Shell scripting notes

Piotr Karamon

Contents

1 Comments

1.1 Normal comments

You can create comment using # symbol. Examples:

```
# this is a comment
echo "hello world" # prints hello world to the scree
```

1.2 Shebang

```
#!/bin/bash
#!/bin/sh
```

Those are some examples of shebangs. Every script should have that comment on the first line. It lets the operating system know which program should execute the script. If you omit it, it is going to default to currently running shell.

2 Variables

2.1 Setting/Getting variables

```
x=2
name="Peter"
echo "$x" "$name" "${name}"
```

We can set our variables using =. Notice that there are no spaces between = and variables name and value. To access a variable we use the prefix $. Also we can use the `${VAR_NAME}` syntax.

2.2 Special variables

name	meaning
`$0` or `${0}`	first argument passed to the script(script name)
`$n` or `${n}`	n-th argument
`${#}`	number of arguments entered (script name does not count)
`$@` or `${@}`	all arguments passed

You can think of `$@` as an array of all the arguments. And `$n` syntax as a way of indexing into that array.

This is prints every argument of its own line

```
for arg in $@; do
   echo "arg" $arg
done
```

2.3 Single quotes vs double quotes

```
name="bob"
echo "$name" '$name'
```

```
bob $name
```

Notice the difference. When we use single quotes we make sure that the text between them is "treated as is". Meaning the interpreter will not try to find and insert variables in to it. We have two general rules concerning quotes:

- single quotes treat text as literal
- double quotes expand everything they can

3 Capturing output of commands

In order to do that we have to use the `$(<COMMAND>)` structure.

The following snippet stores the number of lines in `.bashrc` file into `nbashrcLines` variable and then prints the result.

```
nbashrcLines=$(cat ~/.bashrc | wc -l)
echo "your .bashrc has $nbashrcLines lines"
```

4 Conditionals

4.1 if

```
x=18
if [ "$x" -eq 12 ]; then
    echo "x is equal to 12"
elif [ "$x" -eq 18 ]; then
     echo "x is equal to 18"
else
    echo "x is not equal to 12 and 18"
fi
```

```
x is equal to 18
```

Notice the spaces between `[` and `]` they must be there. `then` keyword can be placed on a separate line or on the same one as `if` provided that we include `;`.

4.2 case

The `case` statement is very useful when a certain variable can have many possible values.

```
command="start-process-3"
case "$command" in
    start)
        echo "we are starting"
        ;;
    end)
        echo "we are ending"
        ;;
    status|state)
        echo "current state: "
        ;;
    start-process-[0-9])
        echo "starting desired process"
        ;;
    *)
      echo "not a valid command"
      ;;
esac
```

```
starting desired process
```

4.3 Checking if something is true

4.3.1 File operators:

operator	Is true if
`-a FILE`	file exists.
`-b FILE`	file is block special.
`-c FILE`	file is character special.
`-d FILE`	file is a directory.
`-e FILE`	file exists.
`-f FILE`	file exists and is a regular file.
`-g FILE`	file is set-group-id.
`-h FILE`	file is a symbolic link.
`-L FILE`	file is a symbolic link.
`-k FILE`	file has its ‘sticky’ bit set.
`-p FILE`	file is a named pipe.
`-r FILE`	file is readable by you.
`-s FILE`	file exists and is not empty.
`-S FILE`	file is a socket.
`-t FD`	FD is opened on a terminal.
`-u FILE`	the file is set-user-id.
`-w FILE`	the file is writable by you.
`-x FILE`	the file is executable by you.
`-O FILE`	the file is effectively owned by you.
`-G FILE`	the file is effectively owned by your group.
`-N FILE`	the file has been modified since it was last read.
`FILE1 -nt FILE2`	file1 is newer than file2 (according to modification date).
`FILE1 -ot FILE2`	file1 is older than file2.
`FILE1 -ef FILE2`	file1 is a hard link to file2.

4.3.2 String operators:

opeartor	Is true if
`-z STRING`	string is empty.
`-n STRING`	string is not empty.
`STRING1 = STRING2`	the strings are equal.
`STRING1 != STRING2`	the strings are not equal.
`STRING1 < STRING2`	STRING1 sorts before STRING2 lexicographical.
`STRING1 > STRING2`	STRING1 sorts after STRING2 lexicographical.

4.3.3 Comparing numbers

We compare using `[ n1 OP n1 ]` where `OP` is one of the operators in the following table

operator	meaning
`-eq`	`==`
`-ne`	`!=`
`-gt`	`>`
`-ge`	`>=`
`-lt`	`<`
`-le`	`<=`

4.3.4 Other operators:

operator	Is true if
`-o OPTION`	the shell option OPTION is enabled.
`-v VAR`	the shell variable VAR is set.
`-R VAR`	the shell variable VAR is set and is a name reference.
`!EXPR`	expr is false.
`EXPR1 -a EXPR2`	both expr1 AND expr2 are true.
`EXPR1 -o EXPR2`	either expr1 OR expr2 is true.
`arg1 OP arg2`	Numbers Arithmetic tests. OP is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, or `-ge`.

5 Integer arithmetic

Double parentheses `((expr))` allow us to use all the usual arithmetic you may know from other programming languages. `((expr))` does not produce an output. We use it to update variables. If you want to **get the output** of arithmetic operation use `$((expr))` syntax.

```
n=3

$((n+=1))
echo $n # 4

$((n*=2))
echo $n # 8

$((n-=2))
echo $n # 6

$((n/=3))
```

```
echo $n # 2

$((n=5*4*3*2*1))
echo $n # 120

i=$(( 20/3 )) # we can assign a variable
echo $i # 6, notice the lack of fractional part
```

6 For loops

```
for name in Bob Tom Jim; do
    echo $name
done
```

```
Bob
Tom
Jim
```

The we want the `do` keyword to be on the same line as `for` we **must** include the `;`;

```
for file in $(ls /home); do
    echo "$file"
done
```

```
piotr
```

7 while loops

```
n=0
while [ $n -lt 5 ]; do
    echo "$n"
    n=$((n+1))
done
```

```
0
1
2
3
4
```

8 Getting input

The most basic way of getting users input is `read`. You can pass a prompt text using the `-p` flag.

```
read -p "tell me your name: " USERNAME
```

This snippet of code lets the user enter his name which is then put into `USERNAME` variable.

The `-s` flag makes the input silent. It is meant for entering passwords and other sensitive information.

The read command can allow your program to accept any data from `STDOUT`. This means you can pipe output of another program/script into yours.

```
while read line; do
    # execute some code for every line of input
done
```

In Unix/Linux a lot of programs deal with lines. The above snippet of code allows you to accept any amount of them and do some processing. Doing something like `ls -l | ./path_to_your_script` means the output of `ls -l` (which lists all the files in the current directory with some information) will be piped into your script. Then for example the `while` from above can work and process this information on a line by line basis.

9 Functions

9.1 Basic function syntax

```
sayhello() {
    echo "hello bash function"
}
sayhello
```

```
hello bash function
```

If you are using bash this syntax is also allowed

```
function sayhello {
    echo "hello i am a function"
}
sayhello
```

```
hello i am a function
```

9.2 Parameters

You can access parameters in the same a script access its parameters.

```
argsexample() {
    echo "arg1=$1"
    echo "arg2=$2"
    echo "all arguments=$@"
    echo "how many arguments=$#"
}
argsexample 123 "Bob"
```

```
arg1=123
arg2=Bob
all arguments=123 Bob
how many arguments=2
```

9.3 Variables inside of a function

```
varsexample() {
    A="hello"
    local B="world"
}
varsexample
echo "value of A:" $A
echo "value of B:" $B
```

```
value of A: hello
value of B:
```

As you can see if a variable has `local` keyword in front of it it will be available only in the function in which it was declared. If you don't put the `local` keyword the variable will be global to the whole script.

9.4 Exit code of functions

Functions return exit codes as usual program/scripts do. You can explicitly return a status code by using `return <STATUS-CODE>`. Then after a call to a function you can capture its exit code by using `${?}`

10 Getopts

Often you want to alter the behavior of your script based on some input. A very common way of customizing the behavior is the use of options. For example `ls -al` in this case we are passing two options to the ls program. `-a` means we want to see hidden files and `l` means we want to get more detail. You can get those options in your script by using `getopts` command.

10.1 How it works?

Lets say we want to accept:

- `l` an argument meaning length.
- `u` an argument tells us if the output ought to be uppercase

We want to call our script in the following manner

```
./path-to-script -l 10 -u
```

A call to `getopts` looks like this:

```
getopts l:u OPTION
```

`l:u` means we want to accept two options, one: `l`, the second: `u`. The `:` symbol means that `l` accepts a value. `getopts` will then set `OPTION` variable to current option. And `OPTARG` variable to additional value. That behaviour means that we usually call `getopts` using a `while` loop. An example will explain:

```
while getopts vl:s OPTION; do
    case ${OPTION} in
    u)
        UPPERCASE='true'
        echo "upper-casing is on"
        ;;
    l)
        LENGTH="${OPTING}"
        echo "using length: $LENGTH"
        ;;
    ?)
        echo "$OPTION is an invalid option"
        ;;
    esac
done
```

11 Combining commands

11.1 The ; operator

The ; can be used to run multiple commands on the same line

```
echo "hello"; echo "world"
```

```
hello
world
```

11.2 The && operator

&& runs the second command **only if** the first one succeeds. If the first command fails (meaning it encountered some kind of an error) the second command will not be executed.

Notice: the `file.txt` does not exist. Meaning `cat` command returns an error if we try to pass this file to it.

```
cat file.txt && echo "file.txt exists"
```

The `echo` command will not run because `cat` fails.

11.3 The || operator

|| means or. It will run the second command **only if** the first one fails. Meaning in this example:

```
cat .bashrc || cat .zshrc
```

we will either display `.bashrc` file or `.zshrc` or none (both files may not exist). We will never display both of them.

11.4 Operators vs if-s

Many `if` statements can be transformed using operators: `;`, `&&`, `||`. They provide more concise syntax and are especially useful when you're working inside a shell(they are a lot easier to write than `if` statments).

12 Find command

The `find` command is used to search for files and directories. It can also be used with conjunction with other commands to perform a large variety of tasks on files. The general form a `find` command looks as follows

```
find <DIRECTORY_PATH> <SELECTION> <ACTION>
```

The `find` command is very powerful. But in most cases you pass a directory to it, which indicates from where it should start searching. Then you provide some information about what you're looking for (filenames, file types, modification dates, ownership etc.) .

12.1 Global options

name	description
`-regextype`	controls how regexes are processed
`-d or --depth`	process directory's contents before the directory itself
`-maxdepth <levels>`	descend at most <levels> of directories below the starting point
`-mindepth <levels>`	do not search at levels less then <levels>

12.2 Tests

Numeric argument `n` can be specified for many tests(`-amin -mtime -gid -inum -links -size -uid`). It is used as follows:

- +n for greater than n
- -n for less than n
- n for exactly n

Test name, arguments	description
`-name <pattern>`	base of file name(the path with leading directories removed) matches shell pattern `<pattern>`. Always enclose your `<pattern>` in quotes
`-iname <pattern>`	like -name but the match is case insensitive
`-path <pattern>`	File name matches shell pattern `<pattern>`. The meta-characters do not
`-ipath <pattern>`	like -path but case insensitive
`-type <c>`	File is of type `<c>`: - f regular file - d directory - p named pipe (FIFO) - l symbolic link - socket - b block(buffered special) - c character(unbuffered special)
`-size <n>[cwbkMG]`	files uses less than, more than or exactly `<n>` units of space, rounding up, - b for 512-byte blocks - c for bytes - w for two-byte words - k for KiB(1024 bytes) - M for MiB(1024*1024 bytes = 1024*1KiB) - G for GiB(1024*1MiB)
`-empty`	File is empty and is either a regular file or a directory
`-regex`	file names matches regular expression pattern. This is a match on **the whole path** not a search. Default regexes are Emacs Regular Expressions
`-amin <n>`	file was last accessed less than, more than or exactly `<n>` minutes ago
`-anewer <reference>`	Time of the last access of the current file is more recent than that of the last data modification of the `<reference>` file.
`-atime <n>`	File was last accessed less than, more than or exactly `<n>*24` hours ago
`-cmin <n>`	file's status was last changed less than, more than or exactly `<n>` minutes ago
`-cnewer <reference>`	look -anewer and -cmin
`-ctime`	look -atime and -cmin
`-executable`	executables and searchable directories
`-group <gname>`	file belongs to group name `<gname>`
`-mmin <n>`	file's data was last modified less than , more than or exactly `<n>` minutes ago following meta-characters are are recognized: (`*, ?, []`).
`-newer <reference>`	Time of the last data modification of the current file is more recent that that of the last data modification of the `<reference>` file. treat `/ or .` specially. `-path` does not use absolute paths it instead starts from the directory you specified.
`-perm <mode>`	`<mode>` can be octal or symbolic.
`-readable`	matches files which are readable by the current user
`-user <uname>`	File is owned by user `<uname>`
`-writable`	Files which are writable by the current user

12.3 Actions

Action	Description
`-delete`	deletes the files, it automatically applies the `-depth` option
`-exec <command> \;`	execute command; true if 0 status is returned. The string `{}` is replaced by the current file path
`-print`	print the current file name

12.4 Operators

operator	description
`!expr`	true if `expr` is false
`(expr)`	force precedence
`expr1 expr2`	true if both expressions are true
`expr1 -o expr2`	true if either of expressions is true

13 Cronjobs

Cron jobs are tasks that you want to be run periodically. Meaning you don't want to execute certain programs/scripts yourself. Cron jobs might include tasks such as updates, cleaning useless data/catches, or things such as changing a wallpaper every now and then. Meaning cronjobs are useful not only for servers but also for typical Linux users. In order to write cronjobs you need a cronjob manager. For Arch Linux is cronie. You can install and enable it by saying:

```
sudo pacman -S cronie
sudo systemctl enable cronie
sudo systemctl start cronie
```

Write `crontab -e` to write cronjobs. It is really important to know who to specify time at which those jobs ought to be run. `MIN HOUR DAY MONTH DOW`

- MIN - 1-60
- HOUR - 1-24
- DAY - 1-31
- MONTH - 1-12
- DOW - 0-6 (0-Sunday 6-Saturday)

Some examples

time specifier	description
`* * * * *`	every minute
`*/15 * * * *`	every fifteen minutes
`*/3 * * * *`	every three minutes
`*/15 * * * 1,2,3,4,5`	every fifteen minutes of Mon-Friday
`*/15 * * * 1-5`	the same as the above

14 File permission/Access Modes

14.1 Permissions overview

Every file in Unix has the following attributes:

- owner permissions - they determine what actions the owner of the file can perform
- group permissions - they determine what actions a user, who is a member of the group that a file belongs to, can perform on the file
- other (world permissions) - the permissions for others indicate what all other users can perform on the file

You can view the permissions on a file by using

```
ls -l <file-path>
```

Permissions are specified on the first column. The output usually looks like this

```
-rwxr-xr-x  1 peter root     226 06-22 01:24
↪  switch_to_monitor
drwxr-xr-x  3 peter peter   4096 06-26 12:07 vimwiki
```

The first character indicate if something is a file or a directory.

- `-` means regular file
- `d` means directory

The three characters(2-4) represent the permissions for the file's owner. `-rwxr-xr-x` means that the owner has read(r), write(w), execute(x) permissions.

The second group of three characters (5-7) contains permissions for the group which the file belongs to. For example `-rwxr-xr-x` means that the group can read(r) and execute(e) the file. But it cannot write to it.

The last group indicates permissions for everyone else. For example `-rwxr-xr--` means that there is only read(r) permission.

14.2 File access modes

14.2.1 Read

Grants the capability to read, i.e., view the contents of the file.

14.2.2 Write

Grants the capability to modify, or remove the content of the file.

14.2.3 Execute

User with execute permissions can run a file as a program.

14.3 Directory access modes

14.3.1 Read

User can look at the content of the directory(view the filenames inside it).

14.3.2 Write

User can add or delete files from the directory.

14.3.3 Execute

Executing a directory does not make sense. It's really a traverse permission. A user must have execute access to the `bin` directory in order to execute the `ls` or the `cd` commands.

14.4 Changing permissions

To change the file or directory permissions you use the `chmod` command. There are two ways of using the `chmod` command.

14.4.1 Symbolic Mode

It's easy and user friendly. There are three operators.

- `+` adds the designated permissions
- `-` removes teh designated permissions
- `=` sets the designated permissions

You can prefix those operators in order to specify which permission group you want to change.

- `u` user who owns the file
- `g` group which owns the file
- `o` other users not in file's group
- `a` all users(default)

Examples

```
# adds read, write, execute permissions for the owner
chmod u+rwx script.sh
# adds execute permission for everyone
chmod +x script.sh
# removes read and write permissions
# from everyone who is not the owner
chmod go-rw script.sh
```

14.4.2 Absolute Mode

In this mode you have to specify **all** permissions at once

Number	Binary	Ref	Octal Permission Representation
0	000	`---`	No permission
1	001	`--x`	Execute permission
2	010	`-w-`	Write permission
3	011	`-wx`	Execute and write permission: 1 (execute) + 2 (write) = 3
4	100	`r--`	Read permission
5	101	`r-x`	Read and execute permission: 4 (read) + 1 (execute) = 5
6	110	`rw-`	Read and write permission: 4 (read) + 2 (write) = 6
7	111	`rwx`	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7

You actually pass only one number in base 8.

```
# give all permissions to the owner
# give read, execute permissions for everyone else
chmod 755 testfile
```

14.5 Changing file owner

The change the owner of the file we use the `chown` command. We use it in the following way.

```
chown bob file1.txt pictures
```

This will make bob(user) the owner of `file1.txt` file and `pictures` directory.

`root` has unrestricted access when it comes to `chown` command. Normal users however can only change the ownership of their own files(they are the owners of them).

14.6 Changing group ownership

The `chgrp` command changes the group ownership of a file.

```
chown classroom file1.txt pictures
```

This will change make the group `classroom` own the `file1.txt` and `pictures`.

14.7 SUID and SGID

Often when a command is executed, it will have to be executed with special prviileges in order to accomplish its task.

For example the `passwd` command allows you to change your own password. The new password is stored in the file `/etc/shadow`. As a regular user you don't have read or write access to this file for security reasons. But in order to change your password you must write to this file. Additional permissions are given to programs via a mechanism known as the `Set User ID (SUID)` and `SET GROUP ID (SGID)` bits.

When you execute a program that has the `SUID` bit enabled, you **inherit the permissions of that program's owner.** Programs which do not have this bit enabled are run with the permissions of the user who started the program.

This is the case with `SGID` as well. Normally, programs execute with your group permissions, but if the `SGID` is set then your group will be changed for the group owner of the program.

You can see if `SUID` or `SGID` set by using `ls -l`. If those bits are set then instead of the typical `x` for execute permission you will see the letter `s`.

```
-rwsr-xr-x 1 root root 51464 01-27 14:47 /bin/passwd
```

15 File descriptor

15.1 Introduction

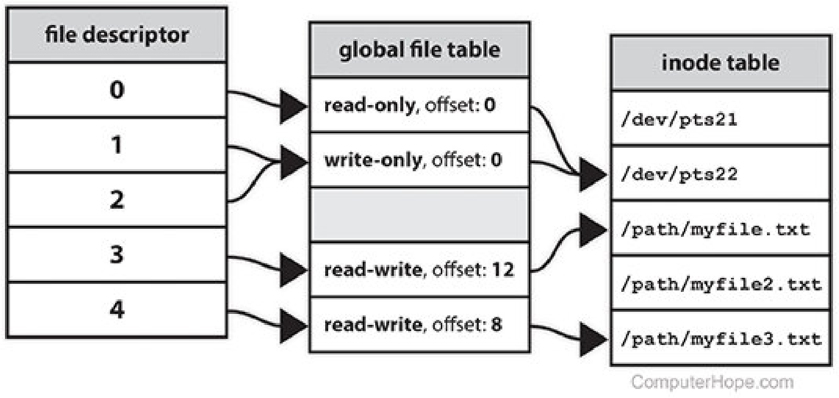
A file descriptor is **a number that uniquely identifies an open file** in a computer's operating system. That number is a **unique non-negative integer**. At least one file descriptor exists for every open file on the system.

File descriptors were first used in Unix, and are used by modern operating systems including Linux, MacOS, BSD. In Windows, file descriptors are knows as file handles.

When a program asks to open a file - or another data resource, like a network socket the kernel:

1. Grants access
2. Creates an entry in the global file table
3. Provides the software with location of that entry.

15.2 Overview



When a process makes a successful request to open a file, the kernel returns a file descriptor which points to an entry in the kernel's global file table. The file table entry contains information such as the inode of the file, byte offset, and the access restrictions for that data stream(read-only, write-only, etc.)

Inode is for short for index node, it is information contained within a Unix system with details about each file, such as the node, owner, file, location of file, etc.

15.3 Default file descriptors

On a Unix-like operating system, the first three file descriptors, by default are:

15.3.1 Standard Input

- abbreviation: stdin
- file descriptor: 0
- description: the default data stream for input, for example, in a command pipeline. The the terminal, this defaults to keyboard input from the user.

15.3.2 Standard Output

- abbreviation: stdout
- file descriptor: 1

- description: the default data stream for output for example when a command prints text. In the terminal, this defaults to the user's screen.

15.3.3 Standard Error

- abbreviation: stderr
- file descriptor: 2
- description: the default data stream for output that relates to an error occuring. In the terminal this defaults ot user's screen.

16 Using file descriptors(redirection)

Name	Description
0	standard input (STDIN)
1	standard output (STDOUT)
2	standard error (STDERR)
&	STDOUT and STDERR

For example, by default all your echo calls print to STDOUT. You can override that behavior by using redirects.

```
echo "hello world" > file.txt
```

This snippet of code will print "hello world" to file.txt. **It will override anything that might be in this file**. If you want to append text to a file use >> instead.

`>` is used to redirect STDOUT

`1>` the same as >

`2>` redirect STDERR

`<` redirect STDIN

Those file descriptors are located in `/dev` folder.

- `/dev/stdin`
- `/dev/stdout`
- `/dev/stderr`

There is a special file `/dev/null` if you redirect to it, it will discard everything that was passed. `program &> /dev/null` discards any output that the program might have produced.

If you want to pipe both `STDERR` and `STDOUT` you can do that using `|&` operator.