# ITERATIVE FLOW CONTROL



## The while Statement

```
#!/bin/bash
echo “Guess the secret color: red, blue, yellow, purple, or orange \n”
read COLOR
while [ $COLOR != “purple” ]
do
echo “Incorrect. Guess again. \n”
read $COLOR
done
echo “Correct.”
```

# Example

```
$ cat test10
#!/bin/bash
# while command test
var1=10
while [ $var1 -gt 0 ]
do
echo $var1
var1=$[ $var1 - 1 ]
done
```



# The until Statement

- The condition in the until statement is the opposite of that in the while statement. For example, you could rewrite the previous example this way:

```
#!/bin/bash
echo “Guess the secret color: red, blue, yellow, purple, or orange \n”
read COLOR
until [ $COLOR = “purple” ]
do
echo “Incorrect. Guess again. \n”
read $COLOR
done
echo “Correct.”
```

# Example

```
#!/bin/bash
# using the until command
var1=100
until [ $var1 -eq 0 ]
do
echo $var1
var1=$[ $var1 - 25 ]
done
```



# The for Command

- The bash shell provides the for command to allow you to create a loop that iterates through a series of values.
- The basic format of the bash shell for command is:

```
for var in list
    do
        commands
done
```

# Reading values in a list

Iterating through a list using for

```
#!/bin/bash
# basic for command
for test in Alabama Alaska Arizona Arkansas California
do
echo The next state is $test
done
```



# Reading complex values in a list

- Here's a classic example of what can cause shell script programmers problems:

```
#!/bin/bash
# another example of how not to use the for command
for test in I don't know if this'll work
do
echo "word:$test"
done
```

# Two ways to solve

- Use the escape character (the backslash) to escape the single quotation mark.
- Use double quotation marks to define the values that use single quotation marks.

```
#!/bin/bash
# another example of how not to use the for command
for test in I don\'t know if "this'll" work
do
echo "word:$test"
done
```



# Reading a list from a variable

```
#!/bin/bash
# using a variable to hold the list
list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut"
for state in $list
do
echo "Have you ever visited $state?"
done
```

# Reading values from a command

```
#!/bin/bash
# reading values from a file
file="states"
for state in `cat $file`
do
echo "Visit beautiful $state"
done
```

```
cat > states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
```



# Reading a directory using wildcards

- Finally, you can use the for command to automatically iterate through a directory of files.
- To do this, you must use a wildcard character in the file or pathname.
- This forces the shell to use **file globbing**.
- File globbing is the process of producing file or path names that match a specified wildcard character.

# Example

```
#!/bin/bash
# iterate through all the files in a directory
for file in /home/ankur/test/*
do
if [ -d "$file" ]
then
echo "$file is a directory"
elif [ -f "$file" ]
then
echo "$file is a file"
fi
done
```



# Changing the field separator

- By default, the bash shell considers the following characters as field separators:
  - A space
  - A tab
  - A newline
- You can change the IFS environment variable in your script to set the field separator.
- For example, if you want to change the IFS value to only recognize the newline character, you need to do this:
  - IFS=$'\n'

# Example

```
#!/bin/bash
# reading values from a file
file="states"
IFS=$’ \ n’
for state in `cat $file`
do
echo "Visit beautiful $state"
done
```



# Caution

- When working on long scripts, it’s possible to change the IFS value in one place, then forget about it and assume the default value elsewhere in the script. This technique can be coded like this:

```
IFS.OLD=$IFS
IFS=$’ \ n’
‹use the new IFS value in code›
IFS=$IFS.OLD
```

# An Interesting Application of IFS

- You can change the value of the IFS variable to anything:
  - IFS=:
- Or you can also do the following:
  - IFS=$'\n':;"

# The C-Style for Command

```
#!/bin/bash
# testing the C-style for loop
for (( i=1; i <= 10; i++ ))
do
echo "The next number is $i"
done
```

# Using multiple variables

- The C-style for command also allows you to use multiple variables for the iteration.

```
#!/bin/bash
# multiple variables
for (( a=1, b=10; a <= 10; a++, b-- ))
do
echo "$a - $b"
done
```



# Nesting Loops

- A loop statement can use any other type of command within the loop, including other loop commands. This is called a *nested loop*

```
#!/bin/bash
# nesting for loops
for (( a = 1; a <= 3; a++ ))
do
echo "Starting loop $a:"
for (( b = 1; b <= 3; b++ ))
do
echo "    Inside loop: $b"
done
done
```

# Mixing Loops

```
#!/bin/bash
# placing a for loop inside a while loop
var1=5
while [ $var1 -ge 0 ]
do
echo "Outer loop: $var1"
for (( var2 = 1; $var2 < 3; var2++ ))
do
var3=$[ $var1 * $var2 ]
echo "    Inner loop: $var1 * $var2 = $var3"
done
var1=$[ $var1 - 1 ]
done
```



# Test Your Brains

```
#!/bin/bash
# using until and while loops
var1=3
until [ $var1 -eq 0 ]
do
echo "Outer loop: $var1"
var2=1
while [ $var2 -lt 5 ]
do
var3=`echo "scale=4; $var1 / $var2" | bc`
echo "    Inner loop: $var1 / $var2 = $var3"
var2=$[ $var2 + 1 ]
done
var1=$[ $var1 - 1 ]
done
```

# Looping on File Data

- Often, you must iterate through items stored inside a file. This requires combining two of the techniques covered:
  - Using nested loops
  - Changing the IFS environment variable



# Example

```
#!/bin/bash
# changing the IFS value
IFS.OLD=$IFS
IFS=$' \ n'
for entry in `cat /etc/passwd`
do
echo "Values in $entry -"
IFS=:
for value in $entry
do
echo "    $value"
done
done
```

# Controlling The Loop

- The break command
- The continue command



# The break command

```
#!/bin/bash
# breaking out of a for loop
for var1 in 1 2 3 4 5 6 7 8 9 10
do
if [ $var1 -eq 5 ]
then
break
fi
echo "Iteration number: $var1"
done
echo "The for loop is completed"
```

# Breaking out of an inner loop

```
#!/bin/bash
# breaking out of an inner loop
for (( a = 1; a < 4; a++ ))
do
echo "Outer loop: $a"
for (( b = 1; b < 100; b++ ))
do
if [ $b -eq 5 ]
then
break
fi
echo "    Inner loop: $b"
done
done
```



# Breaking out of an outer loop

- There may be times when you're in an inner loop but need to stop the outer loop.
- The break command includes a single command line parameter value:
  - break n
- where n indicates the level of the loop to break out of the current loop.

# Example

```
#!/bin/bash
# breaking out of an outer loop
for (( a = 1; a < 4; a++ ))
do
echo "Outer loop: $a"
for (( b = 1; b < 100; b++ ))
do
if [ $b -gt 4 ]
then
break 2
fi
echo "    Inner loop: $b"
done
done
```

# The continue command

- The continue command is a way to prematurely stop processing commands inside of a loop but not terminate the loop completely.

# Example

```
#!/bin/bash
# using the continue command
for (( var1 = 1; var1 < 15; var1++ ))
do
if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
then
continue
fi
echo "Iteration number: $var1"
done
```



# Processing the Output of a Loop

- Finally, you can either pipe or redirect the output of a loop within your shell script.
- You do this by adding the processing command to the end of the done command.

# Example

```
for file in /home/ankur/*
do
if [ -d "$file" ]
then
echo "$file is a directory"
elif
echo "$file is a file"
fi
done > output.txt
```



# Example 2

- This same technique also works for piping the output of a loop to another command:

```
#!/bin/bash
# piping a loop to another command
for state in "North Dakota" Connecticut Illinois Alabama
  Tennessee
do
echo "$state is the next place to go"
done | sort
echo "This completes our travels"
```

# Permanent redirections

- You can redirect your script output to file using exec command.

```
#!/bin/bash
# redirecting all output to a file
exec 1>testout
echo "This is a test of redirecting all output"
echo "from a script to another file."
echo "without having to redirect every individual line"
```

# HANDLING USER INPUT

# Command Line Parameters

- The most basic method of passing data to your shell script is by using command line parameters.
  - $ ./addem 10 30



# Reading parameters

- The bash shell assigns special variables, called positional parameters, to all of the parameters entered in a command line.
  - $0 –Program Name
  - $1 -$9- Command Line Parameters

# Example

```
#!/bin/bash
# using one command line parameter
factorial=1
for (( number = 1; number <= $1 ; number++ ))
do
factorial=$[ $factorial * $number ]
done
echo The factorial of $1 is $factorial
$ ./test1 5
```



# You can also pass Strings

```
#!/bin/bash
# testing string parameters
echo Hello $1, glad to meet you.
$ ./test3 Ankur
```

# Spaces in Strings

- To handle spaces, include them in single or double quotes.

```
$ ./test3 'Ankur Rathore'
Hello Rich Blum, glad to meet you.
$ ./test3 "Ankur Rathore"
```



# More Than Nine Parameters

- If your script needs more than nine command line parameters, you can continue, but the variable names change slightly.
- After the ninth variable, you must use braces around the variable number, such as ${10}

# Example

```
#!/bin/bash
# handling lots of parameters
total=$[ ${10} * ${11} ]
echo The tenth parameter is ${10}
echo The eleventh parameter is ${11}
echo The total is $total
$ ./test4 1 2 3 4 5 6 7 8 9 10 11 12
```



# Reading the program name

- You can use the $0 parameter to determine the name of the program.

```
#!/bin/bash
# testing the $0 parameter
echo The command entered is: $0
$ ./test5
```

This returns the whole path of the command.

# Only Name

```
#!/bin/bash
# using basename with the $0 parameter
name=`basename $0`
echo The command entered is: $name
$ ./test5b
```



# Testing parameters

- It's a good habit to check whether, parameters have been entered or not.

```
#!/bin/bash
# testing parameters before use
if [ -n "$1" ]
then
echo Hello $1, glad to meet you.
else
echo "Sorry, you didn't identify yourself."
fi
```

# Counting parameters

- The special $# variable contains the number of command line parameters included when the script was run.

```
#!/bin/bash
# testing parameters
if [ $# -ne 2 ]
then
echo Usage: test9 a b
else
total=$[ $1 + $2 ]
echo The total is $total
fi
```



# A Trick

```
#!/bin/bash
# grabbing the last parameter
params=$#
echo The last parameter is $params
echo The last paramerer is ${!#}
```

# Grabbing all the data

- The $* and $@ variables provide one-stop shopping for all of your parameters.
  - The $* variable takes all of the parameters supplied on the command line as a single word.
  - The $@ variable on the other hand, takes all of the parameters supplied on the command line as separate words in the same string.



# Example

```
#!/bin/bash
# testing $* and $@
count=1
for param in "$*"
do
echo "\$* Parameter #$count = $param"
count=$[ $count + 1 ]
done
count=1
for param in "$@"
do
echo "\$@ Parameter #$count = $param"
count=$[ $count + 1 ]
done
```

# Being Shifty

- When you use the shift command, it "downgrades" each parameter variable one position by default. Thus, the value for variable $3 is moved to $2, the value for variable $2 is moved to $1,



# Example

```
#!/bin/bash
# demonstrating the shift command
count=1
while [ -n "$1" ]
do
echo "Parameter #$count = $1"
count=$[ $count + 1 ]
shift
done
```