Introduction to Linux Command-Line for Beginners

This tutorial went through some changes since I've posted it on the TuxArena Blog over one year ago, and I believe this version is more complete and well-organized. This introductory tutorial addresses new and average Linux users rather than experts who will already know quite everything is explained here. I am no expert, so if you want to help me improve this tutorial or have some other suggestions or corrections please feel free to contact me at craciun.dan@tuxarena.com or leave a comment on the TuxArena website.

The tutorial aims first-time users and average Linux users and its goal is to make you learn your way around when using the Linux command-line interface (or **CLI** for short). It also encourages some habits which I believe that, although hard to learn at first, will make you more productive later on.

I divided this guide in several sections, so feel free to jump to whichever you want using the links below:

What Is Linux?

To some, the first thing that comes in mind when asked this question is "*Linux is an operating system.*" That's not completely false, but neither accurately true. Linux per se is only the **kernel** of the operating system, the central part of it. A **Linux-based operating system** comprises of the Linux kernel, the GNU tools and utilities (like the Bash shell, or the GCC compiler), desktop environments (like KDE and GNOME), and finally, all the other applications, graphical or not (like a music player or an image editor).

What you are installing on your computer is called a Linux distribution, or a Linux-based operating system. A Linux distribution takes all the parts mentioned above and assembles them, eventually polishing and customizing them. There are hundreds of good Linux distributions out there, and I couldn't contain all of them here. Some examples include the popular Ubuntu, Debian, openSUSE or Fedora. You can find more information about Linux distributions at DistroWatch or on Wikipedia.

Usually Linux is cost-free, but some companies may charge for it or for support. The kernel and all the GNU tools are licensed under a free, permissive license (the GPLv2 or GPLv3), which allows not only to download freely and redistribute it, but also to study and modify the source code.

Introduction: Basic Commands and Concepts

What is a shell?

A shell is a **command interpreter** which allows you to interact with the computer. The way things work is pretty simple: you type in commands, the shell interprets them, performs the tasks it was asked to do, and finally it sends the results to the standard output, which is usually the screen. Here's an example output of the **ls** command, which lists the files in a directory:

```
debian$ ls
bin    dev    home        media  proc  selinux  tmp  vmlinuz
boot   etc    lib         mnt    root  srv      usr  initrd.img
cdrom  floyd  lost+found  opt    sbin  sys      var
```

This is a list of files inside the root directory. The root directory is the first location in the filesystem tree hierarchy, and it is represented by the **slash** character: **/**.

Some of the most popular shells are:

- **bash** - the Bourne-Again Shell, the default shell on most Linux systems
- **sh** - the Bourne Shell, an older shell which is not so widely used anymore
- **csh** - the C Shell, which accepts a syntax which resembles the C programming language
- **tcsh** - an improved version of the C Shell
- **ksh** - the Korn Shell, initially developed in the early 1980s
- **dash** - Debian Almquist Shell, a shell created by the Debian distribution

In this tutorial we will focus on **Bash**, since it is the most widely used and also one of the most powerful shells out there. Bash is a modern implementation of the older Bourne Shell (**sh**), developed by the GNU project, which provides a huge amount of tools and which, together with the Linux kernel, desktop environments like GNOME or KDE and applications which run on top of them, comprise the whole Linux platform. On a Debian or Ubuntu distribution, the default shell used by the system is specified in the file **/etc/passwd** (default being Bash).

Starting up a terminal

To access the shell we will use a shell-like application, also called a terminal emulator. There is a huge number of good terminal applications out there, including the default ones in GNOME or KDE, or Yakuake, Guake, rxvt and so on. For now let's just stick with the default that comes with your system. If you're using GNOME you can access the terminal by going to **Applications->Accessories->Terminal** or pressing Alt+F2 and typing **gnome-terminal** in the run box that appears, followed by Enter. If you're running KDE you can type instead **konsole** after pressing Alt+F2. Here's how your terminal should look like:

Depending on your distribution, the prompt may look something like **user@host$** . The first part before the ampersand is your login username, and the other one is the hostname of your computer.

Moving Around

What are Linux commands?

Linux commands are executable binary files which can be ran to perform certain tasks, like for example listing the files in a directory running an entire graphical application. Examples of frequently used commands are **ls**, **cd**, **pwd**, **date** or **cat**. With the exception of executable files, there is also a category called shell built-ins, which are built into the shell (Bash in our case). We'll deal with those later.

The general form of a Linux command is:

```
command options(s) filename(s)
```

Which specifies a command, followed by one or more parameters, and optionally one or more files to apply it on. For example:

```
$ echo -e 'Hello, world!\n'
```

Will output the text 'Hello, world!' followed by a newline character. The **-e** parameter (also called argument, or switch in this case) tells the echo command to interpret escaped characters, like the trailing **\n**, which will add a newline after the text inside the single quotes. Ignore the leading dollar sign, it just signifies the shell prompt.

A command may or may not have arguments. An argument can be an option or a filename.

Moving around using *cd* and *pwd*

Usually when you start a terminal the default starting location (called the **current working directory**) is your home folder, **/home/your_username/**. Let's say we want to move to the root directory. We will issue the following command:

```
cd /
```

As you can see, we used the **cd** (**c**hange **d**irectory) command, followed by a single argument, the path where we want to go (the root directory is represented by the slash sign). Now let's vizualize the contents of the root directory using the **ls** command:

```
$ ls
bin    dev    home        media  proc  selinux  tmp  vmlinuz
boot   etc    lib         mnt    root  srv      usr  initrd.img
cdrom  floyd  lost+found  opt    sbin  sys      var
```

These are all files and directories located in our root folder. If we would've wanted to list the contents of this folder directly from where we were, in the home folder, we would've called **ls** with a single argument, the slash sign:

```
$ ls /
bin    dev    home        media  proc  selinux  tmp  vmlinuz
boot   etc    lib         mnt    root  srv      usr  initrd.img
cdrom  floyd  lost+found  opt    sbin  sys      var
```

And it would've had the same result.

Now let's go back inside our home directory and list its contents. To do this, we have two options: **cd** without any parameters always changes the working directory to the home of the current user (unless it is aliased). Alternately, you could type:

```
$ cd /home/your_username/
```

And you should be back in your home folder. Don't forget to replace **your_username**. If you don't know what your username is type **echo $USER**. On my computer this would output 'embryo', which is the user I'm currently logged in with:

```
debian$ echo $USER
embryo
```

Now let's see what the current working directory is. We will use the **pwd** (**p**rint **w**orking **d**irectory) command to do so:

```
$ pwd
/home/embryo/
```

Now let's see the contents of our home directory using the **ls** command. It should output something like the following (yours may vary):

```
$ ls
Downloads  Music  my_stuff  textfile
```

There is more to **ls** though. Let's say we want to also list hidden files and show detailed information about each separate file. For this we will group several switches and pass them to ls:

```
$ ls -lha
total 20K
drwxr-xr-x  2 embryo embryo 4.0K 2010-12-19 15:12 Downloads
drwxr-xr-x  2 embryo embryo 4.0K 2010-12-19 15:12 Music
drwxr-xr-x  2 embryo embryo 4.0K 2010-12-19 15:21 my_stuff
-rw-r--r--  1 embryo embryo    0 2010-12-19 15:21 textfile
drwxr-xr-x  5 embryo embryo 4.0K 2010-12-19 15:32 .
drwxr-xr-x 62 embryo embryo 4.0K 2010-12-19 15:12 ..
-rw-r--r--  1 embryo embryo    0 2010-12-19 15:32 .bashrc
```

We grouped three different switches and passed them to ls: the **l** switch tells ls to use the long listing format (with detailed information about each separate file), the **h** switch tells ls to show file sizes in a human-readable format (kilobytes, megabytes or gigabytes where is the case instead of bytes) and finally, the **a** switch tells ls to also show the hidden files (the ones that start with a dot - in our case **.**, **..** and **.bashrc**. You will have more in your home directory.

Regarding the dot and double dot: these are also files, in that they are virtually meaning the current directory and the parent directory, respectively. So, **ls ..** will list the contents of the parent directory, which in our case is **/home/**. **ls .** will list the contents of the current directory just like **ls** without arguments does. Bash uses expansion to replace these with their actual meaning. So issuing **ls ..** is read by Bash exactly like you would type **ls /home/**, since /home/ is the parent directory relative to what the current working directory is.

Absolute and relative paths

An **absolute path** is a complete path, and it always starts with the root directory (slash sign). **/home/embryo/**, **/**, **/usr/bin/**, these are all absolute paths. They specify the complete path up to a certain point.

On the other hand, a **relative path** will always take into consideration the current working directory. So if you are, say, in the directory **/home/** and you want to change to, say, **/home/embryo/Downloads** you could do it like this:

```
$ cd /home/embryo/Downloads
```

In this example we specified the full path to the Downloads directory, or the absolute path. Notice the leading / sign for root. To do it using a relative path we would issue this command:

```
$ cd embryo/Downloads
```

Notice that we only specify the path from where we are, and there is no leading slash sign.

Go back to your home folder (use **cd**) and let's see how the double dot works. As I already mentioned, the double dot means the parent directory of the directory we're currently in. So let's say we want to go to the root directory again, but this time using double dot. Here's the command:

```
cd ../../
```

Which tells cd to go up to the parent directory, and then go up in the tree hierarchy one more time. You should be in the root directory now (check with **pwd**). The trailing slash in the above command is not necessary.

Performing simple tasks

Now let's focus on performing simple tasks like creating directories, view date and time, or simply playing around with some more basic commands.

Creating new files and directories
To create a new directory, we will use the **mkdir** (make directory) command. First, make sure you are in your home directory with **pwd**. If not, type **cd** to get there. Now, let's create a new directory:

```
debian$ mkdir myfolder
```

We created a new directory called **myfolder**. If you list your files now, your new folder should appear:

```
debian$ ls
Downloads  Music  myfolder  my_stuff  textfile
```

Now let's go inside our newly created folder, only this time using **cd ~/myfolder/**. The tilda sign signifies your home directory, and it expands into /home/your_username/. So issuing **cd ~/myfolder/** is the same as issuing **cd /home/embryo/myfolder/**. Now let's create another folder, called **docs**:

```
$ mkdir docs
```

Now let's create an empty file using the **touch** command:

```
$ touch emptyfile
```

The **touch** command creates an empty file with the name of its argument if that file doesn't already exist, or it updates the last modification date and time if it exists. Now since we created an empty folder and an empty file, let's see how we can delete them.

To remove files or folders, we will use the **rm** command. To remove our newly created file type:

```
$ rm emptyfile
```

And to remove the **docs** directory:

```
$ rm -r docs
```

Notice that to remove a directory we need the -r switch.

Copying and moving files
Let's go back inside **/home/your_username/myfolder/** and create a new directory and a new file:

```
mkdir newdir
touch newfile
```

Now let's use the **cp** command to copy the file newfile inside the directory newdir:

```
$ cp newfile newdir
```

This is the same as if we would've typed the following:

```
$ cp /home/embryo/myfolder/newfile /home/embryo/myfolder/newdir/
```

But in our first example we used a relative path.

Now let's create another file inside myfolder, and then move it to newdir:

```
touch file2
mv file2 newdir
```

We used the **mv** command to move it. Now let's create a new file inside myfolder, and rename it. Notice that in Linux renaming is accomplished using the **mv** command, they are one and the same thing. Let's create the a file called **mynewfile** and rename it to **file3** (we will list the contents of the directory after each step to see the changes):

```
$ touch mynewfile
$ ls
mynewfile  newdir  newfile
$ mv mynewfile file3
debian$ ls
file3  newdir  newfile
```

Now let's go inside the **newdir** directory, and use the double dot to move file2 from there inside myfolder, and also rename it (we will list the contents after each step to reflect the changes):

```
$ cd newdir/
$ ls
newfile
$ mv newfile ../renamed_file
$ cd ..
$ ls
file3  newdir  newfile  renamed_file
```

So first we entered in newdir. The file **newfile** was there since we moved it earlier. Next, we moved it to the parent directory, but with a changed name, **renamed_file**. Next, we changed the working directory to the parent of newdir and we listed its contents. As a side note, if you copy or move a file and the new destination name is the name of a directory, the file will be moved inside that directory.

Date and time
To view the current system's date and time, we will use the **date** command:

```
debian$ date
Sun Dec 19 16:06:53 EET 2010
```

Notice that date can be invoked in various ways in order to format the output. For

example, to show only the time in an HH:MM:SS format (hour, minute, second) we would use:

```
$ date +"%H:%M:%S"
16:09:52
```

The **+"%H:%M:%S"** parameter formats the output of the date command. Another useful command is **cal**, which will show a calendar:

```
debian$ cal
   December 2010
Su Mo Tu We Th Fr Sa
            1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

More basic commands
You can try the **whoami** command, which will show your username:

```
$ whoami
embryo
```

The command **uname** is used to print system information. In combination with the **-a** (all) switch, it will print various information including hostname, kernel version, date and operating system:

```
$ uname -a
Linux debian 2.6.26-2-686 #1 SMP Thu Nov 25 01:53:57 UTC 2010 i686 GNU/Linux
```

The **who** command is used to show is logged on, the number of the tty (teletype terminal) or display they are using, and the date and time they logged in:

```
$ who
embryo   :0           2010-12-19 07:55
```

Another useful command is **uptime**, which will show for how long the system has been running:

```
$ uptime
 20:10:31 up 12:15,  1 user,  load average: 0.02, 0.02, 0.00
```

To print a line of text to the screen we'll use the **echo** command:

```
$ echo 'Hello, world!'
Hello, world!
```

Searching for commands
In order to see the path in which a command is located we will use the **whereis** command, which will search in all the standard locations:

```
$ whereis bash
bash: /bin/bash /etc/bash.bashrc /usr/share/man/man1/bash.1.gz
```

Notice that **whereis** will not look into non-standard paths. Instead, we can use the **which** command for that:

```
$ which wesnoth
/home/embryo/usr/bin/wesnoth
```

More Useful CLI Commands

Using cat and less

The **cat** command is used to concatenate one or more files and print the result to the

standard output. If only one file is specified as input, cat will print its contents to the screen. For example, here's the output of the **cat /etc/mtab** command, which prints the contents of the /etc/mtab file:

```
$ cat /etc/mtab
/dev/sda1 / ext3 rw,errors=remount-ro 0 0
tmpfs /lib/init/rw tmpfs rw,nosuid,mode=0755 0 0
proc /proc proc rw,noexec,nosuid,nodev 0 0
sysfs /sys sysfs rw,noexec,nosuid,nodev 0 0
procbususb /proc/bus/usb usbfs rw 0 0
udev /dev tmpfs rw,mode=0755 0 0
tmpfs /dev/shm tmpfs rw,nosuid,nodev 0 0
devpts /dev/pts devpts rw,noexec,nosuid,gid=5,mode=620 0 0
/dev/sda5 /floyd ext3 rw 0 0
```

So **cat** would be one way of vizualizing the contents of a file. However, if your terminal doesn't have scrollback history enabled, or you're in a tty and the file you're trying to read is too big, cat will only display the end of the file on a single page. This is where the **less** command is useful.

less will display the contents of a text file and allow you to navigate up and down to read it. Here are the keyboard shortcuts used by less:

- **J** - scroll down one line (equivalent with Ctrl-N or down arrow)
- **K** - scroll up one line (equivalent with Ctrl-P or up arrow)
- **Ctrl-V** - scroll down one page (equivalent with Page Down)
- **Alt-V** - scroll up one page (equivalent with Page Up)

To exit less press **Q**.

Using tar, gzip and bzip2

These are commands used to create archives and compressed files. **tar** is a tool which creates archives known as tar files, while **gzip** is a compressing tool which uses an advanced compression algorithm. **bzip2** is an advanced compression tool which takes more time to compress/uncompress files, but it offers a better compression ratio, which results and smaller files. bzip2 is usually used to save disk space and Internet bandwidth.

To uncompress a **.tar.gz** file you would issue a command like the following:

```
tar -xzf archive.tar.gz
```

Where **x** stands for extract, **z** specifies that the compressed file is a gzip file, and **f** stands for the filename.

To uncompress a **.tar.bz2** file you would use something like:

```
tar -xjf archive.tar.bz2
```

And finally, to create a **.tar.gz** compressed archive from a directory, use:

```
tar -czf output_file.tar.gz input_directory
```

This will create the compressed file output_file.tar.gz from the contents of input_directory, which will be the root directory.

The Power of the Shell

Standard GNU tools usually have two ways of specifying an option, the short and the long syntax. For example **ls -a** and **ls --all** will do the same thing, namely list all the hidden files in the current directory (files that start with a dot).

Parameters specified between the **[** and **]** brackets are optional, for example the synopsis **ls [OPTION]... [FILE]...** tells us that **OPTION** or **FILE** don't necessarily need to be specified.

Using TAB completion

TAB completion is one of the powerful features of the shell. Typing a few starting letters from a command and pressing TAB will auto-complete the whole command which starts with those few letters, or will offer the longest possible pattern in case

more commands share the same letters in the beginning. For example, if we type **ba** followed by TAB, the shell will show us possible completion variants, like **bash** or **batch**. This may not look interesting for system commands which are usually short, but it is a feature which will come in handy in time.

Using control characters

The shell uses certain key combinations to perform certain actions. These key combinations are called control characters, and although they may be hard to learn at start, once you've mastered them you will find them faster and powerful than using the arrow keys or the Home/End and Page Up/Page Down keys. Most of these key combinations are used by Emacs too, a very powerful development environment (and not only). Also notice that these are easier to use for a person who knows blind-typing, since they don't require to move your fingers away from the typing position. The caret (^) sign in front of every key means "press Ctrl and the following key". Here is the list:

- **^A** - go to the start of line
- **^E** - go to the end of line
- **^H** - erase one character to the left (same effect as Backspace)
- **^D** - erase one character to the right (same effect as Delete, it also exits the shell if there is no character to delete))
- **^U** - erase everything from the cursor to start
- **^K** - erase everything from the cursor to end
- **^P** - bring the previous command in history
- **^N** - bring the next command in history
- **^C** - interrupt character, sends SIGTERM to the current application

You should familiarize with those, since once learned they will prove quite productive.

The ^N and ^P commands will cycle through the commands in the order in which they were given, so if for example you type ^N twice, the shell will bring the command you issued before the last command. ^D is also called the EOF (end-of-file) character and it is used to tell a program that the input has ended and it can return its output. As an example, you can type **wc -l**, which will count the lines in a file, but without providing the filename. In this case wc waits for you to enter lines and after you have ended type Ctrl-D to tell it you entered the text and want to know how many lines you typed in.

Make use of Bash history

Bash keeps a command history so you can recall commands at a later time. Some of the commands available for this include the use of **!!**, which will recall and **execute** the last command, **!command_name**, which will recall and execute the last command in history which starts with **command_name** or **!?command_name?**, which will recall and execute the last command in history which contains the string **command_name**.

Linux Directory Structure

Linux, just like UNIX, uses a tree hierarchical directory structure. The common standard directories on a Linux system are briefly overviewed below:

- **/** - the root directory
  - **/bin** - contains important, vital utilities, like cp, ls or kill
  - **/sbin** - contains system administration utilities available to the root user, like ifconfig
  - **/boot** - contains the Linux kernel image used to boot the system
  - **/dev** - contains special files pointing to system devices
    - **/dev/null** - pseudo device used to pipe away unwanted data
  - **/etc** - contains configuration files for various applications
  - **/home** - contains home directories for system users
  - **/lib** - contains shared libraries available for all programs throughout the system
  - **/media** - mount points for devices
  - **/mnt** - mount points for filesystems
  - **/root** - home directory for the root superuser
  - **/proc** - virtual files providing system information
  - **/tmp** - stores temporary files
  - **/var** - logs, emails, printing jobs
  - **/usr** - all the programs which are not located in /bin or /sbin, documentation, manual pages, icons etc
    - **/usr/bin** - all binary applications not residing in /bin or /sbin
    - **/usr/include** - development headers used by the system
    - **/usr/lib** - shared libraries
    - **/usr/local** - applications which were not installed using the distribution's package manager; manually compiled applications go here

Environment Variables

Environment variables are special variables used by the shell and other applications to gather various information. For example, earlier in this tutorial we used the command **echo $USER**, which showed the currently logged in user. In this example, **$USER** is an environment variable. Some more examples include **$HOSTNAME**, **$PATH**, **$PWD**, **$SHELL**, **$HOME**, **$LANG**. And their output:

```
$ echo $HOSTNAME
debian
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:/home/embryo/bin:/home/embryo/sym:
/home/embryo/usr/local/bin:/home/embryo/usr/bin
$ echo $SHELL
/bin/bash
$ echo $LANG
en_US.UTF-8
```

Some of them can be changed inside the **~/.bashrc** file, for example the **$PATH** variable, which contains the locations in which to search for an executable file.

Bash Built-ins

As the name suggests, built-ins are commands which are provided by the Bash program itself rather than external commands provided by system binaries. To see all the built-ins included with Bash you can type **help**. Some examples would be:

- **.** - or **source**, read and execute shell commands from a filename
- **alias** - define new aliases
- **bg** - place the specified job as argument in the background
- **fg** - place the specified job as argument in the foreground and make it the current job
- **read** - read one line of input from the standard input
- **if** - conditional command; execute commands if the specified condition is true
- **while** - loop command; execute commands as long as the condition specified is true
- **help** - display all the available shell built-ins; **help command_name** will display help on the specified command

You can use the **type** command (which is also a built-in) to see if a command is a file or a shell built-in:

```
$ type type
type is a shell builtin
$ type bash
bash is /bin/bash
$ type cd
cd is a shell builtin
```

If the specified command is an alias, the output will be something like the following:

```
$ type ls
ls is aliased to `ls --color=auto -X'
```

In order to start an application in the background use the **&** sign after the command, e.g. **man bash &** will start the man command with the Bash manual page in background. To bring it in the foreground type **fg**, and then to put it back in the background press Ctrl-Z.

Pipes and Redirecting Output

Using pipes

Pipes are a powerful way to take the output of one command and feed it as input for another command. A pipe is represented by a vertical line (|). For example:

```
$ cat /etc/init.d/ifupdown | grep start
  start|restart)
      /etc/init.d/ifupdown-clean start
    echo "Usage: $0 {start|stop|restart|force-reload}" >&2
```

The **cat** (concatenate) command is used to concatenate the content of one or more files

and print the result to the output. Since in our example the only argument is a single filename, cat should print the contents of that file. The **grep** command is used to print lines that match a certain pattern, specified as a parameter.

The command we issued is **cat /etc/init.d/ifupdown | grep start**. The first command would normally print the entire contents of the file ifupdown to the output, but since we have a pipe here, the contents of the file will be the input for the **grep start** command. In the end, only the lines containing the text 'start' are printed. This command is equivalent to:

```
$ grep start /etc/init.d/ifupdown
```

Redirecting output

There are two common operators for redirecting output: **>** and **>>**. The first one will create a new file and write in it the output which was redirected. If the file already exists it will be **rewritten**. The second one will create a new file if it doesn't already exist and write in it the output which was redirected. If the file already exists, than the content will be **appended** to the already existing file.

Let's see how this works:

```
$ ls
file3  newdir  newfile  renamed_file
$ ls > list_files.txt
$ cat list_files.txt
file3
newdir
newfile
renamed_file
list_files.txt
```

First, we listed the contents of the current directory just to see what files are there. The next command, **ls > list_files.txt** will redirect the output of ls and will write it in the file list_files.txt. Next we show the contents of this file with **cat list_files.txt**. Now let's append the contents of **ls -l** to this file using **>>**, rather than rewriting it with the **>** operator:

```
$ ls -l >> list_files.txt
debian$ cat list_files.txt
file3
newdir
newfile
renamed_file
list_files.txt
total 8
-rw-r--r-- 1 embryo embryo    0 2010-12-19 16:21 file3
drwxr-xr-x 2 embryo embryo 4096 2010-12-19 16:24 newdir
-rw-r--r-- 1 embryo embryo    0 2010-12-19 16:15 newfile
-rw-r--r-- 1 embryo embryo    0 2010-12-19 16:21 renamed_file
-rw-r--r-- 1 embryo embryo   49 2010-12-19 19:20 list_files.txt
```

As you can see, the output of **ls -l** has been appended to list_files.txt.

Creating and Editing Files

In the previous section I showed how to create an empty file using **touch**, and now it's time to see how to create and edit text files using a text editor. For our example we'll use Nano, a user-friendly text editor that runs in a terminal. Nano is usually installed on any Linux distribution.

To create a new file with Nano, we will type in the following command:

```
$ nano newfile.txt
```

This will open up Nano so we can start editing our file:

Write some lines of text and then save the file using Ctrl+O followed by Enter. To quit Nano press Ctrl+X. Now you can view the content of your file using the **cat** command:

```
$ cat newfile.txt
some line of text
```

Getting Help

The -h and --help options

There are several ways of seeing how a command works and what are the valid arguments it can take. Usually all GNU tools provide the **-h** or **--help** parameters, which both do the same thing: show a brief description of the available parameters to the respective command and their meaning. For example, **bash --help** will show the Bash version and several options available to it. However, Bash is quite complex and very powerful, and such a short description usually won't suffice. This is where the manual pages get in the picture.

Manual pages

A manual page (or man page for short) usually contains all the options that a command accepts, with explanations for each of them, and a general description of the application. Only learning to get the habit of searching the man page is very helpful, since sometimes not even online searching will clarify a specific area like a man page does. To see a manual page for an application, just type **man** followed by the application's name, for example **man bash** or **man man**.

To navigate through the man page use the keys K (to scroll up) or J (to scroll down), or Ctrl-P (up) and Ctrl-N (down), or still Ctrl-V (down one page) and Alt-V (up one page). When you're done reading, type Q to exit man.

Helpful Links & Further Reading

There is a huge number of additional websites and resources which cover Linux CLI and Linux in general, and I can't list all of them here. However, I'd like to recommend two particular ones, created specially for beginners. They are:

- TuxFiles.org - a collection of great tutorials for Linux beginners
- LinuxCommand.org - a very good introduction to Linux CLI and basic scripting

Do you have suggestions or corrections to this guide? Please feel free to speak your mind here.

Updated: Dec 19, 2010 | v1.0

If you have suggestions or corrections to these tutorials, please contact me at craciun.dan@tuxarena.com or leave a comment on the TuxArena website.