

Shell Scripting

Creating a basic script

- Start vim, enable line numbers, and enter insert mode.
- Type:

```
#!/bin/bash
echo " to display info about your Linux system"
uname -a
```

echo -n does not output a new line.

- Save the file and name it "name.sh"
- Type: **chmod u+x name.sh** to make the file executable.
- To run the script type: **./name.sh**

Working with variables

Shell scripting | Variables

- ◊ **Variable:** placeholder for data.
- ◊ **Environment variable:** is a placeholder for data that can change; typically, it gets its value automatically from the OS startup or the shell being used.
- ◊ Each user has environment variables with different values to define his or her working environment.
- ◊ The **HOME** environment variable stores the absolute pathname to a user's home directory, so it varies for each user.
- ◊ Some environment variables are the same for all users logged in to a machine, such as the **HOST** environment variable that specifies the computer name.
- ◊ The **env** command allows you to see all environment variables
- ◊ You can use the echo command to see the value of an environment variable.
 - Example:
 - ◊ echo \$HOME
 - ◊ echo \$HOST

- A **shell variable** is similar to an environment variable, but its value is usually assigned in a shell script.
- Shell variables can be created in **two ways**:

Direct Assignment:

```
color = blue
```

The Prompt MEthod:

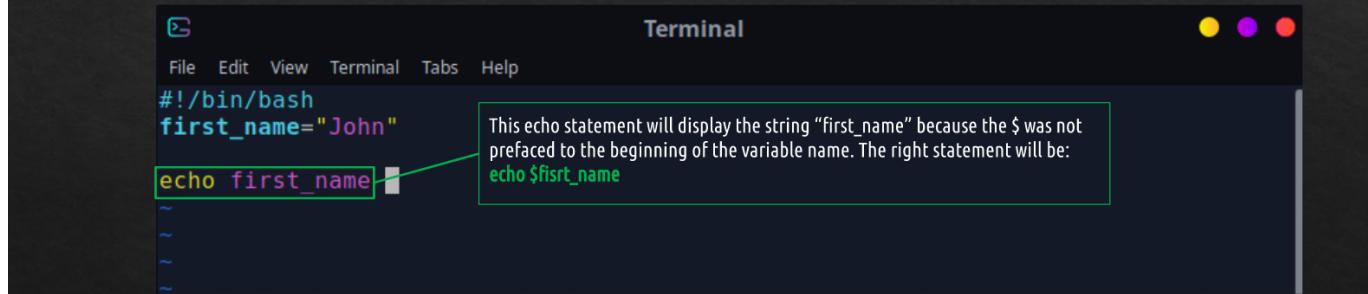
```
echo "Enter a color:";read color
```

- ◊ The **positional parameter method** uses the order of arguments in a command to assign values to variables on the command line.
- ◊ Variables from \$0 to \$9 are available, and their values are defined by what the user enters.

Table 5-6 Positional parameters

Positional parameter	Description	Example
\$0	Represents the name of the script	./scr4 (./scr4 is position 0)
\$1 to \$9	\$1 represents the first argument, \$2 represents the second argument, and so on	./scr4 /home (./scr4 is position 0 and /home is position 1) ./scr4 /home scr1 (./scr4 is position 0, /home is position 1, and scr1 is position 2)
\$*	Represents all the positional parameters except 0	/home scr1 (just /home and scr1)
\$#	Represents the number of arguments that have a value	./scr4 /home scr1 echo \$# (\$* represents positions 1 and 2, which are /home and scr1)

Notice that regardless of the method you use for assigning value to a variable, you always need the \$ in front of it if you want to use it.



```
#!/bin/bash
first_name="John"
echo first_name
```

This echo statement will display the string "first_name" because the \$ was not prefaced to the beginning of the variable name. The right statement will be:
echo \$first_name

- You can use **curly braces** to reference a variable's value: \${variable_name}
- It is useful if you want to **append a string to a variable**.
- **Command Substitution:** allows the output of a command to replace the command itself.

Can be done in **TWO WAYS**:

```
$ (command)
`command`
```

Shell scripting | Exit Status Codes

Exit status code: a number sent to the shell when you run a command.

Type:

```
#!/bin/bash
cd baddir
echo $?
```

- Successful commands usually return the **code 0**, and failures return a value **greater than 0**.
- to see an exit status use the **\$? variable**.

Using Structured Commands

if-then-Else statements, nesting if statements, test, compound testing, and case statements

Shell scripting | Conditions

- The **if statement** is used to carry out certain commands based on testing a condition and the exit status of the command.
- If statements are used to control how the script will execute.
- For instance, you might want a portion of the script to run if the user is in the Marketing Department and have another portion run if the user is in Human Resources.
- Or you may want to run another command if the a given command executes successfully.
- if statement**—Starts the condition being tested
- then statement**—Starts the portion of code specifying what to do if the condition evaluates to true
- else statement**—Starts the portion of code specifying what to do if the condition evaluates to false

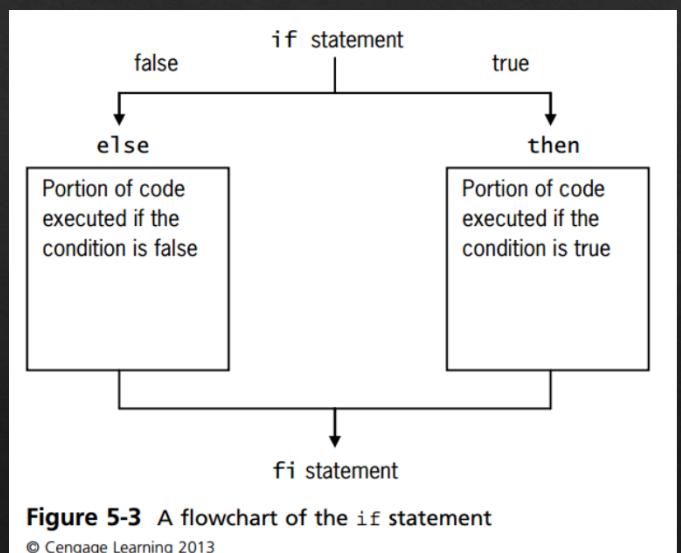


Figure 5-3 A flowchart of the if statement
© Cengage Learning 2013

How does if then works in bash?

IMPORTANT NOTE:

- if statements in bash are different than in if statements in other languages.
- In other programming languages, the object after the if statement is an equation that is evaluated for a **TRUE** or **FALSE** value.

The bash shell **if** statement runs the command defined on the if line.

If the **exit status** of the command is zero (the command completed successfully), the commands listed under the **then** section are executed. If the exit status of the command is anything else, the then commands aren't executed, and the bash shell moves on to the next command in the script.

if command

then

commands

fi

The **fi** statement indicates the end of the if statement

Example

```
#!/bin/bash
if pwd
then
```

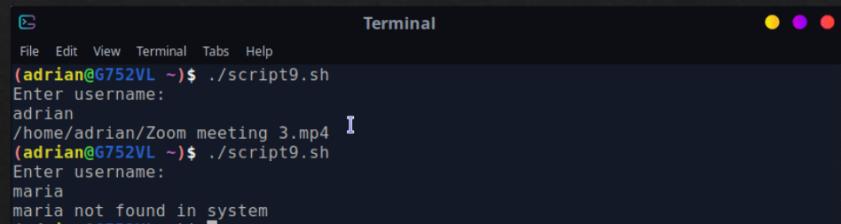
```
    echo "it worked"  
fi
```

if the **pwd** command executes successfully, the string **it worked** will be displayed

Here is a more interesting example!

- This simple script searches the filesystem for all the mp4 files owned by the user. The variable **\$user** stores the username provided by the person who runs the script. The if statement runs the find command only if the grep command executes successfully.
- Notice that in the previous example, the command after **if** showed the output after executing. In this example, we use output redirection to redirect the output of grep to the black hole.

```
#!/bin/bash  
# asks the user for a username  
echo "Enter username: "  
read user  
# checks if the user exists in the system  
# > /dev/null will send the output of grep to the black hole  
# using > /dev/null with if prevents the ouput of the command  
# from being displayed  
if grep $user /etc/passwd > /dev/null  
then  
    # find all mp4 files owned by the user in the user's  
    # home directory  
    find /home/$user -user $user -name *.mp4 2> /dev/null  
else  
    # displayed only if the user is not found in the system  
    # meaning the exit status code of the grep command  
    # is non-zero  
    echo "$user not found in system"  
fi
```



```
Terminal  
(adrian@G752VL ~)$ ./script9.sh  
Enter username:  
adrian  
/home/adrian/Zoom meeting 3.mp4 I  
(adrian@G752VL ~)$ ./script9.sh  
Enter username:  
maria  
maria not found in system  
(adrian@G752VL ~)$
```

Shell scripting | Comparison operators

Table 5-7 File attribute operators in the BASH shell

File attribute operator	Description
-a	Checks whether the file exists
-d	Checks whether the file is a directory
-f	Checks whether the file is a regular file
-r	Checks whether the user has read permission for the file
-s	Checks whether the file contains data
-w	Checks whether the user has write permission for the file
-x	Checks whether the user has execute permission for the file
-O	Checks whether the user is the owner of the file
-G	Checks whether the user belongs to the group owner of the file
file1 -nt file2	Checks whether file1 is newer than file2
file1 -ot file2	Checks whether file1 is older than file2

Numeric Comparison

Comparison	Description	Example
n1 -eq n2	Checks if n1 is equal to n2	If [\$n1 -eq \$n2]
n1 -ge n2	Checks if n1 is greater than or equal to n2	If [\$n1 -ge \$n2]
n1 -gt n2	Checks if n1 is greater than n2	If [\$n1 -gt \$n2]
n1 -le n2	Checks if n1 is less than or equal to n2	If [\$n1 -le \$n2]
n1 -lt n2	Checks if n1 is less than n2	If [\$n1 -lt \$n2]
n1 -ne n2	Checks if n1 is not equal to n2	If [\$n1 -ne \$n2]

String Comparison

Comparison	Description	Example
<code>str1 = str2</code>	Checks if str1 is the same as string str2	<code>If [\$str1 = \$str2]</code>
<code>str1 != str2</code>	Checks if str1 is not the same as str2	<code>If [\$str1 != \$str2]</code>
<code>str1 < str2</code>	Checks if str1 is less than str2	<code>If [\$str1 < \$str2]</code>
<code>str1 \> str2</code>	Checks if str1 is greater than str2	<code>If [\$str1 > \$str2]</code>
<code>-n str1</code>	Checks if str1 has a length greater than zero	<code>If [\$str1 -n]</code>
<code>-z str1</code>	Checks if str1 has a length of zero	<code>If [\$str1 -z]</code>

A **case statement** uses one variable to specify multiple values and matches a portion of the script to each value.

```

1 #!/bin/bash
2 clear
3 echo "-----"
4 echo -e "\twhich command would you like to run?"
5 echo "1) ls"
6 echo "2) ls -a"
7 echo "3) ls -l"
8 echo "4) ls -la"
9 echo "5) ls -laR"
10 echo "-----"
11 read answer
12
13 case $answer in
14     1)
15         ls
16         ;;
17     2)
18         ls -a
19         ;;
20     3)
21         ls -l
22         ;;
23     4)
24         ls -la
25         ;;
26     5)
27         ls -laR
28         ;;
29     *)
30         exit
31         ;;
32 esac
33
~
```

- ❖ **Looping** is used to perform a set of commands repeatedly. In the menu script, the user is given a list of options to choose from, and after a selection is made, the script ends.
- ❖ Shell scripting support different types of loops:

- while loop
- until loop
- for loop

```
while [ condition ]
do
    command1
    command2
    commandN
done
```

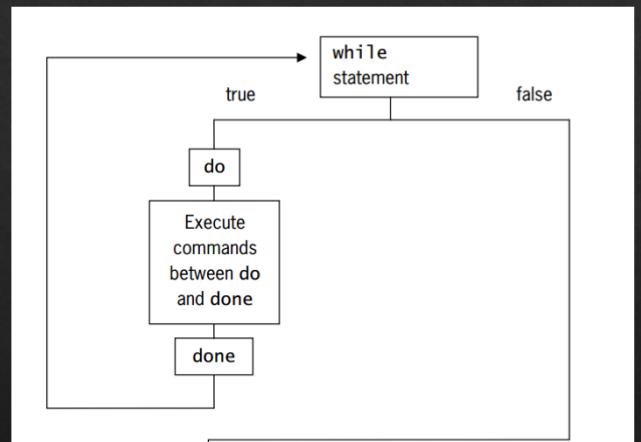


Figure 5-4 A while loop
© Cengage Learning 2013

- ❖ An **until loop** repeats the commands between do and done as long as the tested condition is false (exit status code is greater than 0)—in other words, until the condition is true.

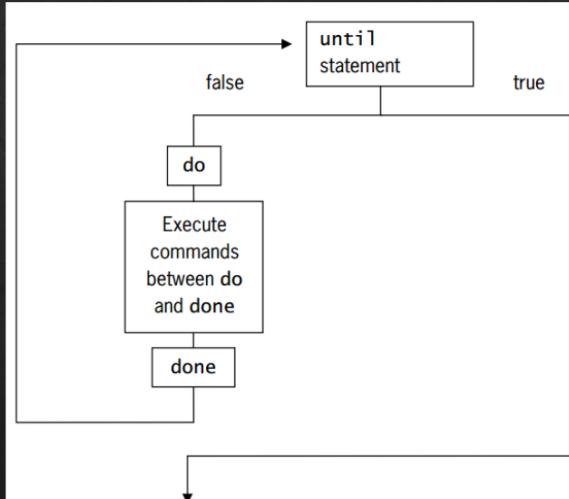


Figure 5-5 An until loop
© Cengage Learning 2013

- ❖ A **for loop** repeats the commands between do and done a specified number of times. Each time the script carries out the commands in the loop, a new value is given to a variable.

```
for VARIABLE in 1 2 3 4 5 .. N
do
    command1
    command2
    commandN
done
```

```
for VARIABLE in file1 file2 file3
do
    command1 on $VARIABLE
    command2
    commandN
done
```