than two entries are specified, the last must be a directory
cp (file1) (file2)	copy a file from path1 to path2. If more than two entries are specified, the last must be a directory
	cp -r (dir1) (dir2) copy a directory and its content
rm (file)	remove a file
	rmdir (dir) remove an empty directory
	rm -r (dir) remove a directory and its content

TEXT & FILES

The following commands allow you to operate on a file without open it.

wc (file)	print the number of lines of the file
cat (file)	print the file from top to bottom
tac (file)	print the file from bottom to top
colrm (n1) (n2) < (file)	for each line of the file, remove characters from (n1) to (n2)
cut -c(n1)-(n2) (file)	for each line of the file, remove characters external to the (n1) – (n2) range
paste (file1) (file2)	print files side by side
tr (xyzk) (abcd) < (file)	substitute (x) with (a), (y) with (b), (z) with (c), ...
	tr (char) "\n" < (file) substitute the character with a new line
diff (file1) (file2)	write differences between two files
	diff -y (file1) (file2) write differences using side by side output format
sort < (file)	sort alphabetically the lines of a file
	sort –n < (file) sort numerically the lines of a file
	sort –r < (file) sort in the reverse order

uniq (file) Given a sorted stream of data from standard input, it removes duplicate lines of data

tail –(n) (file) print the last (n) lines of file (default n=10)

tail –f (file) print the end of file

head –(n) (file) print the first (n) lines of file (default n=10)

grep (string) (file) extract the lines containing the string from the file

grep –A(n) (string) (file) extract the lines containing the string and the following (n) lines from the file

grep –B(n) (string) (file) extract the lines containing the string and the previous (n) lines from the file

grep -v (string) (file) extract the lines NOT containing the string from the file

grep ˆ(string) (file) extract the lines beginning with the string from the file

grep (string)$ (file) extract the lines ending with the string from the file

CHANGE FILE PERMISSIONS

The first column of the *ls –l* command output, is a sequence of 10 characters:

-rw-r--r-- 1 username ... file1
drwxr-xr-x 31 username ... file2

The first character indicates if we are dealing with a file (-) or with a directory (d), the other nine are divided in three sets of three characters. The first set indicates the permissions of the user (u), the second the permissions of the group (g) and the last those of other people (o). The three characters of each set represent, in order, the permissions of:

- read (r) the file/directory
- write (w) the file/directory
- execute (x) the file/directory

A dash (-) indicates that the user/group/others has not the corresponding permission.
To change the permissions use the command chmod:

chmode (set letter)±(permission letter) (filename)
chmode (set letter)=(characters) (filename)

examples:

chmod u+x file 1 ⇒ -rwxr--r-- 1 username ... file1
chmod o-rx file 2 ⇒ drwxr-x--- 31 username ... file2
chmod g=x file 1 ⇒ -rw---xr-- 1 username ... file1
chmod ugo=wr file 1 ⇒ -rw-rw-rw- 1 username ... file1

SSH & SCP

SSH is some kind of an abbreviation of Secure SHell. It is a protocol that allows secure connections between computers. In the most simple case, you can connect to a server that supports ssh with a syntax as short as this:

[user@pc133 ~]$ ssh your_server (for example: ssh hilbert)

Of course, *your_server* should be replaced by a hostname or an IP address of the server you want to connect to.
You can also specify a different username. See the following example:

[user@pc133 ~]$ ssh new_user@your_server   (for example: ssh pippo@hilbert)

The above will make ssh try to connect with the username “new_user” instead of “user”.

The SCP (Secure CoPy) command allows you to copy files over ssh connections. This is pretty useful if you want to transport files between computers, for example to backup something. The scp command can be used in three ways: to copy from a remote server to your computer (to your home in this case):

[user@pc133 ~]$ scp new_user@your_server:/home/new_user/filename   ~
(for example: scp pippo@pc138:~/filename   ~)

to copy from your computer to a remote server (to new_user’s home in this case):

[user@pc133 ~]$ scp filename new_user@your_server:/home/new_user/
(for example: scp filename pippo@pc138:~)

and to copy from a remote server to another remote server. In this case, the data are transferred directly between the servers; your own computer will only tell the servers what to do.

[user@pc133 ~]$ scp new_user@your_server:~/filename   another_user@your_server:~
(for example: scp pippo@pc135:~/filename   pluto@pc129:~)

If you want to copy a directory you should use *scp –r.*

CREATE PUBLIC/PRIVATE KEY PAIRS

SSH (Secure Shell) can be set up with public/private key pairs so that you don't have to type the password each time. Because SSH is the transport for other services such as SCP (secure copy), SFTP (secure file transfer), and other services (CVS, etc), this can be very convenient and save you a lot of typing.
On your local home do:

cd ~/.ssh
ssh-keygen –N “” –t rsa –f id_rsa

The option –N set the new password (nothing in this case), -t set the type of key and –f set the output filename where the keys are written.
Now in the directory .ssh you shoud have two files: id_rsa, containing the private key and id_rsa.pub, containing the public key.
At last you have to copy the file id_rsa.pub to the file ~/.ssh/authorized_keys on your remote hosts (if this file already exists simply add the public key at the end of this file).
NOTE: make sure that the file ~/.ssh/authorized_keys has the following permissions: -rw-------.

CREATE ALIASES

An alias is a simple name or abbreviations for a command, particularly useful when the command is often used or is very complex. You can create it using the alias command:

alias name="(command)" (*example: alias hilb="ssh hilbert"*)

and you can remove it with

unalias name (*example: unalias hilb*)

The aliases are deleted when a shell is closed, to make them permanent write the alias command in the .bashrc file (after the line "user aliases and functions"), then run the command **source .bashrc**

PROCESSES AND MEMORY

top	show all processes on a machine
ps -HU (user)	show the processes of a user on a machine
kill (id)	kill the process identified by the (id) (HANDLE WITH CARE) **kill -9 (id)** force killing of the process identified by the (id)
killall -u (user)	kill all processes of the user (HANDLE WITH EVEN MORE CARE)
du -c (path)	show informations about files/directories space usage in kb **du –c –B M (path)** show informations about files/directories space usage in Mb
df	show informations about partition memory usage

OTHER COMMANDS

ln -s (file1) (file2)	link file1 to file2
bc -l	calculator (write 'quit' to exit)
clear	clear screen
exit	exit from current session
logout	logout
which (filename)	locate a file and print its path
find (path) (filename)	descend recursively the directory tree from (path) searching the file
(command) | tee (file)	print the output of the command either on screen and in the file
whoami	print the username
history	print the recent command history
date	print date and time
sleep (time)	delay for a specific time (in seconds)
cal	print the calendar of current month
bg (program)	run the program in background

fg (program)	run the program in foreground
time (command/program)	print the command/program run time
su -	access as root
sudo (command)	execute a command as root
ping (host)	send ICMP ECHO_REQUEST packets to a network host printing the time taken by each packet to be received
traceroute (host)	send ICMP ECHO_REQUEST packets to a network host printing the response time of each gateway.
reboot	restart your machine
shutdown -h now	swith off your machine now
shutdown -r now	restart your machine now

FILE COMPRESSION/DECOMPRESSION

Compression:

tar	⇒	tar –cf name.tar (directory_to_compress)
tar.gz	⇒	tar –czf name.tar.gz (directory_to_compress)
tar.bz2	⇒	tar –cjf name.tar .bz2 (directory_to_compress)
zip	⇒	zip –r name.zip (directory_to_compress)
rar	⇒	rar name.rar (directory_to_compress)

Decompression:

tar.gz	⇒	tar –zxvf name.tar.gz
tar.bz2	⇒	tar –xvjf name.tar.bz2
tgz	⇒	tar –xzvf name.tgz
gz	⇒	gunzip name.gz
tbz	⇒	tar –xjvf name.tbz
tar	⇒	tar –xvf name.tar
bz2	⇒	bunzip2 name.bz2
zip	⇒	unzip name.zip
rar	⇒	unrar name.rar

TEXT EDITOR: VI

Open files:

vi (filename)	Open file
vi –r (filename)	Open last version of file after crash
vi +n (filename)	Open file at line n
vi + (filename)	Open file at end of file
vi +\string (filename)	Open file at the first occurrence of the string

Insert Mode:

i, Ins	insert text before cursor
a	append text after cursor
ni	insert n times a string in a line (ESC when the string is written)
ni	insert n times a string in a column (return +ESC when the string is written)
o	add a new line after the current line
O	add a new line before the current line
r	overwrite one character
R, Ins+Ins	replace characters
C	rewrite the current line

Command mode:

ESC	enter the command mode from insert or visual modes
.	repeat last command
:w	save file
:q	exit if no changes made
:wq , :x	save and exit
:q!	exit discarding any changes
:n (filename)	open a new file
:wn (filename)	save and open a new file
:r (filename)	insert the file after the current line
J	join the current and the next line
nJ	join the current and the next n-1 lines
u, :u	undo last change
U	undo all change to line

Cursor navigation

h , ←	cursor left
j , ↓	cursor down
k , ↑	cursor up
l , →	cursor right
w	next word
b	start of word
e	end of word
(	back a sentence

)	forward a sentence
{	back a paragraph
}	forward a paragraph
0	beginning of line
$	end of line
1G	start of file
G, gg	end of file
nG, :n	n-th line of file
f (char)	forward to character
F (char)	back to character
H	top of screen
M	middle of screen
L	bottom of screen

Deleting text

x	delete character to right of cursor
X	delete character to left of cursor
d	delete the character under the cursor
D , d$	delete from the cursor to the end of line
d0	delete from the cursor to beginning of line
dd, :d	delete current line
ndw	delete the next n words
ndb	delete the previous n words
ndd	delete n lines starting with current
dG	delete from current line to the end of file
d1G	delete from current line to the start of file
dnG	delete from current to n-th line
:x,yd	delete lines from x to y

Searching in the text

/string	search forward for string
?string	search backwards for string
n	go to the next match
N	go to the previous match
:set ic	ignore case while searching
:set noic	case sensitive searching
:x,yg/string	searching string from line x to y

Replacing text

:s/pattern/string/n	replace pattern with string at the n-th occurence from current position
:s/pattern/string/g	replace pattern with string at all occurences in the file

Visual Mode

1- Mark the start of the text with "v" or CTRL-V. The character under the cursor will be used as the start.
2- Move to the end of the text using cursor navigation commands. The text from the start of the visual mode up to and including the character under the cursor is highlighted.

3- Type an operator command.

Visual line mode

v	enter visual line mode
y	yank (copy) selected characters
Y	yank (copy) the whole line
p	paste on the following line
P	paste on the previous line
d	delete selection
D	delete line
<	move line to the left of a tab
>	move line to the right of a tab
:s/pattern/string/g	substitute the pattern with the string in the line

Visual block mode

CTRL-v	enter visual block mode
Y	yank (copy) block
p	paste on the right of the cursor
P	paste on the left of the cursor
d	delete selected block
<	move block to the left of a tab
>	move block to the right of a tab
:s/pattern/string/g	substitute the pattern with the string in the selected block

BASH SCRIPTING

A shell is not only a command line interface (or interpreter) but it also supports a powerful programming language with which we can create shell programs (sometimes called shell scripts) to create useful software tools in order to save time and reduce efforts.
A shell program is simply a file containing a set of UNIX commands that are to be executed sequentially. The file needs to have execute permissions (user permissions = rwx) set on it so that it can be executed just by typing in the name of the script at the command prompt as **./(script name)**.
The simplest of shell script is a text file containing a list of UNIX commands that are carried out sequentially once the file holding the command is executed, but shell language also allows many features of a high-level programming language, such as variables for storing data, decision-making controls (the if and case statements), looping controls (the for,while and until loops), function calls for program modularity,...
The first line of the script indicates which interpreter should execute the list of commands, for example **#!/bin/bash** uses the bash interpreter. Other lines beginning with # are interpreted as comments

VARIABLES

A variable in bash can contain a number, a character, a string of characters and is not typed, unlike in Fortran or C. Then you have no need to declare a variable, just assigning a value to its reference will create it. The variable values are retrieved by putting a '$' at the beginning. You can also assign the output of a command to a variable using the syntax: var=$(command)

Example

```
#!/bin/bash
a=3
b=7
c=$(( $a+$b ))              # Mathematical operation valid only for integer numbers
echo "the sum is " $c
a=2.22                      # Variables are overwritten
b=6.11
c=$( echo $a+$b | bc –l )   # Mathematical operation always valid
echo "the sum is " $c
curr_dir=$(echo $PWD)
echo "The current directory is " $curr_dir
```

Read external variables

Case 1: variables are listed after the script name (up to 9 variables)

```
./scriptname  var1 var2 var3

#!/bin/bash
a=$1              # var1 is assigned to a
b=$2              # var2 is assigned to b
c=$3              # var3 is assigned to c
n=$#              # n is the number of variables
echo $0           # write the script name
echo $a $b $c $n  # ⇒  var1 var2 var3 3
```

echo $@   *# write all variables ⇒ var1 var2 var3*

Case 2: use read command

./scriptname

#!/bin/bash
echo "insert the variables"
read var   *# read (variable name)*
echo "insert two numbers"
read n1 n2
echo $var $n1 $n2

IF/ELIF/ELSE: verificate conditions

Syntax (elif and else options are not mandatory):

if [condition1]; then
  (operations if the condition1 is verified)
elif [condition2]; then
  (operations if the condition2 is verified)
else
  (operations for other cases)
fi

Concatenation of conditions

[condition1] && [condition2]	True if condition1 AND condition2 are true
[condition1] || [condition2]	True if condition1 OR condition2 are true

Conditions for files

[-d FILE]	True if FILE exists and is a directory.
[-e FILE]	True if FILE exists.
[-f FILE]	True if FILE exists and is a regular file.
[-r FILE]	True if FILE exists and is readable.
[-w FILE]	True if FILE exists and is writable.
[-x FILE]	True if FILE exists and is executable.
[-s FILE]	True if FILE exists and has a size greater than zero.

A ! between the first parenthesis and the option lead to the opposite condition ([! –e FILE] is true if the the file doesn't exist)

Conditions for strings

[-z STRING]	True of the length of "STRING" is zero.
[-n STRING]	True if the length of "STRING" is non-zero.
[STRING1 == STRING2]	True if the strings are equal.
[STRING1 != STRING2]	True if the strings are not equal.

Conditions for integer numbers

[ARG1 -eq ARG2]	True if ARG1 = ARG2
[ARG1 -ne ARG2]	True if ARG1 ≠ ARG2
[ARG1 -gt ARG2]	True if ARG1 > ARG2
[ARG1 -lt ARG2]	True if ARG1 < ARG2
[ARG1 -ge ARG2]	True if ARG1 ≥ ARG2
[ARG1 -le ARG2]	True if ARG1 ≤ ARG2

Examples:

```
#Example 1: Find the greater between two real numbers

# First solution:

echo "Insert two real numbers:"
read a b
ans=$(echo "$a < $b" | bc –l)  # the output will be 1 if the expression is true or 0 if it is
                               # false
echo "Ans: " $ans
if [ $ans -eq 1 ]; then
    echo "$a < $b"
else
    echo "$a > $b"
fi

# Second solution:

int1=${a%%.*}   # Write the part of number before the dot
int2=${b%%.*}
dec1=${a##*.}   # Write the part of number after the dot
dec2=${b##*.}
echo "Integer parts: " ${a%%.*} ${b%%.*}
echo "Decimal parts: " ${a##*.} ${b##*.}
if [ $int1 -gt $int2 ] || [ $int1 -eq $int2 ] && [ $dec1 -gt $dec2 ]; then
    echo "$a > $b"
else
    echo "$a < $b"
fi

Example 2: Check if a file exists and if it is empty:

file="filename"
if [ -e $file ] ; then
    lines=$(wc $file | cut –c1-10)
    if [ $lines –eq 0 ]; then
        echo "$file is empty"
    else
        echo "$file has $lines lines"
    fi
else
```

echo "$file not found"
fi

FOR LOOP: iterate over elements

Syntax:

```
for i in (list of element); do
      (operations)
done

for ((i=(n1); i<=(n2); i++)); do
       (operations)
done
```

Examples

```
for i in 1 2 3 4 5; do
     echo $i
done

for i in $(cat filename); do
    echo $i
done

for ((i=1; i<=9; i++)); do
    echo $i
done
```

WHILE LOOP: iterate while a condition is verified

Syntax

```
while [ (condition) ] ; do
     (operations)
done
```

Examples

```
i=0                            # initialization of the variable
while [ $i -lt 10 ]; do
     echo $i
     i=$(($i+1))               # variable increment
done

cat filename | while read line ; do    # line is the variable
     echo $line
done

cat filename | while read line ; do
     if [ -z $line ]; then
```

```
        break                        # interrupt the loop if an empty line is found
    else
        echo $line
    fi
done
```

CASE STATEMENT: choose among options

Syntax:

```
case (variable) in
    val1)  (operations) ;;
    val2)  (operations) ;;
    ...
esac
```

Examples:

```
# find file type from its extension
for filename in $(ls); do
     ext=${filename##*.}      # write the part of string/number after the last full stop.
     case "$ext" in
          xyz) echo "$filename : geometry file";;
          out) echo "$filename : output file" ;;
          fdf) echo "$filename : input script" ;;
          txt) echo "$filename : text file" ;;
          *) echo " $filename : unknown type" ;;    #   *) means: "in all other cases"
      esac
done

echo "Insert an integer number"
read num
case $num in
      [0-9]) echo "number lower than 10" ;;
      [1-9]?) echo "number between 10 and 99" ;;
      100) echo "100!" ;;
      *) echo "number greater than 100" ;;
esac

echo "Do you agree with this? [yes or no]: "
read yno
case $yno in
     [yY] | [yY][Ee][Ss] )  echo "Agreed" ;;  # This is not case sensitive
     [nN] | [n|N][O|o] ) echo "Not agreed"
                         exit ;;                # Exit from the program
     *) echo "Invalid input"  ;;
esac
```

DEBUG SHELL SCRIPTS

When things in your script don't go according to plan, you need to determine what exactly causes the script to fail. Bash provides extensive debugging features. The most common is to start up the subshell with the -x option (**#!/bin/bash -x**), which will run the entire script in debug mode. Traces of each command plus its arguments are printed to standard output (after that the commands are expanded but before that they are executed). Using the option **–xv** also shell input lines are printed as they are read. It is also possible to debug only a part of the script. Indeed the commands **set –x** (or **set -xv**) and **set +x** (or **set +xv**) allow the debug to be turned on and off.

#!/bin/bash
(commands)
set –x
(commands to debug)
set +x
(commands)

MORE BASH SCRIPT EXAMPLES

PROGRAM 1: Demonstrates the use of comments, user-defined variables and echo.

```
1. #!/bin/bash
2. #Filename: bashdemo1 Author: M.T.Stanhope
3. #Define variables
4. name=John
5. car="Ford Escort"
6. age=21
7. #Display the contents of the variables
8. echo "My name is $name"
9. echo "I am $age years old and I drive a $car."
```

Program output:
My name is John
I am 21 years old and I drive a Ford Escort.

Explanation:
- Any line beginning with a hash sign # is a program comment (except the first line which is a special line identifying the shell interpreter to be used).
- Line 1 identifies the type of shell interpreter being used.
- Lines 4, 5, 6 are variable definitions. Note that there are no spaces on either side of the equals sign and how "Ford Escort" has double quotes around it as it contains a space character.
- Lines 8, 9 demonstrate the use of the echo command used to output text to the screen. Note how the contents of the variables are displayed by putting a dollar sign in front of the variable name.

PROGRAM 2: Demonstrates how the output of Unix commands can be stored in user-defined variables

```
#!/bin/bash
#Filename: bashdemo2 Author: M.T. Stanhope
#Define variables
```

```
4. todaysdate=$(date)
5. myworkingdirectory=$(pwd)
6. #Display the contents of the variables
7. echo "The date is $todaysdate"
8. echo "My present working directory is $myworkingdirectory"
9. echo "This machine is named $(hostname)"
```

Program output:
The date is Fri Aug 17 14:30:58 BST 2001
My present working directory is /home/martin/shelldemos
This machine is named homepc

Explanation:
- Lines 4 - The output of the Unix command date is assigned to the variable named todaysdate.
- Lines 7, 8 - The content of the variable is displayed by putting a dollar sign in front of the variable name.
- Line 9 - This shows how the output of the UNIX command 'hostname' can be directly placed in the string being output using echo. The technique of putting the output of a UNIX command into a variable or into the middle of an echo statement is referred to as '**command substitution**'.

PROGRAM 3: Demonstrating how the read command is used to get input from the user via the keyboard.

```
1. #!/bin/bash
2. #Filename: bashdemo3 Author: M.T. Stanhope
3. echo "Please enter your first name"
4. read firstname
5. echo "Please enter your surname"
6. read surname
7. echo "Please enter your date of birth: (dd/mm/yyyy)"
8. read birth
9. echo "Welcome $firstname $surname, your date of birth is on record  as $birth."
```

Program output (> means user input)**:**
Please enter your first name:
> Joe
Please enter your surname:
> Bloggs
Please enter your date of birth (dd/mm/yyyy):
> 01/04/1980
Welcome Joe Bloggs, your date of birth is on record as 01/04/1980

Explanation
Lines 4,6,8 - Note how the variables are defined and assigned values when they are first used by the read command.

PROGRAM 4: Demonstrating how user input can be obtained from the command line as command line arguments.

```
1. #!/bin/bash
2. #Filename: bashdemo4 Author: M.T. Stanhope
```

```
echo "This program has obtained its input from the command line"
echo "Welcome $1 $2, your date of birth is on record as $3."
echo "The total number of command line arguments = $#"
echo "The command line arguments supplied by the user are: $@"
```

NOTE: This program is executed by entering the command:
./bashdemo4 Joe Bloggs 01/04/1980

Program output:

Welcome Joe Bloggs, your date of birth is on record as 01/04/1980.
The total number of command line arguments = 3
The command line arguments supplied by the user are: Joe Bloggs 01/04/1980

Explanation:
This program is executed by typing in the shell program name followed by 3 pieces of information, referred to as program arguments. The positions of the words on the command line are identified by the following special variables (here is the full list):

$0 The script name
$1 The first argument (Joe in this example)
$2 The second argument (Bloggs in this example)
$3 The third argument (01/04/1980 in this example) ...
$# The number of arguments
$@ A space separated list of all the arguments.

PROGRAM 5: Demonstrating how decisions are made using the if...then statement (testing and branching).

```
#!/bin/bash
#Filename: bashdemo5 Author: M.T. Stanhope
clear
echo "Would you like to see a joke (y/n)?"
read reply
if [ "$reply" = "y" ]
        then
        echo "Question: How many surrealists does it take to change a light bulb?"
        echo "Answer: Fish."
fi
echo
echo "Have a nice day."
```

Program output (examples of both yes and no user responses are shown):
Would you like to see a joke (y/n)?
> y
Question: How many surrealists does it take to change a light bulb?
Answer: Fish
Have a nice day.

Would you like to see a joke (y/n)?
> n
Have a nice day

Explanation:

- Line 3 - The clear command clears the screen.
- Line 5 - Reads the user's input into a variable named reply.
- Line 6 - The start of the if statement. Notice the test is in square brackets with a space either side of them and that there is a space either side of the equals sign.
- Line 7 - The then part of the if statement identifys the part of the if statement that gets executed if the test is true.
- Line 8,9 - The body of the of the if statement. See how the lines have been indented for clarification of what belongs to the then part of the if statement.
- Line 10 - fi is if spelt backwards. It identifies the end of the if statement.
- Line 11 - An echo on its own will generate a blank line. An alternative way of writing the if...then statement saves a line of code as the word 'then' is placed on the same line as the 'if'. Note the use of the semicolon after the test condition and before the word 'then'...

```
if [ "$reply" = "y" ] ; then
    echo "Question: How many surrealists does it take to change a light bulb?"
    echo "Answer: Fish."
fi
```

PROGRAM 6: Demonstrating how decisions are made using the if...then...else statement (testing and branching).

```
1.  #!/bin/bash
2.  #Filename: bashdemo6 Author: M.T. Stanhope
3.  clear
4.  echo "Would you like to see a joke (y/n)?"
5.  read reply
6.  if [ "$reply" = "y" ] ; then
7.        echo "Question: How many surrealists does it take to change a light bulb?"
8.        echo "Answer: Fish."
9.  else
10.       echo "Not in the mood for jokes? Never mind perhaps another day."
11. fi
12. echo
13. echo "Have a nice day."
```

Program output (examples of both yes and no user responses are shown):

Would you like to see a joke (y/n)?
> y
Question: How many surrealists does it take to change a light bulb?
Answer: Fish.
Have a nice day.

Would you like to see a joke (y/n)?
> n
Not in the mood for jokes? Never mind perhaps another day.
Have a nice day

Explanation:
The program is very similar to the previous except for:

- Line 9 - The keyword else identifies the beginning of the part of the if statement that gets executed if the test is false.
- Line 10 - This line is displayed only if the test is false (i.e. if the user types in any letter other than y)

PROGRAM 7: Demonstrating how decisions are made using nested if...then...else statements (testing and branching).

```
1. #!/bin/bash
2. #Filename: bashdemo7 Author: M.T. Stanhope
3. echo "UNIX COMMAND SELECTOR"
4. echo "1. Show date"
5. echo "2. Show hostname"
6. echo "3. Show this month's calendar"
7. echo "Please make your selection (1,2,3)"
8. read menunumber
9. if [ $menunumber -eq 1 ]
10.     then
11.     date
12. elif [ $menunumber -eq 2 ]
13.     then
14.     hostname
15. elif [ $menunumber -eq 3 ]
16.     then
17.     cal
18. else
19.     echo "INVALID CHOICE!"
20. fi
21. echo
22. echo "Thank you for using the Unix command selector."
```

Program output (The example is for the user input of 1):

UNIX COMMAND SELECTOR
1. Show date
2. Show hostname
3. Show this month's calendar
Please make your selection (1,2,3)
> 1
Tues Oct 23 17:32:45 BST 2001
Thank you for using the Unix command selector.

Explanation:

- Lines 3-7 - Display a title, menu and prompt the user to select the number of a menu option.
- Line 8 - The user's response is read into a user defined variable named menunumber.
- Lines 9-22 - A series of nested if...then...else statements each performing a test for one of the possible menu option numbers. The three possible Unix commands are: 1. date, 2. hostname, 3. cal. Notice how -eq is used to test for equality between two numbers.
- Line 23 - Displays a final ending message.

PROGRAM 8: Demonstrating how decisions are made using the case...esac statement (testing and branching).

```
1. #!/bin/bash
2. #Filename: bashdemo9 Author: M.T. Stanhope
3. echo "UNIX COMMAND SELECTOR"
4. echo "1. Show date"
5. echo "2. Show hostname"
6. echo "3. Show this month's calendar"
7. echo "Please make your selection (1,2,3)"
8. read menunumber
9. case $menunumber in
10.      1) date;;
11.      2) hostname;;
12.      3) cal;;
13.      *) echo "INVALID CHOICE!";;
14. esac
15. echo
16. echo "Thank you for using the Unix command selector."
```

Program output (The example is for the user input of 1):

```
UNIX COMMAND SELECTOR
1. Show date
2. Show hostname
3. Show this month's calendar
Please make your selection (1,2,3)
> 1
Tues Oct 23 17:32:45 BST 2001
Thank you for using the Unix command selector.
```

Explanation:

- Line 9 - The first line of the case...esac statement checks to see the value held in the variable named menunumber.
- Lines 10-13 give possible branch conditions depending upon the value held in the variable menunumber. The content of the variable is checked against the value 1, 2 or 3 or anything else (represented by the asterisk). Note the double semicolon at the end of each test condition.
- Line 14 - The word esac is case spelt backwards. It identifies the end of the case statement.

PROGRAM 9: Demonstrating how looping is achieved using the for statement.

```
#!/bin/bash
#Filename: bashdemo10 Author: M.T. Stanhope
echo "Demonstration of looping using the for loop and a list of car names"
for car in  ford vauxhall rover toyota mazda subaru
     do
     echo $car
done
echo
echo "End of demonstration program."
```

Program output:

```
Demonstration of looping using the for loop.
ford
```

vauxhall
rover
toyota
mazda
subaru
End of demonstration program.

Explanation:
Line 4-7 - The general format of a for loop is:

for *variable* in *list_of_items*
    do
    commandA
    commandB
done

The keywords are: for, in, do and done. In this example car is a user-defined variable and the list of data items are: ford vauxhall rover toyota mazda subaru all separated with spaces. The example only has one command belonging to the for loop which resides between the keywords do and done. The content of the variable named car has a different value for each pass through the loop.

PROGRAM 10: Demonstrating how looping is achieved using the for statement.

```
1. #!/bin/bash
2. #Filename: bashdemo11 Author: M.T. Stanhope
3. echo "Demonstration of looping using the for loop and a list of filenames generated by the ls
   command"
4. for myfile in $(ls)
5.     do
6.     cat $myfile
7. done
```

Program output:
The output seen on the screen would be the contents of all the text files held in the current directory. You will first have to create some if none already exist in your directory.

Explanation:
Line 4 - The loop variable is named myfile. The list of data items is generated by the Unix command ls. Each pass through the loop then displays the contents of a file by using the Unix cat command.

PROGRAM 11: Demonstrating how looping is achieved using the for statement.

```
1. #!/bin/bash
2. #Filename: bashdemo12 Author: M.T. Stanhope
3. echo "Demonstration of looping using the for loop and a list of filenames held in a text file named
   myfilelist"
4. for myfile in $(cat myfilelist)
5.     do
6.     cat $myfile
7. done
```

Program output:

The output seen on the screen would be the contents of all the text files that have their names listed in the text file named myfilelist.

Explanation:
Line 4 - The loop variable is named myfile. The list of data items is held in a text file named myfilelist. The list of items is generated by the Unix command cat myfilelist. Each pass through the loop then displays the contents of a file that has its name listed in the text file named myfilelist.

PROGRAM 12: Demonstrating how looping is achieved using the for statement.

```
1. #!/bin/bash
2. #Filename: bashdemo13 Author: M.T. Stanhope
3. echo "Demonstration of looping using the for loop and a list of arguments supplied at the
   command line"
4. echo "Program invoked by the command: ./bashdemo13 filename1 filename2 filename3"
5. for myfile in $@
6.        do
7.       cat $myfile
8. done
```

Program output:
The output seen on the screen would be the contents of all the text files listed on the command line when the shell program is executed.

Explanation:
Line 5 - The loop variable is named myfile. The list of data items is represented by the special variable name $@

PROGRAM 13: Demonstrating how looping is achieved using the while statement.

```
#!/bin/bash
#Filename: bashdemo14 Author: M.T.Stanhope
quit=n
while [ "$quit" = "n" ]
        do
        clear
        echo "1. Show Date"
        echo "2. Show Host Name"
        echo "Q. Quit"
        echo "Enter choice"
        read choice
        case $choice in
                1) date
                   echo "Write n to continue"
                   read quit;;
                2) hostname
                   echo "Write n to continue"
                   read quit;;
                Q|q) quit=y;;
                *) echo "Invalid choice!"
                   sleep 1;;
```

```
22. esac
23. done
24. echo "PROGRAM FINISHED"
```

Program output:

1. Show Date
2. Show Host Name
Q. Quit
Enter choice:
> 1
Mon Sep 1 11:40:14 BST 2003
Write n to continue...

Explanation:

Line 4 - This is the beginning of the while loop. It consists of the word 'while' followed by some test. If the test is true, the code between the do and the done lines is executed. If the test is false then the next line to be executed is the one after the done line. In the above example, the only way the test condition is made false is by the user picking the Q or q option which assigns 'y' to the variable named quit.

PROGRAM 14: Useful menu driven shell program example.

```
#!/bin/bash
#Filename: bashdemo16 Author: M.T.Stanhope
quit=n
while [ "$quit" = "n" ]
        do
        clear
        echo
        echo "1. Show Date"
       echo "2. Show Host Name"
       echo "3. Show Calendar"
       echo "4. Display Text File"
      echo "Q.Quit"
      echo
      echo "Enter choice"
      read choice
      case $choice in
            1) date
               echo "Write n to continue"
               read quit;;
            2) hostname
               echo "Write n to continue"
               read quit;;
            3) cal
               echo "Write n to continue"
               read quit;;
            4) echo "Enter name file to be displayed"
               read myfilename
               if [ -e "$myfilename" ] && [ -r "$myfilename" ]
                     then
                     clear
                     cat $myfilename
```

```
32.             else
33.                  echo "Cannot display $filename"
34.             fi
35.             echo "Write n to continue"
36.             read quit;;
37.           Q|q) quit=y;;
38.           *) echo "Invalid choice!"
39.              sleep 1;;
40. esac
41. done
42. echo "PROGRAM FINISHED"
```

Program output:

1. Show Date
2. Show Host Name
3. Show Calendar
4. Display Text File
Q. Quit
Enter choice:
> 1
Mon Sep 1 11:50:30 BST 2003
Write n to continue...

Explanation:

Most of the programming structures used in this example have been covered earlier (while...do...done, case...in...esac, if...then...else...fi). The only thing that is new is the test that is performed in checking if the file name entered is that of a file and that the file is readable (see line 28).

PROGRAM 15: Check if a file is a script or not.

```
1. #!/bin/bash
2. for i in $(ls); do
3. line1=$(head -1 $i | awk '{print $1}')
4. if [ $line1 == '#!/bin/bash' ]; then
5.      echo "$i is a shell script"
6. fi
7. done
```

Program output

The output of the program should be the list of your script files

Explanation

- Line 2 - the command head -1 prints the first line of each file, awk selects the first field of the line.
- Line 3 - If the string extracted from the file is '#!/bin/bash' then the file is a script

STRINGS & ARRAYS

STRINGS

Length of a string

${#var} number of character of a string

var = 'this_is_a_string'
echo ${#var} ⇒ 16

Extract substrings

${var:X:Y} extract Y characters, starting from the (X+1)-th, from the variable var

var = 'this_is_a_string'
echo ${var:5:8} ⇒ is_a_str
*echo ${var:3:2} ⇒ s_*

Operators #, ##, %, %%

${var#(condition)}	delete the shortest possible match from the left satisfying the condition
${var##(condition)}	delete the longest possible match from the left satisfying the condition
${var%(condition)}	delete the shortest possible match from the right satisfying the condition
${var%%(condition)}	delete the longest possible match from the right satisfying the condition

var='this_is_a_string'
*For # and ## let's consider the condition *s and let's look for all substrings that satisfy this condition:*

t	*this_i*	*this_is_a_s←*	*this_is_a_string*
th	*this_is←*	*this_is_a_st*	
thi	*this_is_*	*this_is_a_str*	
this←	*this_is_a*	*this_is_a_stri*	
*this_*	*this_is_a_*	*this_is_a_strin*	

*echo ${var#*s} delete this ⇒ _is_a_string*
*echo ${var##*s} delete this_is_a_s ⇒ tring*

For % and %% you have to consider the substrings from the right (g, ng, ing, ring, tring, string,...)
echo ${var%s} delete string ⇒ this_is_a_*
echo ${var%%s} delete s_is_a_string ⇒ thi*

Delete and subtitute substring

${var/substring/}	# Delete the first occurrence of substring
${var//substring/}	# Delete all occurrence of substring
${var/substring1/substring2}	# Substitute the first substring with the second at the first occurrence
${var//substring1/substring2}	# Substitute the first substring with the second for all occurrences

ARRAYS

If you're used to a "standard" UNIX shell you may not be familiar with bash's array feature. Although not as powerful as similar constructs in the P languages (Perl, Python, and PHP) and others, they are often quite useful. Bash arrays have numbered indexes only, but they are sparse, ie you don't have to define the elements for all the indexes. Undefined elements are set to zero. An entire array can be assigned by enclosing the array items in parenthesis:

arr=(item1 item2 item3 ...)

Individual elements can be assigned with the familiar array syntax:

arr[0]=item1
arr[1]=item2

Notice that, by default, array indexes start from zero. Once defined the array, the following constructs are available (notice that the "@" sign can be used instead of the "*" in constructs such as ${arr[*]}):

${arr[n]}	# The element with index n
${arr[*]}	# All of the elements in the array
${!arr[*]}	# All of the indexes in the array
${#arr[*]}	# Number of elements in the array
${#arr[n]}	# Length of the element with index n
${arr[*]:X}	# Extract the elements from ${arr[X]} to the end of the array
${arr[*]:X:Y}	# Extract Y elements from the array starting from ${arr[X]}

The following example shows some simple array usage (note the "[index]=value" assignment to assign a specific index):

```
#!/bin/bash

array=(one two three four [5]=five)
echo "Array size: ${#array[*]}"

echo "Array items:"
for item in ${array[*]}
do
	printf "   %s\n" $item
done

echo "Array indexes:"
for index in ${!array[*]}
do
	printf "   %d\n" $index
done

echo "Array items and indexes:"
for index in ${!array[*]}
do
	printf "%4d: %s  \n" $index ${array[$index]}
done
```

Arrays of strings

- upper & lowercase

${array[@],} # The first character of each element will be in lowercase
${array[@],,} # All elements will be in lowercase
${array[@]^} # The first character of each element will be in uppercase
${array[@]^^} # All elements will be in uppercase

- delete and substitute substrings

${array[@]/substring/} # Delete the first occurrence of substring for all elements
${array[@]//substring/} # Delete all occurrence of substring for all elements
${array[@]/substring1/substring2} # Substitute the first substring with the second at the first occurrence for all elements.
${array[@]//substring1/substring2} # Substitute the first substring with the second for all occurrences for all elements.

SED & AWK

SED: sed [options] (path)

Delete lines:

sed '(n)d' filename	delete the (n)-th line of the file
sed '$d' filename	delete the last line from the file
sed '(n1),(n2)d' filename	delete the lines from (n1) to (n2)
sed '/^$/d' filename	delete empty lines
sed '/(string1)/d' filename	delete the lines containing (string1)
sed '/(string1)/,/(string2)/d' filename	delete the lines from that containing (string 1) to that containing (string2)

Replacement of d with !d generates the opposite effect (for example *sed '3!d' filename* deletes all lines except the third)

Insert lines (add a new line):

sed '(n)i (string)' filename	insert the (string) in the line (n)
sed '/(string1)/i (string2)' filename	insert the (string2) in the line containing (string1)

Substute strings:

sed -e '/(string1)/s//(string2)/' filename	substitute the (string2) to the (string1) at the first occurence in the file
sed -e '/(string1)/s//(string2)/g' filename	substitute the (string2) to the (string1) for all occurences in the file
sed -e '/(string1)/s//(string2)/(n)' filename	substitute the (string2) to the (string1) at the (n)th occurence in the file

Substitute lines:

sed '(n)c\ (string)' filename	substitute the (n)-th line with the (string)
sed '/(string1)/c\ (string2)' filename	substitute all lines containing (string1) with (string2)

Print lines:

sed -n -e '(n)p' filename	print the (n)-th line of the file
sed -n -e '$p' filename	print the last line of the file
sed -n -e '/(string)/p' filename	print the lines containing the (string)
sed -n -e '/(string1)/,/(string2)/p' filename	print the lines from that containing (string1) up to that containing (string2)

Replacement of p with !p generates the opposite effect (for example *sed '3!p' filename* prints all lines except the third)

Number lines:

sed = (filename)	number lines of the file
sed -n '/ (string) /=' (filename)	number lines containing the (string)

AWK: awk [options] '{ options }' (path)

Awk repeats a command over all lines. The lines of the file are called records, while the columns are called fields. Some variables are automatically defined when awk is called:

- **NR** record number, an index incremented by one for each line
- **FNR** file record number, the index of lines (default FNR==NR)
- **FS** input field separtor for columns (default = blank)
- **OFS** output field separator for columns (default = blank)
- **RS** input record separtor for lines (default = newline, \n)
- **ORS** output record separator for lines (default = newline, \n)
- **NF** number of fields (number of columns)

Some syntax rules:

- **$(n)** (n)th field
- **FNR==(n){ }** execute this block at line (n)
- **FNR>=(n){ }** execute this block from line (n) to end of file (also allowed FNR<=(n), FNR<(n), FNR>(n))
- **FNR>=(n)&&FNR<=(m){ }** execute this block from the (n)-th to the (m)-th line
- **FNR<=(n)||FNR>(m){ }** execute this block until the (n)-th and from the (m)-th line to the end of file
- **BEGIN{ }** execute first this block when awk is called.
- **END{ }** execute this block at the end of file
- **/string/{ }** execute the block for lines containing the string

Print statements: awk '{print field1, field2, ...}' filename

Examples

awk '{print $1, $NF}' filename	*print the first and the last fields of the file*
awk '{print $1 $NF}' filename	*print the first and the last field (colums) of the file without separator (comma means one blank)*
awk 'FNR==NR{FS=OFS=" "}{print $1 FS $NF}' filename	*as the first case, but with explicit default values*
awk '{print $1, "string", $2 }' filename	*print the string between the two fields*
awk '{print $0}' filename	*print all fields*
awk 'FNR<=3 {print $0}' filename	*print the first three line*
awk 'END{print $0}' filename	*print the last line*
awk 'FNR==1||FNR==3{print $0}' filename	*print the first and the third line of the file*
awk 'FNR>2&&FNR<=4{print $0}' filename	*print the third and the fourth line of the file*
awk 'END{OFS=":"}{print $1 OFS $NF}' filename	*print the first and last field of last line with : in between*
awk 'END{print NR}' filename	*print the number of lines*
awk '/string/{print $0}' filename	*print all lines containing the string*

Formatted print statements: awk '{printf [format], field1, field2, ...}'

For each field a format specifier has to be declared. A format specifier starts with the character '%' and ends with a format-control letter specifying the kind of value to print and determining how to output the item. The rest of the format specifier is made up of optional modifiers that can control how many characters of the item's value are printed, as well as how much space it gets. The modifiers come between the '%' and the format-control letter. The format should finish with the newline symbol \n, otherwise the output will be written side by side.

Format control letters:

"%c"	ASCII character
"%d" or "%i"	integer number
"%e"	scientific (exponential) notation number
"%f"	real number
"%s"	string
"%g"	exponential or real, whichever uses fewer characters

Specifiers

-	left-adjusted output (default right-adjusted)
x.y	x is the minimum number of characters used to print the integer part of a real number, y is the number of characters used for the decimal part of a real or exponential number
x	x is the minimum number of character used to print an integer
.x	x is the maximum number of character used to print a string

!!!! commas after the format and between the fields are important as well as the blanks in the format statements.

Examples:

consider the file containing: *23.55567 2700*

awk '{printf "%f %d\n", $1, $2}' file ⇒ *23.555670 2700*
awk '{printf "%2.8f %5d\n", $1, $2}' file ⇒ *23.55567000 2700*
awk '{printf "%1.1f %10d\n", $1, $2}' file ⇒ *23.6   2700*
awk '{printf "%1.1f %-10d\n", $1, $2}' file ⇒ *23.6 2700*
awk '{printf "%2.2e %f\n", $1, $2}' file ⇒ *2.36e+01 2700.000000*
awk '{printf "%3g %2s\n", $1, $2}' file ⇒ *23.5557 2700*
awk '{printf "%.3g %.2s\n", $1, $2}' file ⇒ *23.6 27*
awk '{printf "%.1g %.6s\n", $1, $2}' file ⇒ *2e+01 2700*
awk '{printf "%.1g string %.6s\n", $1, $2}' file ⇒ *2e+01 string 2700*

Variables and operations

Variables used in the awk command must be defined in a BEGIN block, and if they are not initialized, their values are set to zero
Variables external to the awk program must be imported as
awk -v var_int1=$varext1 -v var_int2=$varext2 '{ (options) }' filename
Variables in awk are not identified by $, since it indicates the fiels.
The results of aritmethical operation among field values, variables and constants can be printed directly in the output using print/printf

Examples

consider the file containing: *23.55567 2700*

awk '{printf "%f %f %f\n", $1, $2, $1+$2}' file
⇒ *23.555670 2700.000000 2723.555670*
awk '{printf "%d %d %f\n", $1, $2, $1*$2}' file ⇒ *23 2700 63600.309000*

awk '{printf "%f %f %f\n", $1+100, $2/10}' file ⇒ *123.555670 270.000000*
awk 'BEGIN{a=20; b=70}{print $1+a, $2-b, a-b}' file ⇒ *43.5557 2630 -50*
a=2; awk -v b=$a '{print $2+b}' file ⇒ *2702*

ADVANCED AWK EXAMPLES: IF-ELSE / FOR

cat file	cat file2
2.2 3.4 4.4	-3.4 1.4 1.2
2.3 5.5 -1.1	3.0 6.4 9.0
2.4 2.2 -3.3	0.4 4.4 7.0
2.5 6.0 0.3	2.4 5.4 0.1

1- For each line print the higher between the first and the second field:

```
# If-else statement inside awk:
awk 'FNR==NR{ if ($1 > $2)
   {print $1}
    else
    {print $2}
    }' file

# As before but with else if statement.
awk 'FNR==NR{ if ($1 > $2)
    {print $1}
    else if ($1 < $2)
    {print $2}
    else
    {print "equal"}
    }' file

# As the first but with format specified in a BEGIN block:
awk 'BEGIN {
      FORMAT="%s higher than %s\n"
      FORMAT1="%s\n"
      {printf FORMAT1,"Filename = file"}
      {printf "\n"}
      }
      FNR==NR{ if ($1>$2)
      {printf  FORMAT, $1, $2}
      else
      {printf FORMAT, $2, $1}
      }' file

#The most compact (and criptic) solution:
awk '{print ($1>$2) ? $1 : $2}' file
         # this means: if the condition ($1>=$2) is verified print $1, otherwise (:) print $2. Of course
        # instead of $2 you can specify another condition, for example:
awk '{print ($1>$2)&&($1>$3) ? $1 : ($2>$3) ? $2 : $3}' file
         # find the maximum value for each line.
```

2- Find the maximum, the minimum and the average of the first column

awk '{sum+=$1}{x[NR]=$1}END{asort(x);print " max = "x[NR]"\n","min = "x[1]"\n", "average = "sum/NR "\n"}' file
 # The first block calculates the sum of the elements of the first column (+= means "add $1 to the # variable sum"). The second block puts the elements of the column into an array. At last the # array is sorted in ascending order and the max, min and average values are printed.

3- Find the averages over all lines

awk '{sum=0; for(i=1;i<=NF;i++)
sum+=$i}{printf "Sum = %f\n", sum/NF}' file
 # The variables sum must be set to zero before sum over a line, so it is not defined in a begin block
 # After for( ) you should start a new line

4- Sum the matrices in the two files

```
awk 'FNR==NR{for(i=1; i<=NF; i++)
      _[FNR,i]=$i ; next
      }{for(i=1; i<=NF; i++)
      printf("%s%s", $i+_[FNR,i], (i==NF) ? "\n" : FS)
      }' file file2
        # next allows to pass to the other file
        # _[FNR,i] indicates a matrix element
```

HILBERT & QUEUES

ssh hilbert connect to hilbert

cp (path) /nfshome/(user)/(path) copy files from hilbert to the home
cp /nfshome/(user)/(path) (path) copy files from the home to hilbert
! Copy commands run only on hilbert, not in your local home

qsub file.job submit a job
qsub –W depend=afterany:(id) file.job append a job to another identified by (id)

qstat show jobs of all users
qstat -u (user) show jobs of the user
qstat -n show informations about the nodes
qstat -n (id) show informations about the nodes for a specific job
qstat -q show informations about queues
qstat -f show detailed informations about jobs
qstat -f (id) show detailed informations for a specific job

qdel (id) delete a job