# **UNIX \_Day 1**

The UNIX operating system has for many years formed the backbone of the Internet, especially for large servers and most major university campuses. However, a free version of UNIX called [**Linux**](http://www.linux.org/) has been making significant gains against Macintosh and the Microsoft Windows 95/98/NT environments, so often associated with personal computers. Developed by a number of volunteers on the Internet such as the Linux group and the GNU project, much of the open-source software is copyrighted, but available for free. This is especially valuable for those in educational environments where budgets are often limited.

UNIX commands can often be grouped together to make even more powerful commands with capabilities known as **I/O redirection** ( **<** for getting input from a file input and **>**for outputing to a file ) and **piping** using **|** to feed the output of one command as input to the next. Please investigate manuals in the lab for more examples than the few offered here.

The following charts offer a summary of some simple UNIX commands. These are certainly not all of the commands available in this robust operating system, but these will help you get started.

**Ten ESSENTIAL UNIX Commands**

These are ten commands that you really need to know in order to get started with UNIX. They are probably similar to commands you already know for another operating system.

|  |  |  |
| --- | --- | --- |
| **Command** | **Example** | **Description** |
| 1.    **ls** | ls ls -alF | Lists files in current directory List in long format |
| 2.    **cd** | cd tempdir cd .. cd ~dhyatt/web-docs | Change directory to tempdir Move back one directory Move into dhyatt's web-docs directory |
| 3.    **mkdir** | mkdir graphics | Make a directory called graphics |
| 4.    **rmdir** | rmdir emptydir | Remove directory (must be empty) |
| 5.    **cp** | cp file1 web-docs cp file1 file1.bak | Copy file into directory Make backup of file1 |
| 6.    **rm** | rm file1.bak rm \*.tmp | Remove or delete file Remove all file |
| 7.    **mv** | mv old.html new.html | Move or rename files |
| 8.    **more** | more index.html | Look at file, one page at a time |
| 9.    **lpr** | lpr index.html | Send file to printer |
| 10.  **man** | man ls | Online manual (help) about command |

**Ten VALUABLE UNIX Commands**

Once you have mastered the basic UNIX commands, these will be quite valuable in managing your own account.

|  |  |  |
| --- | --- | --- |
| **Command** | **Example** | **Description** |
| 1.    **grep <str><files>** | grep "bad word" \* | Find which files contain a certain word |
| 2.    **chmod <opt> <file>** | chmod 644 \*.html chmod 755 file.exe | Change file permissions read only Change file permissions to executable |
| 3.    **passwd** | passwd | Change passwd |
| 4.    **ps <opt>** | ps aux ps aux  **|**   grep dhyatt | List all running processes by #ID List process #ID's running by dhyatt |
| 5.    **kill <opt> <ID>** | kill -9 8453 | Kill process with ID #8453 |
| 6.    **gcc (g++) <source>** | gcc file.c -o file g++ fil2.cpp -o fil2 | Compile a program written in C Compile a program written in C++ |
| 7.    **gzip <file>** | gzip bigfile gunzip bigfile.gz | Compress file Uncompress file |
| 8.    **mail         (pine)** | mail me@tjhsst.edu **<** file1 pine | Send file1 by email to someone Read mail using pine |
| 9.    **telnet <host>         ssh <host>** | telnet vortex.tjhsst.edu ssh -l dhyatt jazz.tjhsst.edu | Open a connection to vortex Open a secure connection to jazz as user dhyatt |
| 10.  **ftp <host> ncftp <host/directory>** | ftp station1.tjhsst.edu ncftp metalab.unc.edu | Upload or Download files to station1 Connect to archives at UNC |

**Ten FUN UNIX Commands**

These are ten commands that you might find interesting or amusing. They are actually quite helpful at times, and should not be considered idle entertainment.

|  |  |  |
| --- | --- | --- |
| **Command** | **Example** | **Description** |
| 1.    **who** | who | Lists who is logged on your machine |
| 2.    **finger** | finger | Lists who is on computers in the lab |
| 3.    **ytalk <user@place>** | ytalk dhyatt@threat | Talk online with dhyatt who is on threat |
| 4.    **history** | history | Lists commands you've done recently |
| 5.    **fortune** | fortune | Print random humerous message |
| 6.    **date** | date | Print out current date |
| 7.    **cal <mo> <yr>** | cal 9 2000 | Print calendar for September 2000 |
| 8.    **xeyes** | xeyes & | Keep track of cursor (in "background") |
| 9.    **xcalc** | xcalc & | Calculator ("background" process) |
| 10.  **mpage <opt> <file>** | mpage -8 file1   **|**  lpr | Print 8 pages on a single sheet and send to printer (the font will be small!) |

**Ten HELPFUL UNIX Commands**

These ten commands are very helpful, especially with graphics and word processing type applications.

|  |  |  |
| --- | --- | --- |
| **Command** | **Example** | **Description** |
| 1.    **netscape** | netscape & | Run Netscape browser |
| 2.    **xv** | xv & | Run graphics file converter |
| 3.    **xfig / xpaint** | xfig & (xpaint &) | Run drawing program |
| 4.    **gimp** | gimp & | Run photoshop type program |
| 5.    **ispell <fname>** | ispell file1 | Spell check file1 |
| 6.    **latex <fname>** | latex file.tex | Run LaTeX, a scientific document tool |
| 7.    **xemacs / pico** | xemacs (or pico) | Different editors |
| 8.    **soffice** | soffice & | Run StarOffice, a full word processor |
| 9.    **m-tools (mdir, mcopy,         mdel, mformat, etc. )** | mdir a: mcopy file1   a: | DOS commands from UNIX (dir A:) Copy file1 to A: |
| 10.  **gnuplot** | gnuplot | Plot data graphically |

**Ten USEFUL UNIX Commands:**

These ten commands are useful for monitoring system access, or simplifying your own environment.

|  |  |  |
| --- | --- | --- |
| **Command** | **Example** | **Description** |
| 1.    **df** | df | See how much free disk space |
| 2.    **du** | du -b subdir | Estimate disk usage of directory in Bytes |
| 3.    **alias** | alias lls="ls -alF" | Create new command "lls" for long format of ls |
| 4.    **xhost** | xhost + threat.tjhsst.edu xhost - | Permit window to display from x-window program from threat Allow no x-window access from other systems |
| 5.    **fold** | fold -s file1  **|**   lpr | Fold or break long lines at 60 characters and send to printer |
| 6.    **tar** | tar -cf subdir.tar subdir tar -xvf subdir.tar | Create an archive called subdir.tar of a directory Extract files from an archive file |
| 7.    **ghostview (gv)** | gv filename.ps | View a Postscript file |
| 8.    **ping    (traceroute)** | ping threat.tjhsst.edu traceroute www.yahoo.com | See if machine is alive Print data path to a machine |
| 9.    **top** | top | Print system usage and top resource hogs |
| 10.  **logout (exit)** | logout or exit | How to quit a UNIX shell. |

**#1) cal**: Displays the calendar.

**Syntax**: cal [[month] year]

**Example**: display the calendar for April 2018

$ cal 4 2018

**#2) date:** Displays the system date and time.

**Syntax**: date [+format]

**Example**: Display the date in dd/mm/yy format

$ date +%d/%m/%y

**#3) banner**: Prints a large banner on the standard output.

**Syntax**: banner message

**Example**: Print “Unix” as the banner

$ banner Unix

**#4) who**: Displays the list of users currently logged in

**Syntax**: who [option] … [file][arg1]

**Example**: List all currently logged in users

$ who

**#5) whoami**: Displays the user id of the currently logged-in user.

**Syntax**: whoami [option]

**Example**: List currently logged in user

$ whoami

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

The Bourne Shell

The C Shell

The Korn Shell

The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

**Extended Shell Scripts**

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/sh

# Author : Zara Ali

# Copyright (c)

# Script follows here:

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Here is a sample run of the script −

$./test.sh

What is your name?

Zara Ali

Hello, Zara Ali

$

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

**Variable Names**

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

\_ALI

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invalid variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as **!**, **\***, or **-** is that these characters have a special meaning for the shell.

**Defining Variables**

Variables are defined as follows −

variable\_name=variable\_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

VAR1="Zara Ali"

VAR2=100

**Accessing Values**

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

[Live Demo](http://tpcg.io/AP7zgT)

#!/bin/sh

NAME="Zara Ali"

echo $NAME

The above script will produce the following value −

Zara Ali

**Read-only Variables**

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

[Live Demo](http://tpcg.io/tawT1C)

#!/bin/sh

NAME="Zara Ali"

readonly NAME

NAME="Qadiri"

The above script will generate the following result −

/bin/sh: NAME: This variable is read only.

**Unsetting Variables**

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

**Variable Types**

When a shell is running, three main types of variables are present −

**Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.

**Environment Variables** − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

**Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

These variables are reserved for specific functions.

For example, the **$** character represents the process ID number, or PID, of the current shell −

$echo $$

The above command writes the PID of the current shell −

29949

The following table shows a number of special variables that you can use in your shell scripts −

|  |  |
| --- | --- |
| **Sr.No.** | **Variable & Description** |
| 1 | **$0**  The filename of the current script. |
| 2 | **$n**  These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#**  The number of arguments supplied to a script. |
| 4 | **$\***  All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**  All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**  The exit status of the last command executed. |
| 7 | **$$**  The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!**  The process number of the last background command. |

**Command-Line Arguments**

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1"

echo "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

Here is a sample run for the above script −

$./test.sh Zara Ali

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

**Special Parameters $\* and $@**

There are special parameters that allow accessing all the command-line arguments at once. **$\*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $\* or $@ special parameters −

#!/bin/sh

for TOKEN in $\*

do

echo $TOKEN

done

Here is a sample run for the above script −

$./test.sh Zara Ali 10 Years Old

Zara

Ali

10

Years

Old

**Note** − Here **do...done** is a kind of loop that will be covered in a subsequent tutorial.

**Exit Status**

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

$./test.sh Zara Ali

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

$echo $?

0

$

Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

**Defining Array Values**

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

NAME01="Zara"

NAME02="Qadir"

NAME03="Mahnaz"

NAME04="Ayan"

NAME05="Daisy"

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

array\_name[index]=value

Here *array\_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands −

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

If you are using the **ksh** shell, here is the syntax of array initialization −

set -A array\_name value1 value2 ... valuen

If you are using the **bash** shell, here is the syntax of array initialization −

array\_name=(value1 ... valuen)

**Accessing Array Values**

After you have set any array variable, you access it as follows −

${array\_name[index]}

Here *array\_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept −

[Live Demo](http://tpcg.io/AMsECl)

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Index: ${NAME[0]}"

echo "Second Index: ${NAME[1]}"

The above example will generate the following result −

$./test.sh

First Index: Zara

Second Index: Qadir

You can access all the items in an array in one of the following ways −

${array\_name[\*]}

${array\_name[@]}

Here **array\_name** is the name of the array you are interested in. Following example will help you understand the concept −

[Live Demo](http://tpcg.io/r8Dol0)

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Method: ${NAME[\*]}"

echo "Second Method: ${NAME[@]}"

The above example will generate the following result −

$./test.sh

First Method: Zara Qadir Mahnaz Ayan Daisy

Second Method: Zara Qadir Mahnaz Ayan Daisy

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators −

Arithmetic Operators

Relational Operators

Boolean Operators

String Operators

File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers −

[Live Demo](http://tpcg.io/zURE2C)

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

The above script will generate the following result −

Total value : 4

The following points need to be considered while adding −

There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.

The complete expression should be enclosed between **‘ ‘**, called the backtick.

**Arithmetic Operators**

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[ $a == $b ]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

**Relational Operators**

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-relational-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a -eq $b ] is not true. |
| **-ne** | Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. | [ $a -ne $b ] is true. |
| **-gt** | Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true. | [ $a -gt $b ] is not true. |
| **-lt** | Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true. | [ $a -lt $b ] is true. |
| **-ge** | Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -ge $b ] is not true. |
| **-le** | Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -le $b ] is true. |

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[ $a <= $b ]** is correct whereas, **[$a <= $b]** is incorrect.

**Boolean Operators**

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-boolean-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **!** | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| **-o** | This is logical **OR**. If one of the operands is true, then the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

**String Operators**

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-string-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **=** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a = $b ] is not true. |
| **!=** | Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true. | [ $a != $b ] is true. |
| **-z** | Checks if the given string operand size is zero; if it is zero length, then it returns true. | [ -z $a ] is not true. |
| **-n** | Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true. | [ -n $a ] is not false. |
| **str** | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false. |

**File Test Operators**

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

[Show Examples](https://www.tutorialspoint.com/unix/unix-file-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true. | [ -k $file ] is false. |
| **-p file** | Checks if file is a named pipe; if yes, then the condition becomes true. | [ -p $file ] is false. |
| **-t file** | Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. | [ -t $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-s file** | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true. |

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

The **if...else** statement

The **case...esac** statement

**The if...else statements**

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement −

[if...fi statement](https://www.tutorialspoint.com/unix/if-fi-statement.htm)

[if...else...fi statement](https://www.tutorialspoint.com/unix/if-else-statement.htm)

[if...elif...else...fi statement](https://www.tutorialspoint.com/unix/if-elif-statement.htm)

Most of the if statements check relations using relational operators discussed in the previous chapter.

**The case...esac Statement**

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here −

[case...esac statement](https://www.tutorialspoint.com/unix/case-esac-statement.htm)

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

 A loop is a powerful programming tool that enables you to execute a set of commands repeatedly.

[The while loop](https://www.tutorialspoint.com/unix/while-loop.htm)

[The for loop](https://www.tutorialspoint.com/unix/for-loop.htm)

[The until loop](https://www.tutorialspoint.com/unix/until-loop.htm)

[The select loop](https://www.tutorialspoint.com/unix/select-loop.htm)

You will use different loops based on the situation. For example, the **while** loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, **while** and **for** loops are available in most of the other programming languages like **C**, **C++** and **PERL**, etc.

**Nesting Loops**

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting **while** loop. The other loops can be nested based on the programming requirement in a similar way −

**Nesting while Loops**

It is possible to use a while loop as part of the body of another while loop.

Syntax

while command1 ; # this is loop1, the outer loop

do

Statement(s) to be executed if command1 is true

while command2 ; # this is loop2, the inner loop

do

Statement(s) to be executed if command2 is true

done

Statement(s) to be executed if command1 is true

done

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine −

#!/bin/sh

a=0

while [ "$a" -lt 10 ] # this is loop1

do

b="$a"

while [ "$b" -ge 0 ] # this is loop2

do

echo -n "$b "

b=`expr $b - 1`

done

echo

a=`expr $a + 1`

done

This will produce the following result. It is important to note how **echo -n** works here. Here **-n** option lets echo avoid printing a new line character.

0

1 0

2 1 0

3 2 1 0

4 3 2 1 0

5 4 3 2 1 0

6 5 4 3 2 1 0

7 6 5 4 3 2 1 0

8 7 6 5 4 3 2 1 0

9 8 7 6 5 4 3 2 1 0