Introducción a la programación bash - LINUX

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

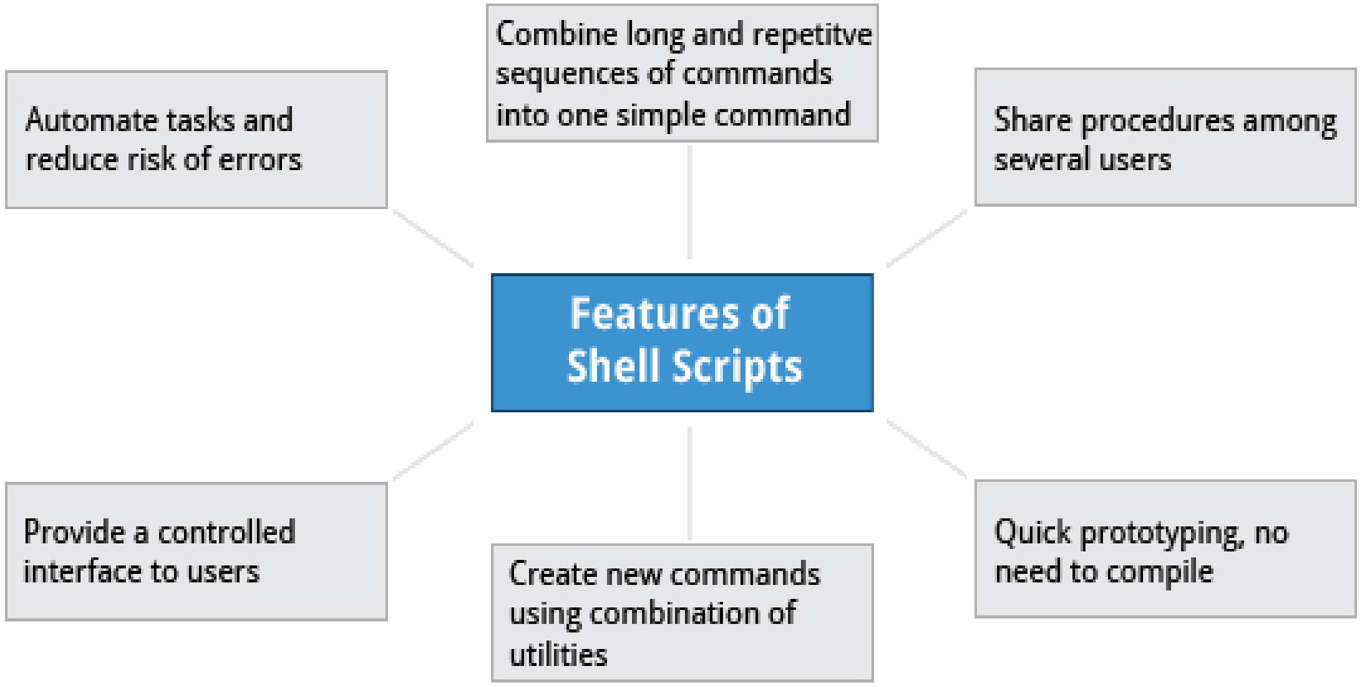
Universidad de Cantabria

`corcuerp@unican.es`

Índice General

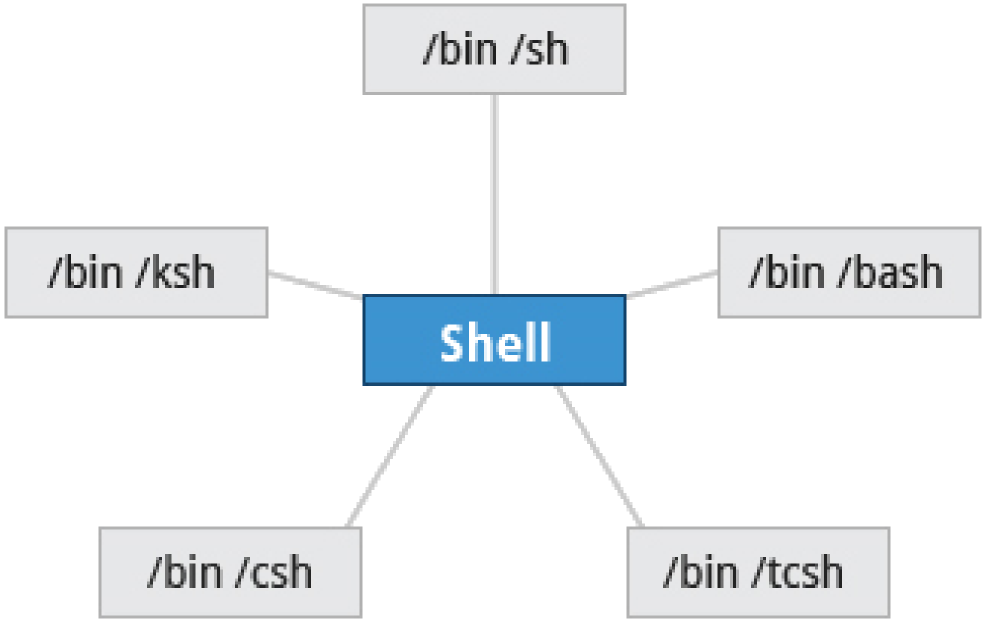
Scripts

- In order to automate sets of commands you'll need to write shell scripts, the most common of which are used with bash.


Combine long and repetitve sequences of commands into one simple command
Automate tasks and reduce risk of errors
Share procedures among several users
Features of Shell Scripts
Provide a controlled interface to users
Create new commands using combination of utilities
Quick prototyping, no need to compile

Command Shell choices

- A shell is a command line **interpreter** which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in.
- Linux provides a wide choice of shells.



Bash Shell Scripting

- The #!/bin/bash in the first line should be recognized by anyone who has developed any kind of script in UNIX environments.
- A simple bash script that displays a two-line message on the screen follow:

```
 $ cat > exscript.sh
   #!/bin/bash
   echo "HELLO"
   echo "WORLD"
```

- press ENTER and CTRL-D to save the file, or just create exscript.sh in your favorite text editor. Then, type `chmod +x exscript.sh` to make the file executable. To run it by simply typing `./exscript.sh` or by doing:

```
 $ bash exscript.sh
   HELLO
   WORLD
```

Interactive example using bash Scripts

- The user will be prompted to enter a value, which is then displayed on the screen. The value is stored in a temporary variable, sname. We can reference the value of a shell variable by using a **$** in front of the variable name, such as $sname. The script is saved in file ioscript.sh with the following content:

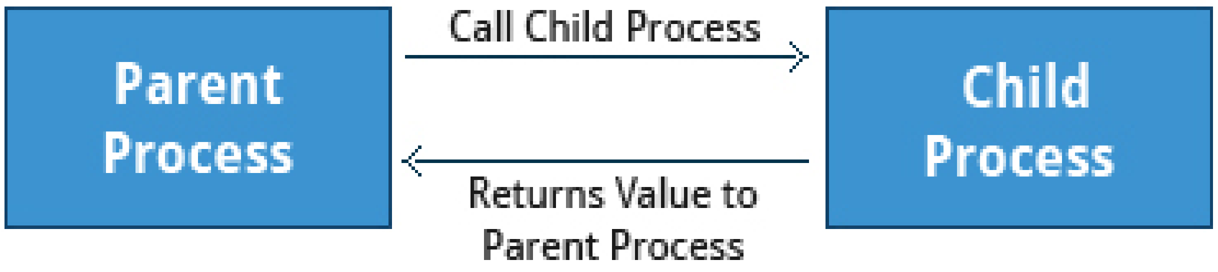
```
#!/bin/bash
# Interactive reading of variables
echo "ENTER YOUR NAME"
read sname
# Display of variable values
echo $sname
```

Interactive example using bash Scripts

- make it executable by doing `chmod +x ioscript.sh`
- to execute the script write `./ioscript.sh` is executed, the user will receive a prompt ENTER YOUR NAME. The user then needs to enter a value and press the Enter key. The value will then be printed out.
- Additional note: The hash-tag/pound-sign/number-sign (**#**) is used to start **comments** in the script and can be placed anywhere in the line (the rest of the line is considered a comment).

Return Values

- All shell scripts generate a return value upon finishing execution; the value can be set with the exit statement. Return values permit a process to monitor the exit state of another process often in a parent-child relationship. This helps to determine how this process terminated and take any appropriate steps necessary, contingent on success or failure.



Viewing Return Values

- As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value. An easy way to demonstrate success and failure completion is to execute **ls** on a file that exists and one that doesn't, as shown in the following example, where the return value is stored in the environment variable represented by **`$?`**:

```
$ ls /etc/passwd
/etc/ passwd
$ echo $?
0
```

In this example, the system is able to locate the file /etc/passwd and returns a value of 0 to indicate success; the return value is always stored in the $? environment variable.

Basic Syntax and Special Characters

- Scripts require you to follow a standard language syntax. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some **special character** usages within bash scripts:

Character	Description
#	Used to add a comment, except when used as \#, or as #! when starting a script.
\	Used at the end of a line to indicate continuation on to the next line
;	Used to interpret what follows as a new command
$	Indicates what follows is a variable

Basic Syntax examples

- When **#** is inserted at the beginning of a line of commentary, the whole line is ignored.

```
# This line will not get executed
```

- The concatenation operator (\) is used to concatenate large commands over several lines in the shell. Example: to copy the file /var/ftp/pub/userdata/custdata/read from server1.linux.com to the /opt/oradba/master/abc directory on server3.linux.co.in write the command using the \ operator as:

```
scp abc@server1.linux.com:\
/var/ftp/pub/userdata/custdata/read \
abc@server3.linux.co.in:\
/opt/oradba/master/abc/
```

Basic Syntax examples: Putting Multiple Commands on a Single Line

- The ; (semicolon) character is used to separate commands and execute them sequentially as if they had been typed on separate lines. The three commands in the following example will all execute even if the ones preceding them fail:

```
$ make ; make install ; make clean
```

- To abort subsequent commands if one fails use the **&&** (and) operator as in:

```
$ make && make install && make clean
```

- A final refinement is to use the **||** (or) operator as in:

```
$ cat file1 || cat file2 || cat file3
```

In this case, you proceed until something succeeds and then you stop executing any further steps.

Functions

- A **function** is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called **subroutines**. Using functions in scripts requires two steps:
  1. Declaring a function
  2. Calling a function
- The function declaration requires a name which is used to invoke it. The proper syntax is:

```
function_name () {
    command...
 }
```

Functions

- Example: the following function is named display:

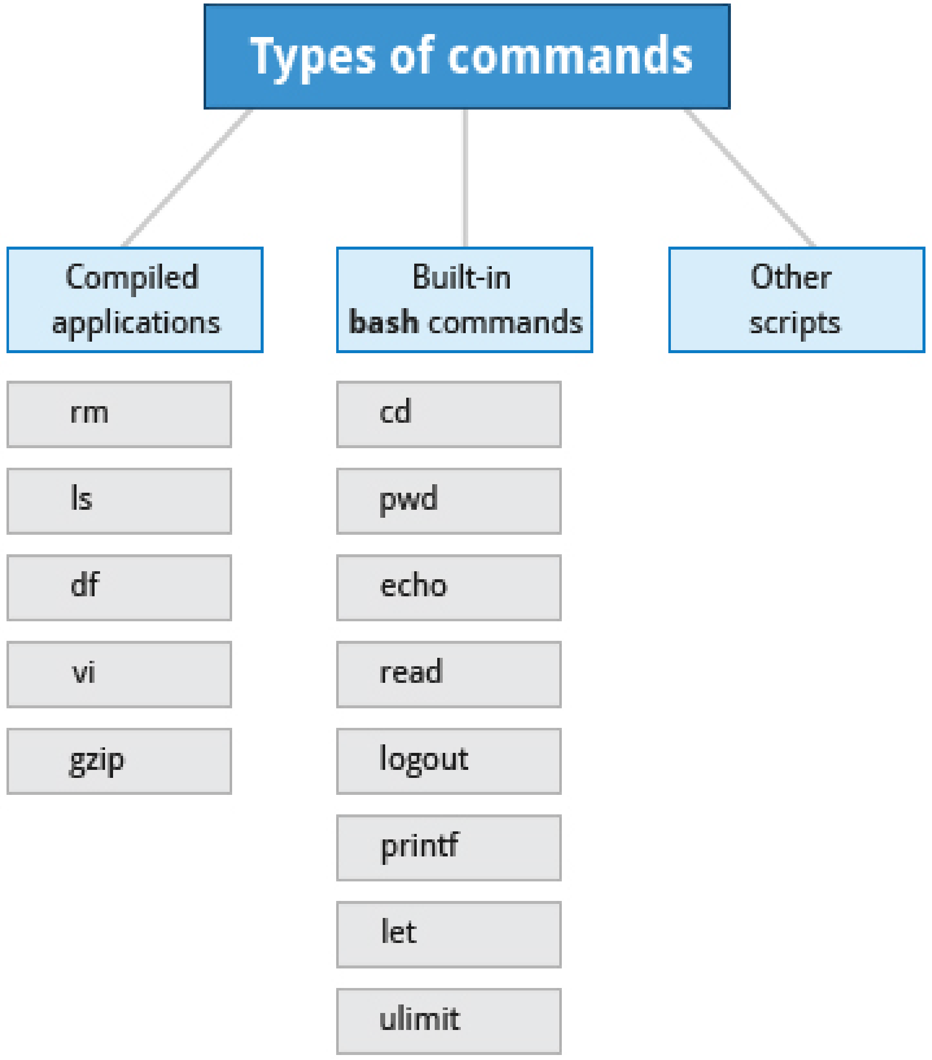
```
display () {
    echo "This is a sample function"
}
```

- The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary. We can pass an **argument** to the function. The first argument can be referred to as $1, the second as $2, etc.

Built-in Shell Commands

- **Shell scripts** are used to execute sequences of commands and other types of statements. Commands can be divided into the following categories:
  - Compiled applications: binary executable files that you can find on the filesystem (rm, ls, df, vi, and gzip)
  - Built-in bash commands: which can only be used to display the output within a terminal shell or shell script (cd, pwd, echo, read, logout, printf, let, and ulimit)
  - Other scripts
- A complete list of bash built-in commands can be found in the bash man page, or by simply typing help.

Built-in Shell Commands



Command Substitution

- If you need to **substitute the result** of a command as a portion of another command.,it can be done in two ways:
  - By enclosing the inner command with backticks (`)
  - By enclosing the inner command in $( )
- No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done.
- The $( ) method allows command nesting. New scripts should always use this more modern method. For example:

```
$ cd /lib/modules/$(uname -r)/
```

The output of the command "uname –r" becomes the argument for the cd command.

Environment Variables

- Almost all scripts use variables containing a value, which can be used anywhere in the script. These variables can either be **user** or **system** defined. Many applications use such environment variables for supplying inputs, validation, and controlling behavior.
- Some examples of standard **environment variables** are HOME, PATH, and HOST. When referenced, environment variables **must be** prefixed with the **$** symbol as in $HOME. Example: to display the value stored in the PATH variable:

```
$ echo $PATH
```

- You can get a list of environment variables with the **env**, **set**, or **printenv** commands

Exporting Variables

- By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the **export** statement as in:

```
export VAR=value
```

or

```
VAR=value ; export VAR
```

- While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

Script Parameters

- Users often need to pass parameter values to a script, such as a filename, date, etc. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers as in:

```
$ ./script.sh /tmp
$ ./script.sh 100 200
```

Script Parameters

- Within a script, the parameter or an argument is represented with a $ and a number. The table lists some of these parameters.

Parameter	Meaning
$0	Script name
$1	First parameter
$2, $3, etc.	Second, third parameter, etc.
$*	All parameters
$#	Number of arguments

Using Script Parameters

- The following script (contained in the file named **script3.sh**) :

```
#!/bin/bash
echo "The name of this program is: $0"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "The third argument is: $3"
echo "All arguments passed from cli are : $*"
echo
echo "All done with $0"
```

Using Script Parameters

- Make the script executable with **`chmod +x`**. Run the script giving it three arguments as in:

```
$./script3.sh one two three
```

and the script is processed as follows:

`$0` prints the script name: script3.sh

`$1` prints the first parameter: one

`$2` prints the second parameter: two

`$3` prints the third parameter: three

`$*` prints all parameters: one two three

The final statement becomes: All done with script3.sh

Output Redirection

- Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file (called output **redirection**).
- The **>** character is used to write output to a file. For example, the following command sends the output of free to the file /tmp/free.out:

```
$ free > /tmp/free.out
```

- To check the contents of the `/tmp/free.out` file, at the command prompt type `cat /tmp/free.out`.
- Two > characters (**>>**) will append output to a file if it exists, and act just like > if the file does not already exist.

Input Redirection

- Just as the output can be redirected to a file, the input of a command can be read from a file. The process of reading input from a file is called input redirection and uses the **<** character. If you create a file called script8.sh with the following contents:

```
#!/bin/bash
echo “Line count”
wc -l < /temp/free.out
```

- and then execute it with

```
chmod +x script8.sh ; ./script8.sh
```

it will count the number of lines from the /temp/free.out file and display the results.

The if Statement

- The actions of if depend on the evaluation of conditions:
  - Numerical or string comparisons
  - Return value of a command (0 for success)
  - File existence or permissions
- In compact form, the syntax of an if statement is:

```
if TEST-COMMANDS; then CONSEQUENT-COMMANDS;   fi
```

- A more general definition is:

```
if condition
then
     statements
else
     statements
fi
```

Using the if Statement

- The following `if` statement checks for the `/etc/passwd` file, and if the file is found it displays the message

```
/etc/passwd exists.:
if [ -f /etc/passwd ]
then
  echo "/etc/passwd exists."
fi
```

- Notice the use of the **square brackets** ([ ]) to delineate the test condition. There are many other kinds of tests you can perform, such as checking whether two numbers are equal to, greater than, or less than each other and make a decision accordingly.

Testing for Files

- bash provides a set of file conditionals, that can used with the if statement, including:

Condition	Meaning
-e file	Check if the file exists.
-d file	Check if the file is a directory.
-f file	Check if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.)
-s file	Check if the file is of non-zero size.
-g file	Check if the file has sgid set.
-u file	Check if the file has suid set.
-r file	Check if the file is readable.
-w file	Check if the file is writable.
-x file	Check if the file is executable.

- You can view the full list of file conditions using the command `man 1 test`.

Example of Testing of Strings

- You can use the `if` statement to compare strings using the operator `==` (two equal signs). The syntax is as follows:

```
if [ string1 == string2 ] ; then
    ACTION
fi
```

Numerical Tests

- You can use specially defined **operators** with the `if` statement to compare numbers. The various operators that are available are listed in the table.

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-ge	Greater than or equal to
-le	Less than or equal to

- The syntax for comparing numbers is as follows:

```
exp1 -op exp2
```

String manipulations

- A **string variable** contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Examples: abcde, 123, abcde-123, &acbde=%123
- String operators include those that do comparison, sorting, and finding the length. Example:

Operator	Meaning
[string1 > string2]	Compares the sorting order of string1 and string2.
[string1 == string2]	Compares the characters in string1 with the characters in string2.
myLen1=${#string1}	Saves the length of string1 in the variable myLen1.

Arithmetic Expressions

- Arithmetic expressions can be evaluated in the following three ways (spaces are important!):
- Using the **expr** utility: **expr** is a standard but somewhat deprecated program. The syntax is as follows:

```
expr 8 + 8
echo $(expr 8 + 8)
```

- Using the `$((...))` syntax: This is the built-in shell format. The syntax is as follows:

```
echo $((x+1))
```

- Using the built-in shell command let. The syntax is as follows:

```
let x=( 1 + 2 ); echo $x
```

Boolean Expressions

- Boolean expressions evaluate to either TRUE or FALSE, and results are obtained using the various Boolean operators listed in the table.

Operator	Operation	Meaning
&&	AND	The action will be performed only if both the conditions evaluate to true.
||	OR	The action will be performed if any one of the conditions evaluate to true.
!	NOT	The action will be performed only if the condition evaluates to false.

Tests in Boolean Expressions

- Boolean expressions return either TRUE or FALSE. We can use such expressions when working with multiple data types including strings or numbers as well as with files. For example, to check if a file exists, use the following conditional test:

```
[ -e <filename> ]
```

- Similarly, to check if the value of number1 is greater than the value of number2, use the following conditional test:

```
[ $number1 -gt $number2 ]
```

- The operator `-gt` returns TRUE if number1 is greater than number2.

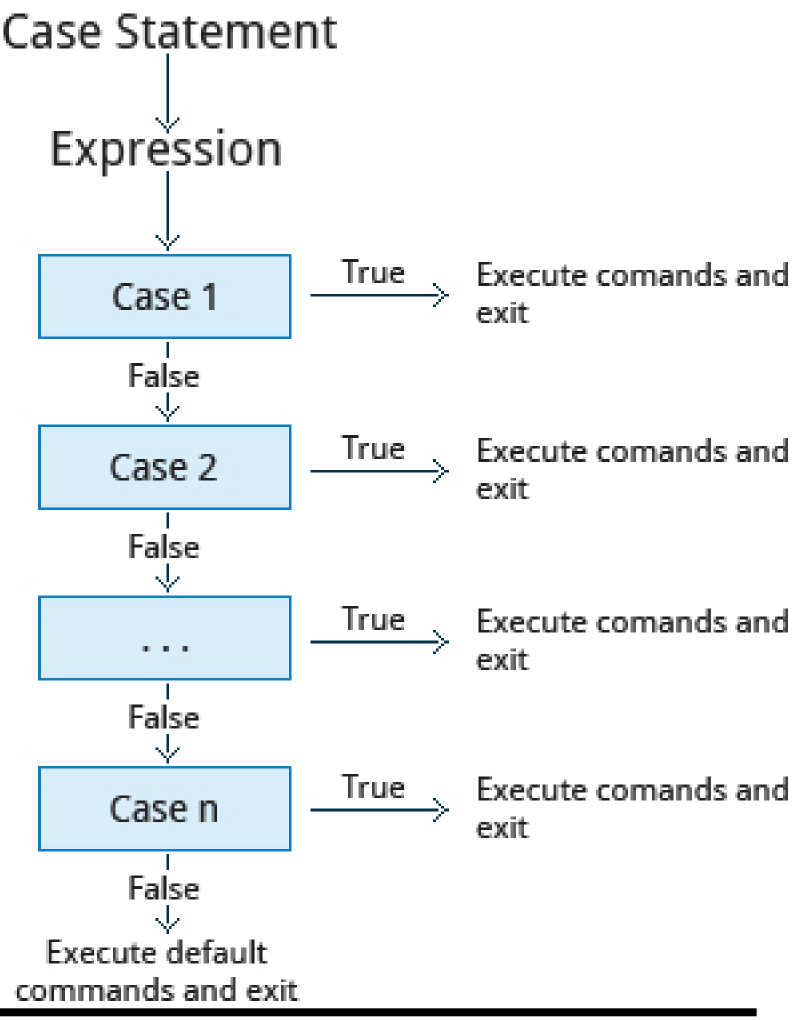
The case Statement

- The `case` statement is used in scenarios where the actual value of a variable can lead to different execution paths. case statements are often used to handle command-line options.
- The advantages of using the case statement are:
  - It is easier to read and write.
  - It is a good alternative to nested, multi-level if-then-else-fi code blocks.
  - It enables you to compare a variable against several values at once.
  - It reduces the complexity of a program.

Structure of the case Statement

- Here is the basic structure of the case statement:

```
case expression in
  pattern1) execute commands;;
  pattern2) execute commands;;
  pattern3) execute commands;;
  pattern4) execute commands;;
  *) execute some default commands \
       or nothing ;;
esac
```



case statement example

- Script with the case statement:

```
#!/bin/bash
# prompt user to enter a character
read charac
case "$charac" in
      "a"|"A") echo "You have typed a vowel!";;
      "a"|"A") echo "You have typed a vowel!";;
      "a"|"A") echo "You have typed a vowel!";;
      "a"|"A") echo "You have typed a vowel!";;
      "a"|"A") echo "You have typed a vowel!";;
      *) "You have typed a consonant";;
esac
exit 0
```

Looping Constructs

- By using looping constructs, you can execute one or more lines of code repetitively. Usually you do this until a conditional test returns either true or false as is required.
- Three type of loops are often used in most programming languages:
  - `for`
  - `while`
  - `until`
- All these loops are easily used for repeating a set of statements until the exit condition is true.

The 'for' Loop

- The for loop operates on each element of a list of items. The syntax for the for loop is:

```
for variable-name in list
do
  execute one iteration for each item in
   the list until the list is finished
done
```

- In this case, `variable-name` and `list` are substituted by you as appropriate. As with other looping constructs, the statements that are repeated should be enclosed by do and done.

The while Loop

- The while loop repeats a set of statements as long as the control command returns true. The syntax is:

```
while condition is true
do
   Commands for execution
   ----
done
```

- The set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition. Often it is enclosed within square brackets ([ ]).

The until Loop

- The until loop repeats a set of statements as long as the control command is false. Thus it is essentially the opposite of the while loop. The syntax is:

```
until condition is false
do
  Commands for execution
  ----
done
```

- Similar to the while loop, the set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition.

Redirecting Errors to File and Screen

- In UNIX/Linux, all programs that run are given three open file streams when they are started as listed in the table:

File stream	Description	File Descriptor
stdin	Standard Input, by default the keyboard/terminal for programs run from the command line	0
stdout	Standard output, by default the screen for programs run from the command line	1
stderr	Standard error, where output error messages are shown or saved	2

- Using redirection we can save the stdout and stderr output streams to one file or two separate files for later analysis after a program or command is executed.

Examples

- Practice Linux, bash
- Examples bash

Referencias

- Bash scripting cheatsheet
- Bash Reference Manual
- Advanced Bash-Scripting Guide