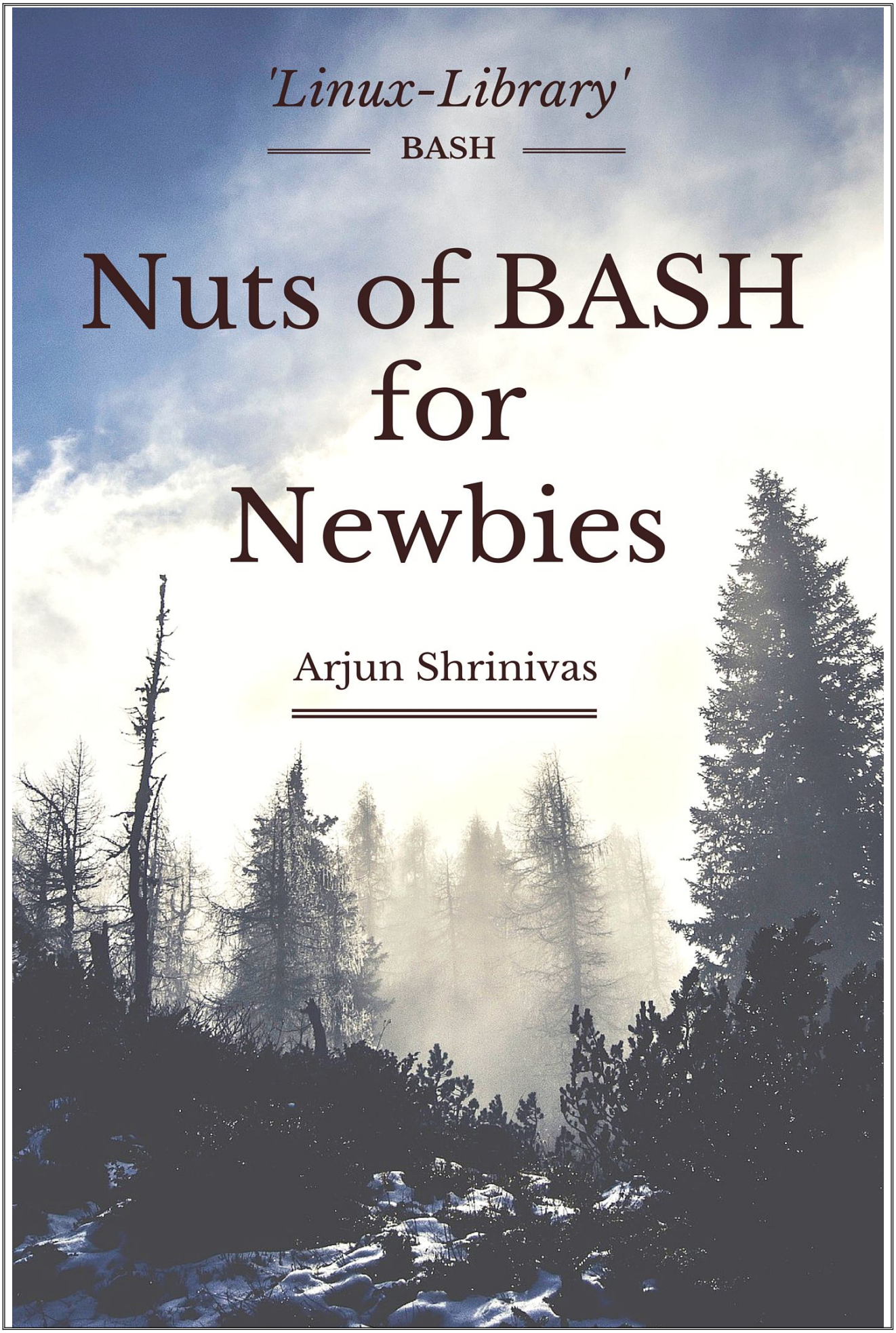
'Linux-Library'
BASH
Nuts of BASH for Newbies
Arjun Shrinivas

Nuts of Bash

For

Extreme Beginners

Author : Arjun Shrinivas
www.linux-library.in

Contents

1
Bash Shell Commands (Built-ins)

In computing, a shell builtin is a command or a function, called from a shell, that is executed directly in the shell itself, instead of an external executable program which the shell would load and execute.

Shell builtins work significantly faster than external programs, because there is no program loading overhead. However, their code is inherently present in the shell, and thus modifying or updating them requires modifications to the shell. Therefore, shell builtins are usually used for simple, almost trivial, functions, such as text output. Because of the nature of some operating systems, some functions of the systems must necessarily be implemented as shell builtins. The most notable example is the cd command, which changes the working directory of the shell. Since each executable program runs in a separate process, and working directories are specific to each process, loading cd as an external program would not affect the working directory of the shell that loaded it.

- **pwd**

This is a Shell built-in command. It can be found in most of the Unix Shells like sh, bash, ash, ksh, zsh. This command is used to know where are we at current in the file system, in the sense in which directory are we at current. If we input at the shell the output will be the directory at where we are.

```
$ pwd
/home/arjun
```

- **test**

It is a command-line utility found in Unix-like operating systems that evaluates conditional expressions. We can use this in various conditions like if, while. Syntax of test command is like below

```
$ test    <expression>
      Or
$ <expression>
```

- **type**

'type' is a unix built-in command to know about other commands. If you are checking an alias then you might also know for which command it is an alias of.

```
$ type ll
ll is aliased to `ls -alF'
```

- **file**

'file' command is used to check the type of a file.

```
$ file test.sh
test.sh: POSIX shell script, ASCII text executable
```

- **whereis**

'whereis' is a Unix command used to locate some special files of a Unix command like the binary, source and manual page files. Unix type is usually preferred to locate a binary file, because it is POSIX and it can identify shell aliases.

```
# whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
# whereis init
init: /sbin/init /etc/init.d /etc/init /lib/init /usr/share/man/man8/init.8.gz /usr/share/man/man5/init.5.gz
```

- **which**

'which' is a Unix command to know the path of any binary.

```
# which bash
/bin/bash
# which python
/usr/bin/python
# which init
/sbin/init
```

We have so many other commands like cd, help, command, alias, unalias, etc..

2
Shell Tests

In general we use the 'test' command to perform the tests. We have learned a little bit about the 'test' command in the previous chapter(Built-in commands).

This command allows you to do various tests and sets its exit code to 0 (TRUE) or 1 (FALSE) whenever such a test succeeds or not. Using this exit code, it's possible to let Bash react on the result of such a test.

In the second form of the command, the [ ] (brackets) must be surrounded by blank spaces. This is because [is a program and POSIX compatible shells require a space between the program name and its arguments. One must test explicitly for file names in the C shell. File-name substitution (globbing) causes the shell script to exit.

Below are some of the flags which we can use along with 'test' while running through the scripts / programs using conditions. Exits with a status of 0 (true) or 1 (false) depending on the evaluation of EXPR. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators as well, and numeric comparison operators.

File operators:

-a FILE	True if file exists.
-b FILE	True if file is block special.
-c FILE	True if file is character special.
-d FILE	True if file is a directory.
-e FILE	True if file exists.
-f FILE	True if file exists and is a regular file.
-g FILE	True if file is set-group-id.
-h FILE	True if file is a symbolic link.
-L FILE	True if file is a symbolic link.
-k FILE	True if file has its `sticky' bit set.
-p FILE	True if file is a named pipe.
-r FILE	True if file is readable by you.
-s FILE	True if file exists and is not empty.

-S FILE True if file is a socket.
-t FD True if FD is opened on a terminal.
-u FILE True if the file is set-user-id.
-w FILE True if the file is writable by you.
-x FILE True if the file is executable by you.
-O FILE True if the file is effectively owned by you.
-G FILE True if the file is effectively owned by your group.
-N FILE True if the file has been modified since it was last read.

FILE1 -nt FILE2 True if file1 is newer than file2 .
FILE1 -ot FILE2 True if file1 is older than file2.
FILE1 -ef FILE2 True if file1 is a hard link to file2.

String operators:

-z STRING True if string is empty.
-n STRING True if string is not empty.
STRING1 = STRING2 True if the strings are equal.
STRING1 != STRING2 True if the strings are not equal.
STRING1 < STRING2 True if STRING1 sorts before STRING2 lexicographically.
STRING1 > STRING2 True if STRING1 sorts after STRING2 lexicographically.

Other operators:
-o OPTION True if the shell option OPTION is enabled.
! EXPR True if expr is false.
EXPR1 -a EXPR2 True if both expr1 AND expr2 are true.
EXPR1 -o EXPR2 True if either expr1 OR expr2 is true.

Arithmetic tests / Number Arguments:
arg1 OP arg2
OP is one of -eq, -ne, -lt, -le, -gt, or -ge.
Arithmetic binary operators return true if ARG1 is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than ARG2.

Exit Status:
Returns success if EXPR evaluates to true; fails if EXPR evaluates to false or an invalid argument is given.

Now let us see some examples of shell tests.

Ex 1: Check for the existence of the /etc/passwd file.

```
if [ -f /etc/passwd ]
then
    echo “Hey, we found the passwd file!”
else
    Echo “Sorry buddy, I’m unable to find the passwd file!”
fi
```

Ex 2: Check for the existence of arguments of a script.

```
if [ $# == 0 ]
then
    echo “Sorry, i need at least one argument”
fi
```

Ex 3: Check for the existence of a CDROM file. If exists mount it.

```
if [ -b /dev/sr0 ]
then
    echo “Mounting ‘sr0’ on ‘/mnt’”
    mount -t iso9660 /dev/sr0 /mnt
else
    echo “Sorry, I am unable to find /dev/sr0”
fi
```

3
Exit Status

The exit status of an executed command is the value returned by the waitpid system call or equivalent function. Exit statuses fall between 0 and 255, though, as explained below, the shell may use values above 125 specially. Exit statuses from shell builtins and compound commands are also limited to this range. Under certain circumstances, the shell will use special values to indicate specific failure modes.

For the shell's purposes, a command which exits with a zero exit status has succeeded. A non-zero exit status indicates failure. This seemingly counter-intuitive scheme is used so there is one well-defined way to indicate success and a variety of ways to indicate various failure modes. When a command terminates on a fatal signal whose number is N, Bash uses the value 128+N as the exit status.

If a command is not found, the child process created to execute it returns a status of 127. If a command is found but is not executable, the return status is 126.

If a command fails because of an error during expansion or redirection, the exit status is greater than zero.

The exit status is used by the Bash conditional commands and some of the list constructs.

All of the Bash builtins return an exit status of zero if they succeed and a non-zero status on failure, so they may be used by the conditional and list constructs. All builtins return an exit status of 2 to indicate incorrect usage.

Exit Code Number	Meaning	Example	Comments
1	Catchall for general errors	let "var1 = 1/0"	Miscellaneous errors, such as "divide by zero" and other impermissible operations
2	Misuse of shell builtins (according to	empty_function() {}	Missing keyword or command, or permission problem

Exit Code Number	Meaning	Example	Comments
	Bash documentation)		(and diff return code on a failed binary file comparison).
126	Command invoked cannot execute	/dev/null	Permission problem or command is not an executable
127	"command not found"	illegal_command	Possible problem with $PATH or a typo
128	Invalid argument to exit	exit 3.14159	exit takes only integer args in the range 0 - 255 (see first footnote)
128+n	Fatal error signal "n"	kill -9 $PPID of script	**$?** returns 137 (128 + 9)
130	Script terminated by Control-C	*Ctrl-C*	Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255*	Exit status out of range	exit -1	exit takes only integer args in the range 0 - 255

According to the above table, exit codes 1 - 2, 126 - 165, and 255 have special meanings, and should therefore be avoided for user-specified exit parameters. Ending a script with exit 127 would certainly cause confusion when troubleshooting (is the error code a "command not found" or a user-defined one?). However, many scripts use an exit 1 as a general bailout-upon-error. Since exit code 1 signifies so many possible errors, it is not particularly useful in debugging.

Let us see some examples handling the exit status.

1. Verify the existence of “/tmp” directory. If we found it let us clean it up.

```
cd /tmp
if [ "$?" = "0" ]; then
    rm *
else
    echo "Cannot change directory!" 1>&2
    exit 1
fi
```

1 is the standard output (stdout).
2 is the standard error (stderr).
> is for standard redirection

& indicates that what follows is a file descriptor and not a filename


Let us meet again
I hope you have enjoyed diving into the programming world.
Keep on programming to have a great joy.
To stay updated please visit our website and our blog.
Stay Tuned
www.linux-library.in
linux-library.blogspot.in
www.facebook.com/thechamberoflinuxgeeks