**Linux**

1. **Linux History**

🡪All modern operating systems have their roots in **1969** when **Dennis Ritchie** and **Ken**

**Thompson** developed the C language and the Unix operating system at AT&T Bell Labs.

🡪They shared their source code (yes, there was open source back in the Seventies) with the rest of the world, including the hippies in Berkeley California. By 1975, when AT&T started selling Unix commercially, about half of the source code was written by others. The hippies were not happy that a commercial company sold software that they had written; the resulting (legal) battle ended in there being **two** **versions** of Unix: the official AT&T Unix, and the free BSD Unix.

🡪Today more than 97 percent of the world's supercomputers (including the complete top 10), more than 80 percent of all smartphones, many millions of desktop computers, around 70 percent of all web servers, a large chunk of tablet computers, and several appliances (dvdplayers, washing machines, dsl modems, routers, self-driving cars, space station laptops...) run Linux. Linux is by far the most commonly used operating system in the world.

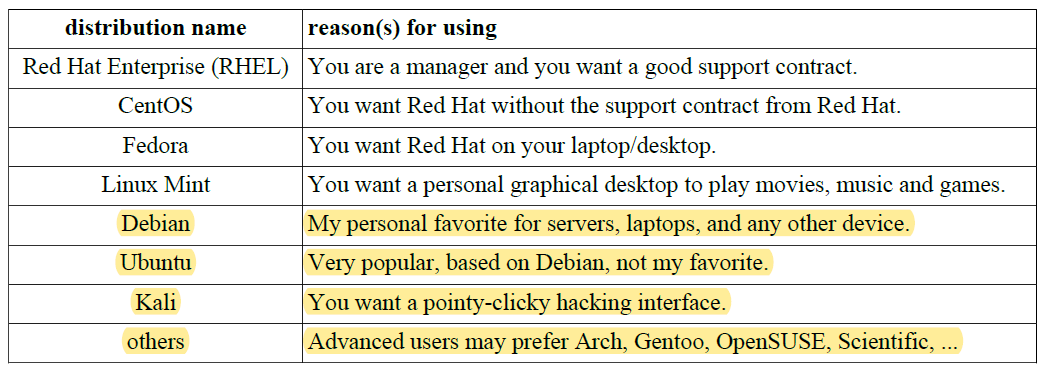
1. **Distributions**

🡪A Linux **distribution** is a collection of (usually open source) software on top of a Linux kernel. A distribution (or short, distro) can bundle server software, system management tools, documentation and many desktop applications in a **central secure software repository**. A distro aims to provide a common look and feel, secure and easy software management and often a specific operational purpose.

🡪Let's take a look at some popular distributions.

1. **Red Hat :** Red Hat is a billion dollar commercial Linux company that puts a lot of effort in developing Linux. They have hundreds of Linux specialists and are known for their excellent support. They give their products (Red Hat Enterprise Linux and Fedora) away for free. While **Red Hat Enterprise Linux (RHEL)** is well tested before release and supported for up to seven years after release, **Fedora** is a distro with faster updates but without support.
2. **Ubuntu** : Canonical started sending out free compact discs with Ubuntu Linux in 2004 and quickly became popular for home users (many switching from Microsoft Windows). Canonical wants Ubuntu to be an easy to use graphical Linux desktop without need to ever see a command line. Of course they also want to make a profit by selling support for Ubuntu.
3. **Debian** : There is no company behind Debian. Instead there are thousands of well organized developers that elect a Debian Project Leader every two years. Debian is seen as one of the most stable Linux distributions. It is also the basis of every release of Ubuntu. Debian comes in three versions: stable, testing and unstable. Every Debian release is named after a character in the movie Toy Story.
4. **Others** : Distributions like CentOS, Oracle Enterprise Linux and Scientific Linux are based on Red Hat Enterprise Linux and share many of the same principles, directories and system administration techniques. Linux Mint, Edubuntu and many other \*buntu named distributions are based on Ubuntu and thus share a lot with Debian. There are hundreds of other Linux distributions.
5. **Which To Choose?**

🡪Below are some very personal opinions on some of the most popular Linux Distributions. Keep in mind that any of the below Linux distributions can be a stable server and a nice graphical desktop client.



🡪When you are new to Linux in 2015, go for the latest Mint or Fedora. If you only want to practice the Linux command line then install one Debian server and/or one CentOS server (without graphical interface).

1. **Installation is not included in the notes, which is easily available in the internet**
2. **Man Pages**

🡪This chapter will explain the use of **man** pages (also called **manual pages**) on your Unix or Linux computer.

🡪You will learn the man command together with related commands like **whereis**, **whatis** and **mandb**.

🡪Most Unix files and commands have pretty good man pages to explain their use. Man pages also come in handy when you are using multiple flavours of Unix or several Linux distributions since options and parameters sometimes vary.

1. **man $command** : Type **man** followed by a command (for which you want help) and start reading. Press **q to quit** the manpage. Some man pages contain examples (near the end).



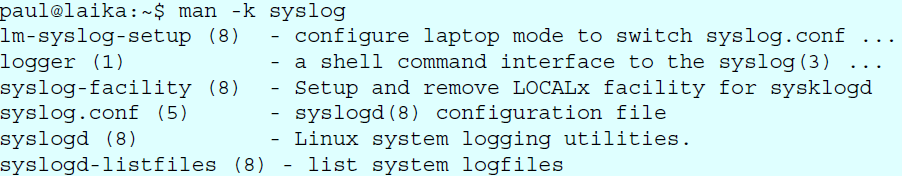
1. **man $configfile** : Most **configuration files** have their own manual.



1. **man $daemon** : This is also true for most **daemons** (background programs) on your system..



1. **man -k (apropos)** : **man -k** (or **apropos**) shows a list of man pages containing a string.



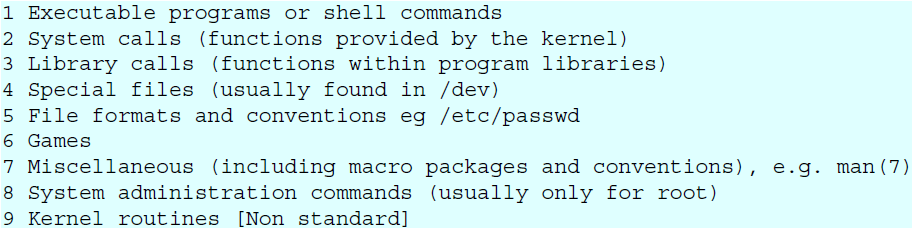
1. **whatis** : To see just the description of a manual page, use **whatis** followed by a string.



1. **whereis** : The location of a manpage can be revealed with **whereis**.



1. **man sections** : By now you will have noticed the numbers between the round brackets. **man man** will explain to you that these are section numbers. Executable programs and shell commands reside in section one.



1. **man $section $file** : Therefor, when referring to the man page of the passwd command, you will see it written as **passwd(1);** when referring to the **passwd file**, you will see it written as **passwd(5)**. The screenshot explains how to open the man page in the correct section.

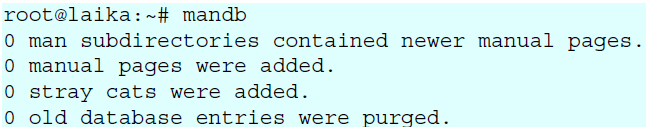


1. **man man** : If you want to know more about man, then Read **The Fantastic Manual (RTFM).**

🡪Unfortunately, manual pages do not have the answer to everything...



1. **mandb** : Should you be convinced that a man page exists, but you can't access it, then try running mandb on Debian/Mint.



1. **Working With Directories**

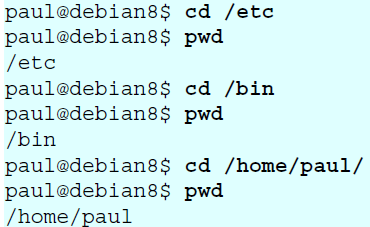
🡪This module is a brief overview of the most common commands to work with directories: **pwd, cd, ls, mkdir and rmdir**. These commands are available on any Linux (or Unix) system.

🡪This module also discusses **absolute** and **relative paths** and **path completion** in the bash shell.

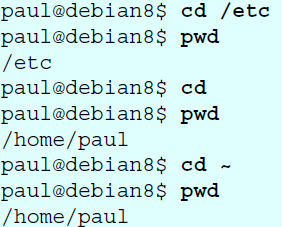
1. **pwd :** The **you are here** sign can be displayed with the **pwd** command (Print Working Directory). Go ahead, try it: Open a command line interface (also called a terminal, console or xterm) and type **pwd**. The tool displays your **current directory.**

****

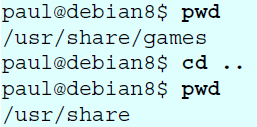
1. **cd :** You can change your current directory with the **cd** command (Change Directory).



1. **cd ~ : The cd is also a shortcut to get back into your home directory.** Just typing cd without a target directory, will put you in your home directory**. Typing cd ~ has the same effect.**

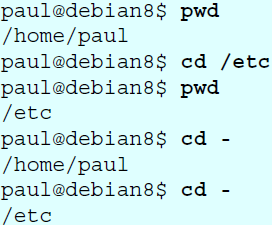
****

1. **cd .. :** To go to the parent directory (the one just above your current directory in the directory tree), type cd ..

****

🡪To stay in the current directory, type cd . ;-) We will see useful use of the . character representing the current directory later.

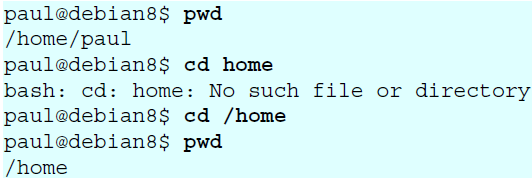
1. **cd - :** Another useful shortcut with **cd** is to just type **cd -** to go to the previous directory.

****

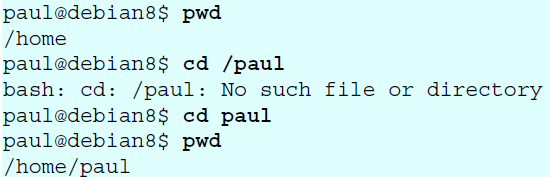
1. **absolute and relative paths**

🡪You should be aware of **absolute and relative paths** in the file tree. When you type a path starting with a **slash (/),** then the **root** of the file tree is assumed. If you don't start your path with a slash, then the current directory is the assumed starting point.

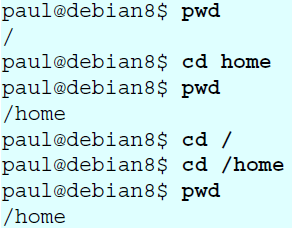
🡪The screenshot below first shows the current directory **/home/paul**. From within this directory, you have to type **cd /home** instead of cd home to go to the **/home** directory.



🡪When inside **/home**, you have to type **cd paul** instead of **cd /paul** to enter the subdirectory **paul** of the current directory **/home**.



🡪In case your current directory is the **root directory /**, then both **cd /home** and **cd home** will get you in the **/home** directory.



🡪This was the last screenshot with **pwd** statements. From now on, the current directory will often be displayed in the prompt. Later in this book we will explain how the shell variable **$PS1** can be configured to show this.

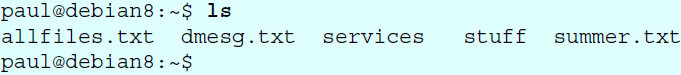
1. **Path Completion**

🡪**The tab key can help you in typing a path without errors**.

🡪Typing cd /et followed by the tab key will expand the command line to cd /etc/. When typing cd /Et followed by the tab key, nothing will happen because you typed the wrong path (upper case E).

🡪You will need fewer key strokes when using the **tab key**, and you will be sure your typed **path** is correct!

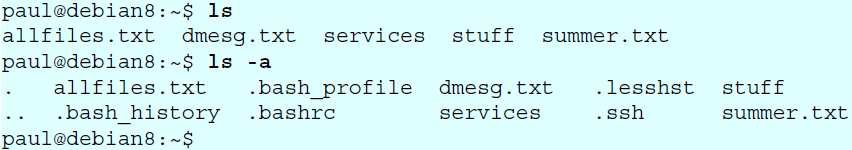
1. **ls :** You can list the contents of a directory with **ls**.



1. **ls -a :** A frequently used option with ls is **-a to show all files**.

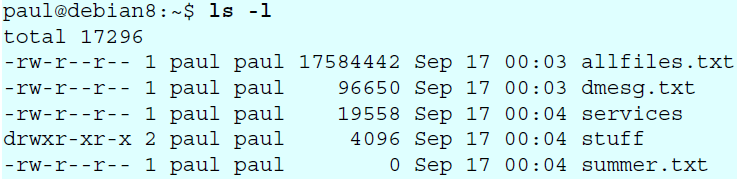
🡪**Showing all files means including the hidden files.**

🡪**When a file name on a Linux file system starts with a dot, it is considered a hidden file and it doesn't show up in regular file listings.**



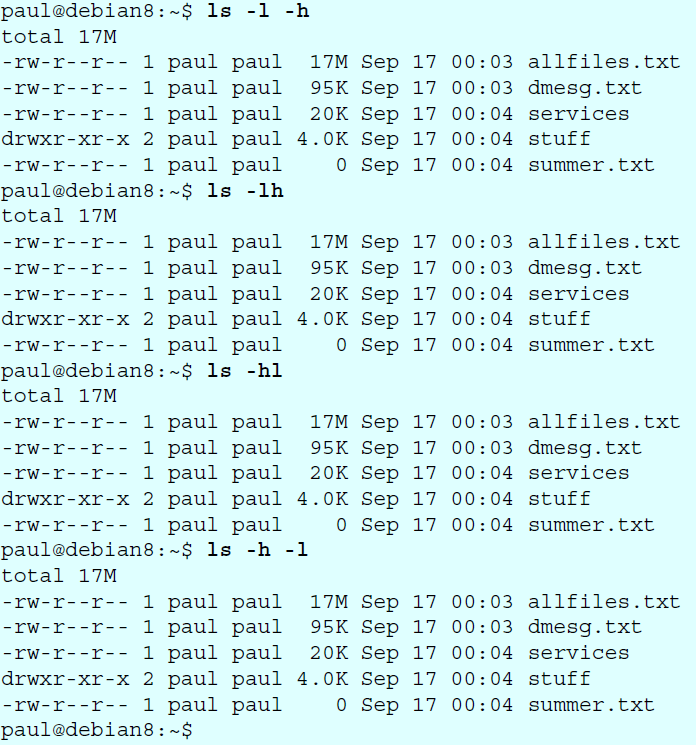
1. **ls -l :** Many times you will be using options with ls to display the contents of the directory in different formats or to display different parts of the directory.

🡪Typing just ls gives you a list of files in the directory. Typing ls -l (that is a letter L, not the number 1) gives you a long listing.

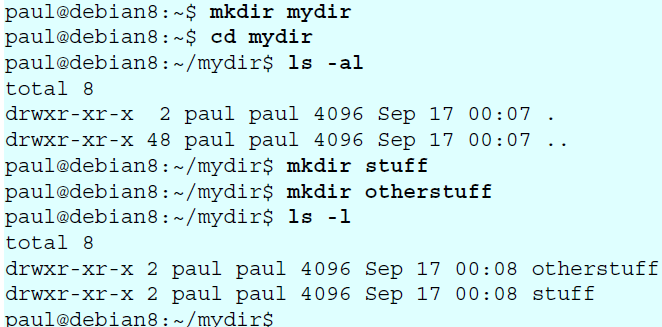


1. **ls -lh :** Another frequently used ls option is **-h**. It shows the numbers (file sizes) in a more human readable format. Also shown below is some variation in the way you can give the options to ls. We will explain the details of the output later in this book.

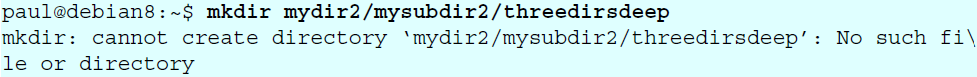
**🡪**Note that we use the letter L as an option in this screenshot, not the number 1.

****

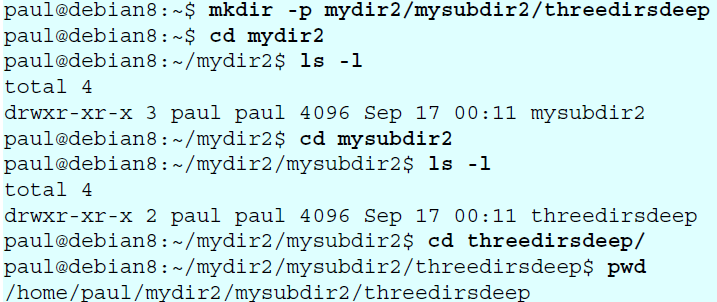
1. **mkdir :** Walking around the Unix file tree is fun, but it is even more fun to create your own directories with **mkdir**. You have to give at least one parameter to **mkdir**, the name of the new directory to be created. Think before you type a leading / .

****

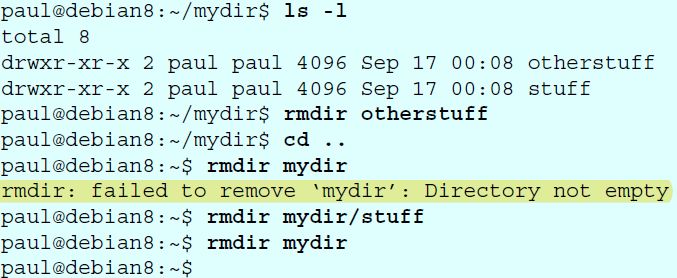
1. **mkdir -p :** The following command will fail, because the **parent directory** of threedirsdeep does not exist.

****

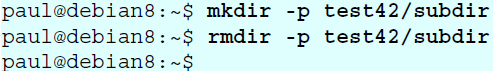
**🡪**When given the option **-p**, then **mkdir** will create **parent directories** as needed.

****

1. **rmdir :** When a directory is empty, you can use rmdir to remove the directory.

****

1. **rmdir -p :** And similar to the **mkdir -p** option, you can also use **rmdir** to recursively remove directories.

****

1. **Working With Files**

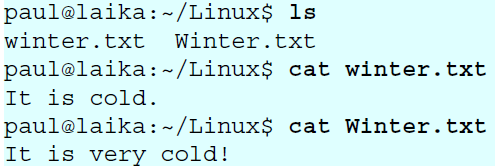
🡪In this chapter we learn how to recognise, create, remove, copy and move files using

commands like **file**, **touch**, **rm**, **cp**, **mv** and **rename**.

**🡺All files are case sensitive**

🡪Files on Linux (or any Unix) are **case sensitive**. This means that **FILE1** is different from **file1**, and **/etc/hosts** is different from **/etc/Hosts** (the latter one does not exist on a typical Linux computer).

🡪This screenshot shows the difference between two files, one with upper case **W**, the other with lower case **w**.

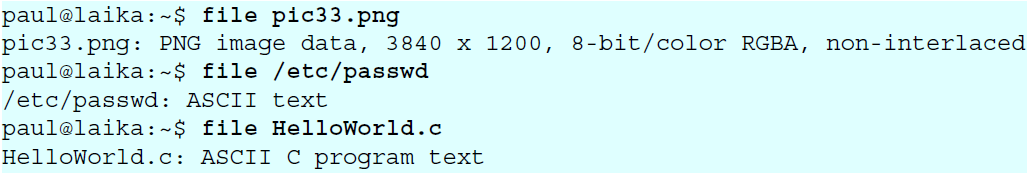


**🡺Everything is a file**

🡪A **directory** is a special kind of **file**, but it is still a (case sensitive!) **file**. Each terminal window (for example **/dev/pts/4**), any hard disk or partition (for example **/dev/sdb1**) and any process are all represented somewhere in the **file** **system** as a **file**. It will become clear throughout this course that everything on Linux is a file.

**🡺file**

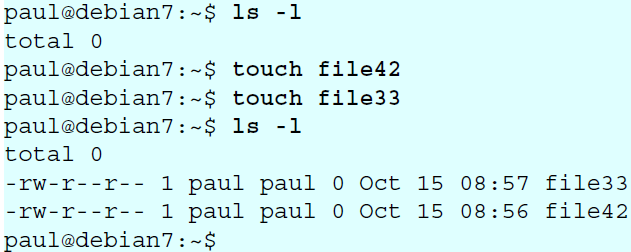
🡪The **file** utility determines the file type. Linux does not use extensions to determine the file type. The command line does not care whether a file ends in .txt or .pdf. As a system administrator, you should use the file command to determine the **file** type. Here are some examples on a typical Linux system.



**🡺touch**

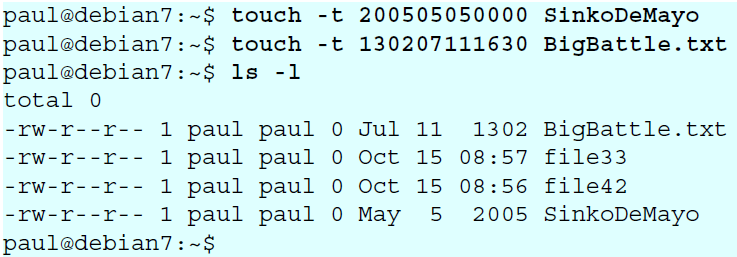
* 1. **Create an empty file (touch)**

🡪One easy way to create an empty file is with **touch**. (We will see many other ways for creating files later in this book.)



* 1. **touch -t**

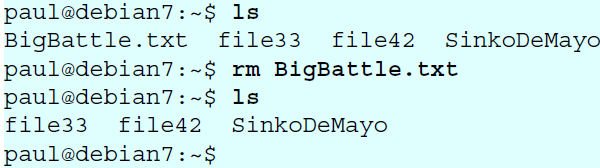
🡪The **touch** command can set some properties while creating empty files. Can you determine what is set by looking at the next screenshot? If not, check the manual for **touch**.



**🡺rm**

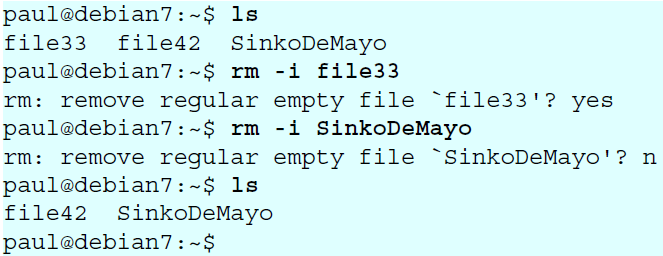
* 1. **Remove File (rm)**

🡪When you no longer need a file, use **rm** to remove it. Unlike some graphical user interfaces, **the command line in general does not have a** **waste bin** or **trash can to recover files**. When you use rm to remove a file, the file is gone. Therefore, be careful when removing files!



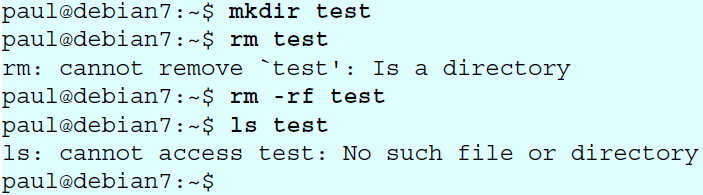
* 1. **rm -i**

**🡪To prevent yourself from accidentally removing a file, you can type rm -i.**

****

* 1. **rm -rf**

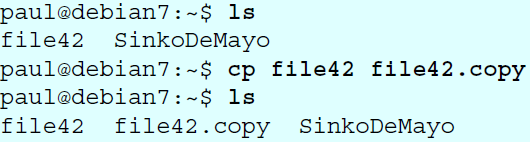
🡪By default, **rm -r** will not remove non-empty directories. However **rm** accepts several options that will allow you to remove any directory. The **rm -rf** statement is famous because it will erase anything (providing that you have the permissions to do so). When you are logged on as root, be very careful with **rm -rf (the f means force and the r means recursive)** since being root implies that permissions don't apply to you. You can literally erase your entire file system by accident.



**🡺cp**

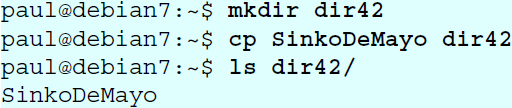
* 1. **copy One File (cp)**
     1. **Copy to same directory**

🡪To copy a file, use cp with a source and a target argument.



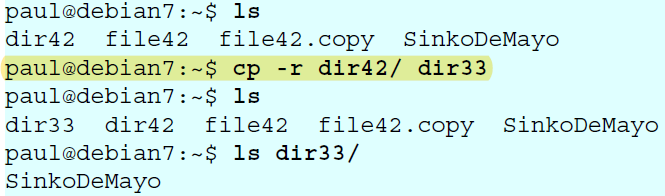
* + 1. **Copy to another directory**

🡪If the target is a directory, then the source files are copied to that target directory.



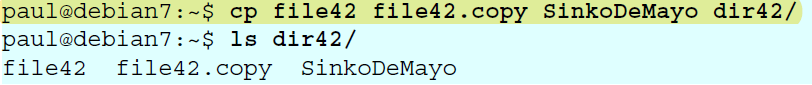
* 1. **cp -r**

🡪To copy complete directories, use **cp -r** (the **-r option forces recursive copying of all files in all subdirectories**).



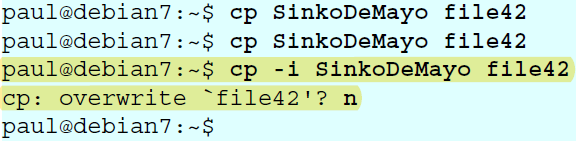
* 1. **Copy multiple files to directory**

**🡪You can also use cp to copy multiple files into a directory. In this case, the last argument (a.k.a. the target) must be a directory.**

****

* 1. **cp -i**

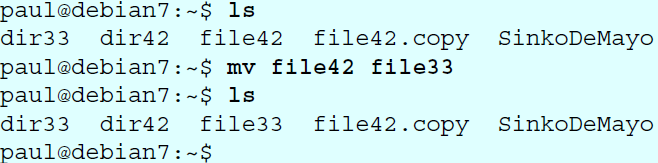
**🡪To prevent cp from overwriting existing files, use the -i (for interactive) option.**



**🡺mv**

* 1. **rename file with mv**

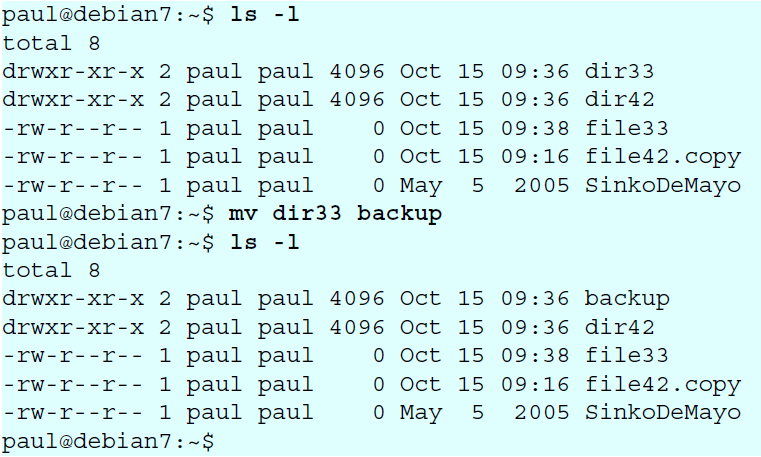
🡪Use **mv** to rename a file or to move the file to another directory.



🡪When you need to rename only one file then mv is the preferred command to use.

* 1. **rename directories with mv**

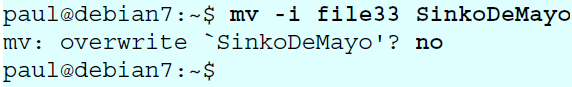
🡪The same **mv** command can be used to rename directories.



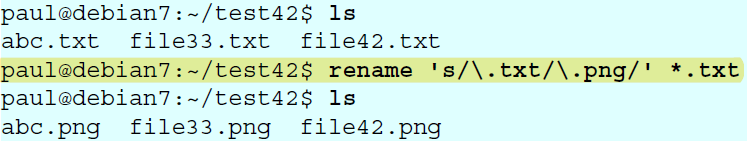
* 1. **mv -i**

🡪The mv also has a -i switch to cp and rm

🡪This screenshot shows that mv -i will ask permission to overwrite an existing file

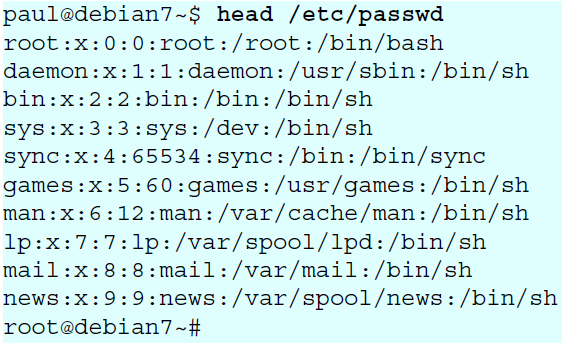


* 1. **Also rename in the Debian/ubuntu can also be done by help of the rename keyword**🡪Below a rename example that switches all occurrences of txt to png for all file names ending in .txt.

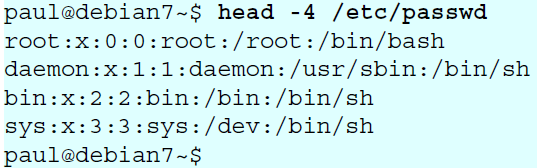
****

1. **Working With File Contents**
   1. **head**

🡪You can use **head** to display the first ten lines of a file



🡪The **head** command can also display the first **n** lines of a file

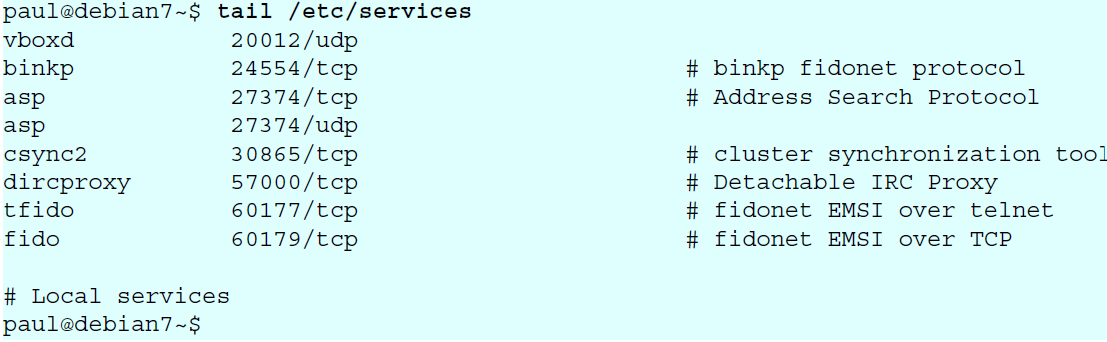


🡪The head can also display the first **n bytes.**

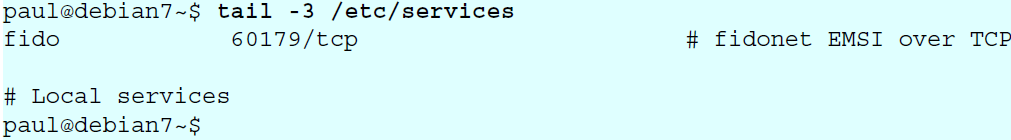
****

* 1. **tail**

🡪Similar to the head, tail command will display the last ten lines of a file.



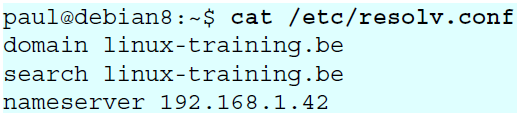
🡪You can give the tail to the number of the lines you want to see



🡪The tail command also has some other useful commands which we will see in further sections

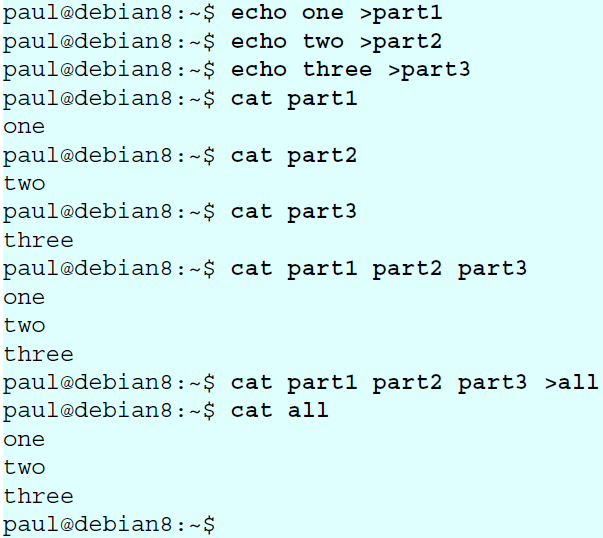
* 1. **cat**

🡪The cat command is one of the most universal tools, yet all it does is copy **standard input to standard output**. **In combination with the shell this can be very powerful and diverse**. Some examples will give a glimpse into the possibilities. The first example is simple**, you can use cat to display a file on the screen**. If the file is longer than the screen, it will scroll to the end.



* 1. **concatenate**

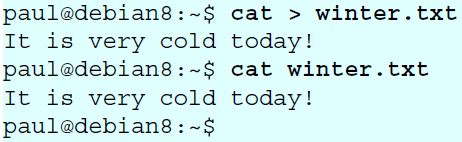
🡪**cat** is short for **concatenate**. One of the basic uses of **cat** is to concatenate files into a bigger (or complete) file.



* 1. **Create files**

🡪You can use cat to create flat text files. Type the cat > winter.txt command as shown in the screenshot below.

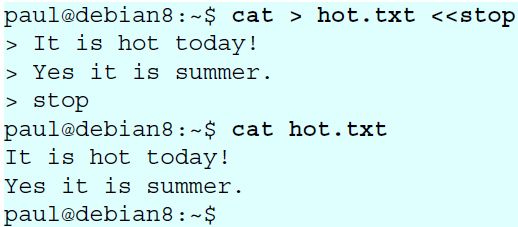
🡪**Then type one or more lines, finishing each line with the enter key. After the last line, type and hold the Control (Ctrl) key and press d**.



**🡪The Ctrl d key combination will send an EOF (End of File) to the running process ending the cat command.**

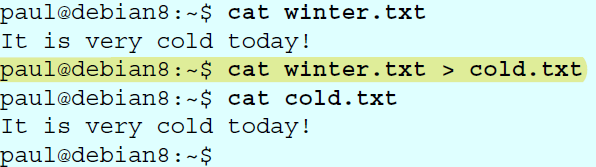
* 1. **Custom end marker**

🡪You can choose an end marker for **cat with << as is shown in this screenshot**. This construction **is called a here directive and will end the cat command**.



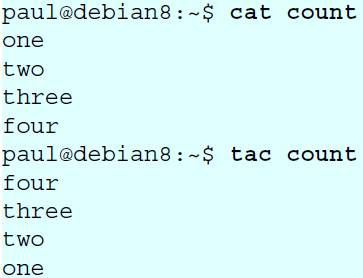
* 1. **Copy files**

🡪In the third example you will see that cat can be used to copy files. We will explain in detail what happens here in the bash shell chapter.



* 1. **tac**

🡪Just one example will show you the purpose of **tac** (cat backwards).



* 1. **more and less**

🡪The **more** command is useful for displaying files that take up more than one screen. **More** will allow you to see the contents of the file page by page. Use the **space bar to see the next page**, or **q** to **quit**. Some people prefer the less command to more.

1. **The Linux File Tree**
2. **filesystem hierarchy standard**

🡪Many Linux distributions partially follow the Filesystem Hierarchy Standard. The FHS

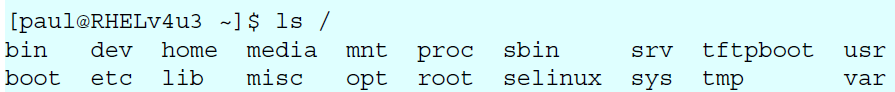
may help make more Unix/Linux file system trees conform better in the future. The FHS

is has been designed to be used by Unix distribution developers, package developers, and system implementers.

1. **the root directory /**

🡪All Linux systems have a directory structure that starts at the root directory. The **root**

**directory** is represented by a **forward slash**, like this: **/**. **Everything that exists on your Linux system can be found below this root directory**. Let's take a brief look at the contents of the root directory.

****

1. **binary directories**

🡪**Binaries** are files that contain compiled source code (or machine code). Binaries can be **executed** on the computer. Sometimes binaries are called **executables**.

🡪The **/bin** directory contains **binaries** for use by all users. According to the FHS the **/bin** directory should contain **/bin/cat** and **/bin/date** (among others).

1. **/boot**

🡪The **/boot** directory contains all files needed to boot the computer. These files don't change very often. On Linux systems you typically find the **/boot/grub** directory here.

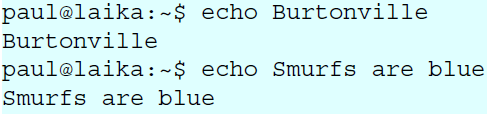
1. **Commands and Arguments**

🡪This chapter introduces you to **shell expansion** by taking a close look at **commands** and **arguments**. Knowing **shell expansion** is important because many **commands** on your Linux system are processed and most likely changed by the **shell** before they are executed.

🡪The command line interface or **shell** used on most Linux systems is called **bash**, which stands for **Bourne again shell**. The **bash** shell incorporates features from **sh** (the original Bourne shell), **csh** (the C shell), and **ksh** (the Korn shell).

🡪This chapter frequently uses the echo command to demonstrate shell features. **The echo**

**command is very simple: it echoes the input that it receives**.

****

1. **Arguments**

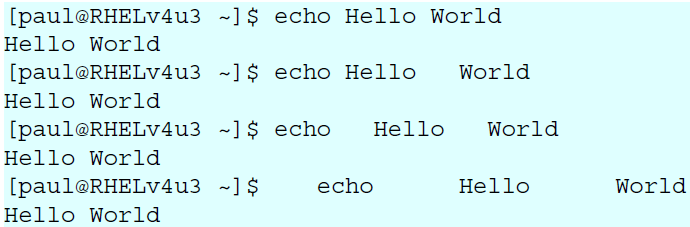
🡪One of the primary features of a shell is to perform a **command line scan**. When you enter a command at the shell's command prompt and press the enter key, then the shell will start scanning that line, cutting it up in **arguments**. While scanning the line, the shell may make many changes to the **arguments** you typed.

🡪This process is called **shell expansion**. When the shell has finished scanning and modifying that line, then it will be executed.

1. **White space removal**

🡪Parts that are separated by one or more consecutive **white spaces** (or tabs) are considered separate **arguments**, any white space is removed. The first **argument** is the command to be executed, the other **arguments** are given to the command. The shell effectively cuts your command into one or more arguments.

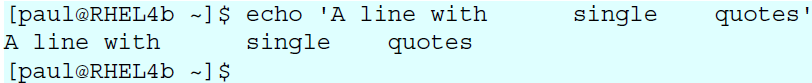
🡪This explains why the following four different command lines are the same after **shell expansion**.



🡪The **echo** command will display each argument it receives from the shell. The **echo** command will also add a new white space between the arguments it received.

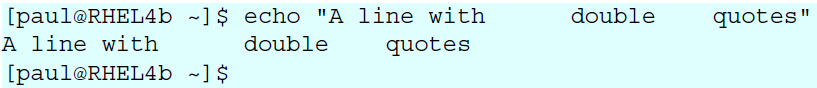
1. **single quotes**

🡪You can prevent the removal of white spaces by quoting the spaces. **The contents of the quoted string are considered as one argument**. In the screenshot below the **echo** receives only one **argument**.



1. **double quotes**

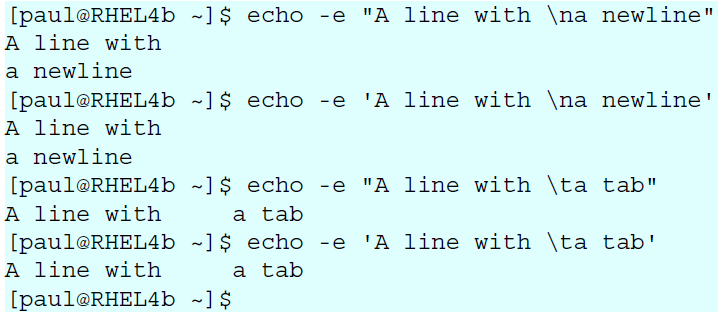
🡪You can also prevent the removal of white spaces by double quoting the spaces. Same as above, **echo** only receives **one argument**.



🡪Later we will see the importance of the single and double quotes in the further section.

1. **echo and quotes**

🡪Quoted lines can include special escaped characters recognized by the **echo** command (when using **echo -e**). The screenshot below shows how to use **\n** for a newline and **\t** for a tab (usually eight white spaces).



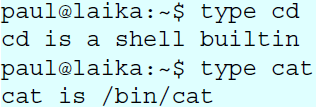
🡪The echo command can generate more than white spaces, tabs and newlines. Look in the man page for a list of options.

1. **Commands**
   1. **external or built-in commands?**

🡪Not all commands are external to the shell, some are **builtin**. **External commands** are programs that have their own binary and reside somewhere in the file system. Many external commands are located in **/bin** or **/sbin**. **Builtin commands** are an integral part of the shell program itself.

* 1. **Type**

🡪To find out whether a command given to the shell will be executed as an **external command** or as a **builtin command**, use the **type** command.



🡪As you can see, the **cd** command is **builtin** and the **cat** command is **external**.

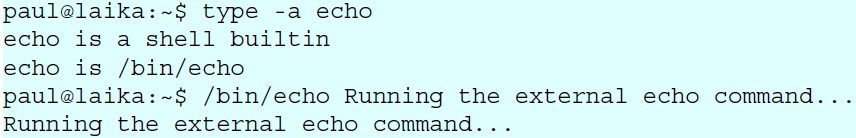
🡪You can also use this command to show you whether the command is **aliased** or not.



* 1. **running external commands**

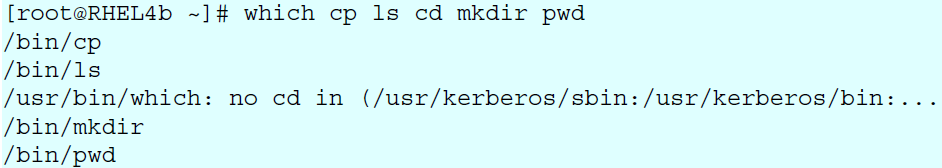
🡪Some commands have both builtin and external versions. When one of these commands is executed, the builtin version takes priority.

**🡪To run the external version, you must enter the full path to the command.**

****

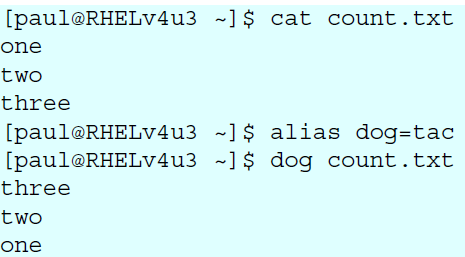
* 1. **which**

🡪The **which** command will search for binaries in the **$PATH** environment variable (variables will be explained later). In the screenshot below, it is determined that **cd** is **builtin**, and **ls**, **cp**, **rm**, **mv**, **mkdir**, **pwd**, and **which** are **external commands**.



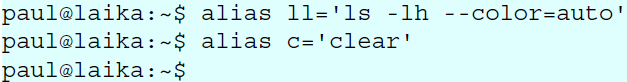
* 1. **aliases**
     1. **create an alias**

🡪The shell allows you to create **aliases**. Aliases are often used to create an easier to remember name for an existing command or to easily supply parameters.



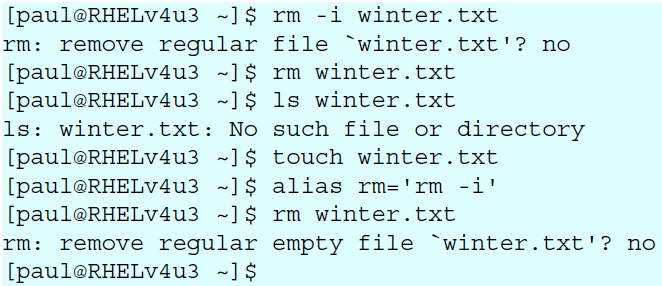
* + 1. **abbreviate commands**

🡪An **alias** can also be useful to abbreviate an existing command.



* + 1. **default options**

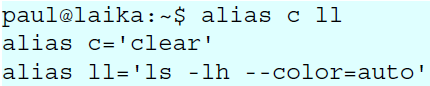
🡪Aliases can be used to supply commands with default options. The example below shows how to set the -i option default when typing **rm**.



🡪Some distributions enable default aliases to protect users from accidentally erasing files ('rm -i', 'mv -i', 'cp -i')

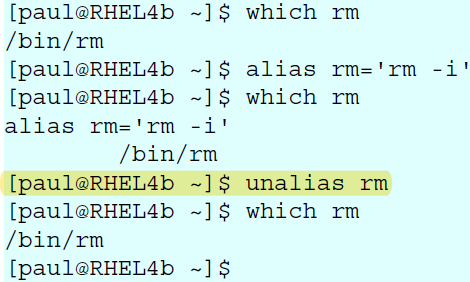
* + 1. **viewing aliases**

🡪You can provide one or more aliases as arguments to the **alias** command to get their definitions. Providing no arguments gives a complete list of current aliases.



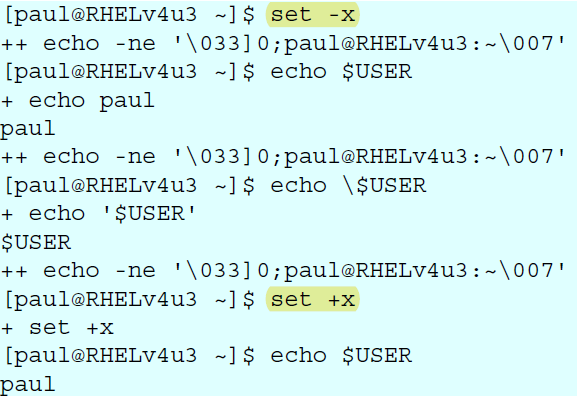
* + 1. **unalias**

🡪You can undo an alias with the **unalias** command.



* 1. **displaying the shell expression**

🡪You can display shell expansion with **set -x**, and stop displaying it with **set +x**.



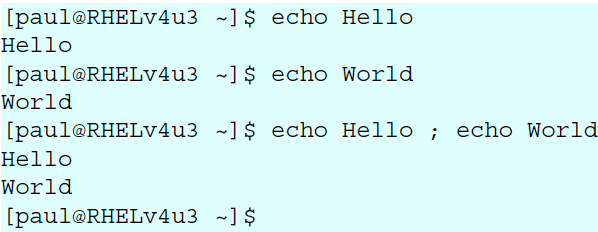
1. **Control Operators**

🡪In this chapter we put more than one command on the command line using **control**

**operators**. We also briefly discuss related parameters ($?) and similar special characters(&).

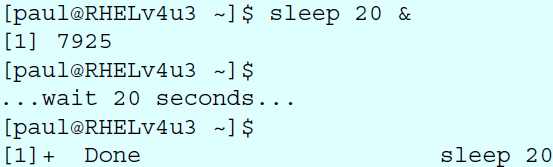
1. **; semicolon**

🡪You can put two or more commands on the same line separated by a semicolon **;** . The shell will scan the line until it reaches the semicolon. All the arguments before this semicolon will be considered a separate command from all the arguments after the semicolon. Both series will be executed sequentially with the shell waiting for each command to finish before starting the next one.



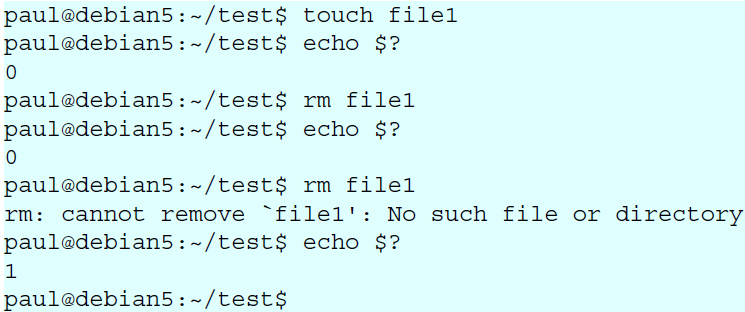
1. **& ampersand**

🡪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.



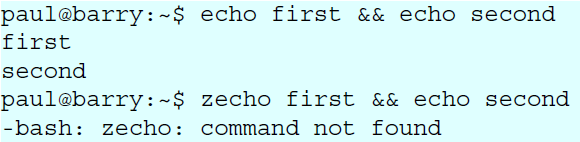
1. **$? Dolar question mark**

🡪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 **$?**.



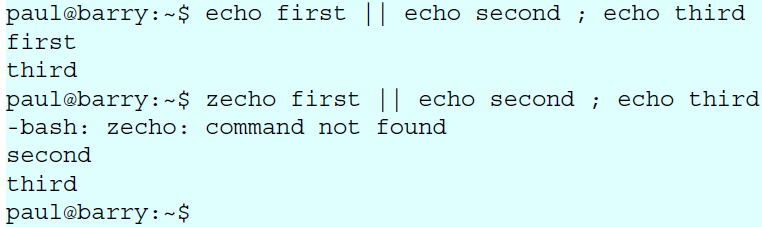
1. **&& double ampersand**

🡪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).



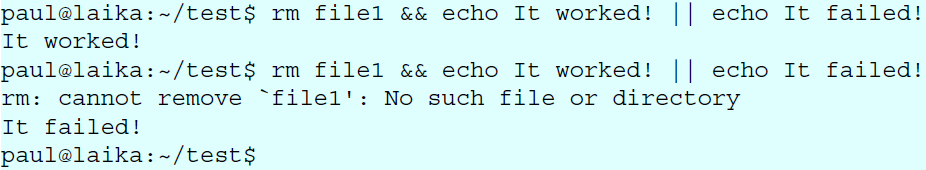
1. **|| double vertical bar**

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



1. **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.



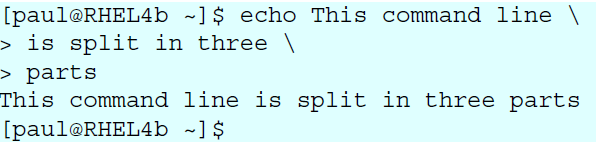
1. **# pound sign**

🡪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.

1. **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.



1. **Shell Variables**

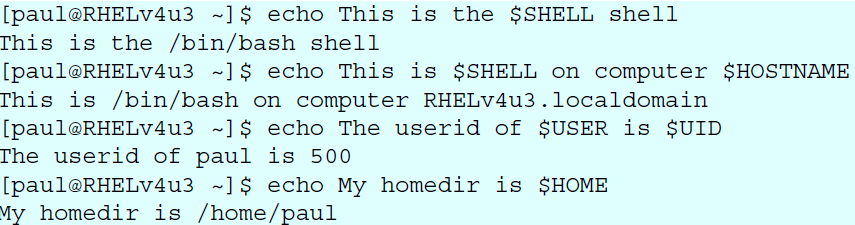
🡪In this chapter we learn to manage environment variables in the shell. These variables are

often needed by applications.

1. $ dollar sign

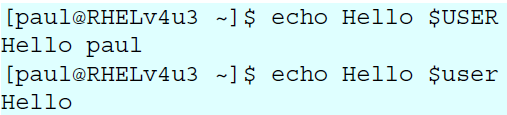
🡪Another important character interpreted by the shell is the dollar sign **$**. The shell will look for an **environment variable** named like the string following the **dollar sign** and replace it with the value of the variable (or with nothing if the variable does not exist).

🡪These are some examples using $HOSTNAME, $USER, $UID, $SHELL, and $HOME.



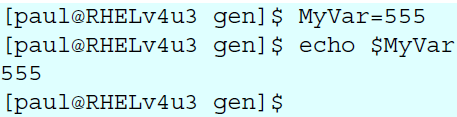
1. case sensitive

🡪This example shows that shell variables are case sensitive!



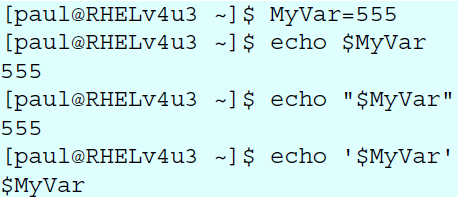
1. creating variables

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

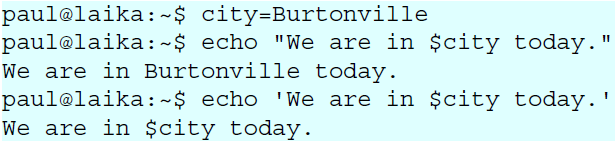


1. quotes

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



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



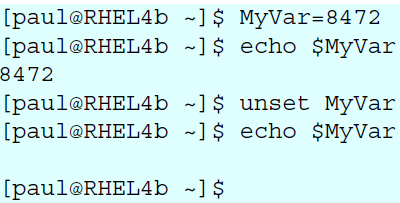
1. set

🡪You can use the **set** command to display a list of environment variables.

🡪On Ubuntu and Debian systems, the **set** command will also list shell functions after the shell variables. Use **set | more** to see the variables then.

1. unset

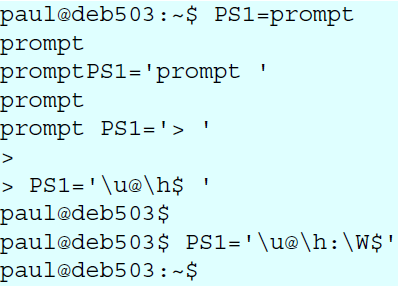
🡪Use the **unset** command to remove a variable from your shell environment.



1. $PS1

🡪The **$PS1** variable determines your shell prompt. You can use backslash escaped special characters like **\u** for the username or **\w** for the working directory. The **bash** manual has a complete reference.

🡪In this example we change the value of **$PS1** a couple of times.



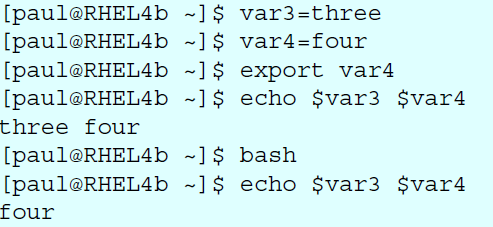
1. $PATH

🡪The **$PATH** variable is determines where the shell is looking for commands to execute (unless the command is builtin or aliased). This variable contains a list of directories, separated by colons.

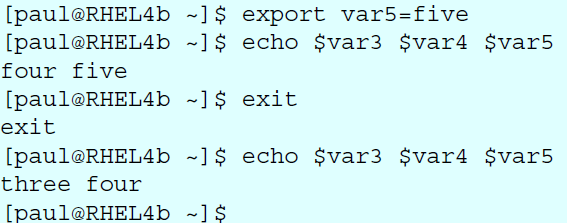


1. export

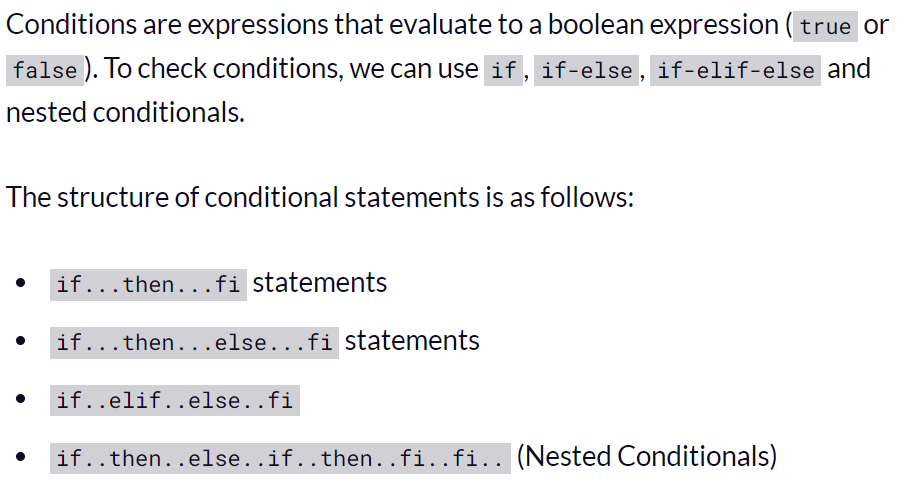
🡪You can export shell variables to other shells with the **export** command. This will export the variable to child shells.

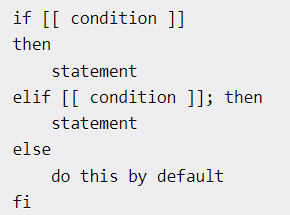


🡪But it will not export to the parent shell (previous screenshot continued).

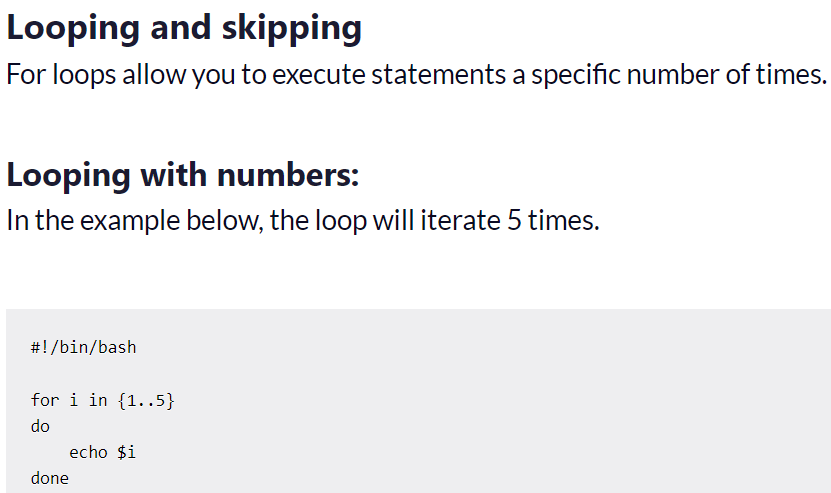


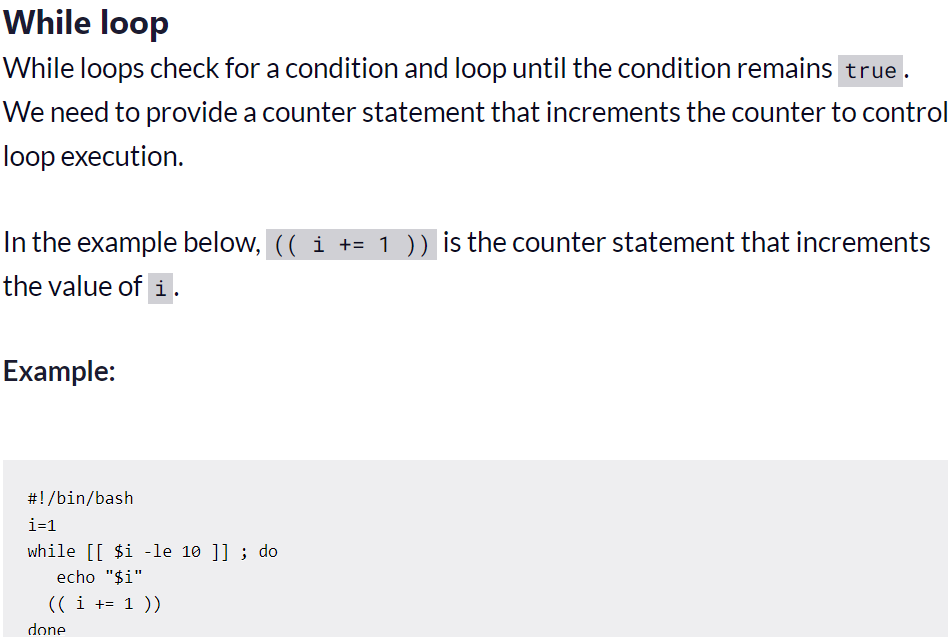
1. **Conditional Statements (Decision Making)**

****

****

1. **Looping and skipping**

****

****