

# Yet More Script Stuff

```
#!/bin/bash
# scriptname: testparms
#
# shows how $* deals with more than 9 args
#       without special treatment
#
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12"
echo $*
echo $#
```

Run:

```
$ testparms a b c d e f g h i j k l
a b c d e f g h i a0 a1 a2
a b c d e f g h i j k l
12
```

## What Happened?

\$\* doesn't know how to deal with args 10, 11, and 12, so it interpolates as it can.

S\* sees \$10 as arg1 contents and a 0 thus a0

S\* sees \$11 as arg1 contents and a 1 thus a1

S\* sees \$12 as arg1 contents and a 0 thus a2

## Shift

The shift command takes an argument, uses it, then tosses it away and moves the remaining arguments up one position

```
#!/bin/bash
# scriptname: shifter
echo $1 $2 $3
shift
echo $1 $2 $3
shift
echo $1 $2 $3
```

```
Run:
$ shifter a b c
a b c
b c
c
$
```

Note that once an arg is shifted off of the list, it has gone to the bit bucket.

## The difference between \$\* and \$@

While the differences between the \$\* and \$@ may seem subtle, it is important to distinguish between them.

As you have seen \$\* represents the complete list of characters as one string. \$@ represents the contents as separate strings.

```
#!/bin/bash  
# scriptname: atstar  
echo $*  
echo $@
```

Run

```
$ atstar x "this is a string" 477  
"x this is a string 477"  
"x" "this is a string" "477"  
$
```

Note that the double-quotes are shown to display the difference. They are not echoed out.

When using these variables within your programs, be aware that the shell stores them in two different ways.

## Using \$\* and \$@ in loops

```
rgrogan@padme cs534scripts $ cat ampr.bash
```

```
#!/bin/bash  
# for loop with ampersand  
# scriptname: ampr.bash
```

```
echo 'this loop uses $@'
```

```
echo
```

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

Run

```
$ ampr.bash 1 2 3 4 5  
this loop uses $@
```

```
1  
2  
3  
4  
5  
$
```

```
rgrogan@padme cs534scripts $ cat star.bash
```

```
#!/bin/bash  
# for loop with asterisk  
# scriptname: star.bash
```

```
echo 'this loop uses $*'
```

```
echo
```

```
for arg in "$*"  
do  
    echo "$arg"  
done
```

```
Run
```

```
$ star.bash 1 2 3 4 5  
this loop uses $*
```

```
1 2 3 4 5
```

```
$
```

# Redirecting stdin to a script

```
#!/bin/bash
# scriptname: getname
echo -n "Please enter your name: "
read NAME          # reads from keyboard
echo "Your name is $NAME"
```

run:

```
$ getname
Please enter your name: rozz
Your name is rozz
```

OK. What do you think the following does?

**read < afile**

The read operation will only read the first line (up to the end-of-line marker) from afile - it doesn't read the entire file.

## More Tidbits on Testing Expressions

The **test** command allows you to:

- test the length of a string
- compare two strings
- compare two numbers
- check on a file's type
- check on a file's permissions
- combine conditions together

**test** actually comes in two flavours:

**test an\_expression**  
**and**  
**[ an\_expression ]**

They are both the same thing - it's just that **[** is soft-linked to **/usr/bin/test**

**test** actually checks to see *what name* it is being called by

if it is **[** then it expects a **]** at the end of the expression.

## Example

```
if [ "$1" = "hello" ]
then
echo "hello to you too!"
else
echo "hello anyway"
fi
```

is functionally identical to

```
if test "$1" = "hello"
then
echo "hello to you too!"
else
echo "hello anyway"
fi
```

**NOTE:** Note that the variable \$1 is surrounded in quotes.

This is to take care of the case when \$1 doesn't exist - in other words, ***there were no parameters passed.***

If we had simply put \$1 and there wasn't any \$1, then an error would have been displayed:

**test: =: unary operator expected**

<b>Expression</b>	<b>True if</b>
-z string	length of string is 0
-n string	length of string is not 0

<b>Expression</b>	<b>True if</b>
string1 = string2	if the two strings are identical
string != string2	if the two strings are NOT identical
String	if string is not NULL

<b>Expression</b>	<b>True if</b>
int1 -eq int2	first int is equal to second
int1 -ne int2	first int is not equal to second
int1 -gt int2	first int is greater than second
int1 -ge int2	first int is greater than or equal to second
int1 -lt int2	first int is less than second
int1 -le int2	first int is less than or equal to second

<b>Expression</b>	<b>True if</b>
<b>-r file</b>	<b>file exists and is readable</b>
<b>-w file</b>	<b>file exists and is writable</b>
<b>-x file</b>	<b>file exists and is executable</b>
<b>-f file</b>	<b>file exists and is a regular file</b>
<b>-r file</b>	<b>File exists and is readable</b>
<b>-w file</b>	<b>file exists and is writable</b>
<b>-x file</b>	<b>file exists and is executable</b>
<b>-f file</b>	<b>file exists and is a regular file</b>
<b>-d file</b>	<b>file exists and is directory</b>
<b>-h file</b>	<b>file exists and is a symbolic link</b>
<b>-c file</b>	<b>file exists and is a character special file</b>
<b>-b file</b>	<b>file exists and is a block special file</b>
<b>-p file</b>	<b>file exists and is a named pipe</b>
<b>-u file</b>	<b>file exists and it is setuid</b>
<b>-g file</b>	<b>file exists and it is setgid</b>
<b>-k file</b>	<b>file exists and the sticky bit is set</b>
<b>-s file</b>	<b>file exists and its size is greater than 0</b>

<b>Expression</b>	<b>Purpose</b>
<b>!</b>	<b>reverse the result of an expression</b>
<b>-a</b>	<b>AND operator</b>
<b>-o</b>	<b>OR operator</b>
<b>( expr )</b>	<b>group an expression, parentheses have special meaning to the shell so to use them in the test command you must quote them</b>

**NOTE: test uses different operators to compare *strings and numbers***

**using -ne on a string comparison and**

**!= on a numeric comparison**

**is incorrect and will give undesirable results.**

## More About Case

```
echo -n "Your Answer, yes or no: "  
read ANSWER  
case $ANSWER in  
Y* | y*) ANSWER="YES"  
;;  
N* | n*) ANSWER="NO"  
;;  
) ANSWER="MAYBE" # the trailing else  
;;  
esac  
echo $ANSWER
```

In this case, the | (single pipe stands for OR),

\* stands for any number of characters

## More on While Loops

The format of the **while** construct is:

```
while command  
do  
    commands  
done
```

that is, while command is true, commands are executed

### Example

```
while [ $1 ]  
do  
echo $1  
shift    # look familiar?  
done
```

## **while also allows the redirection of input**

**Consider the following:**

```
#!/bin/bash  
# scriptname: linelist  
#  
# This script reads a file line by line and  
# echo's it to the screen with a line number.  
  
count=0  
while read BUFFER      # while the file is not  
                        # empty, read a line  
                        # from the file  
  
do  
count=[ $count += 1 ] # Increment the count  
echo "$count $BUFFER" # Echo it out  
done < $1             # Take input line from the file
```

## break and continue

Occasionally you will want to *jump* out of a loop.

This is done with the *break* command.

For example:

```
while true
do
read BUFFER
if [ "$BUFFER" = "" ]           # empty line
then
break           # jump to next executable statement
                # after done
fi
echo $BUFFER
done
```

This code takes a line from the user and prints it until the user enters a blank line.

There is a less used form, **break n** (where n is a number).

This form works by executing break "n" times. This can break you out of embedded loops.

For example:

```
for file in $*
do
while read BUFFER
do
if [ "$BUFFER" = "ABORT" ]
then
break 2
fi
echo $BUFFER
done < $file
done
```

This code prints the contents of multiple files, but if it encounters a line containing the word "**ABORT**" in any one of the files, it stops processing.

# Continue

Like **break**, **continue** is used to alter the looping process. However, unlike **break**, **continue** keeps the looping process going...it just fails to finish the remainder of the ***current loop*** by returning to the ***top of the loop***.

For example:

```
while read BUFFER
do
charcount=`echo $BUFFER | wc -c | cut -f1`
if [ $ charcount -gt 80 ]
then
continue
fi
echo $BUFFER
done < $1
```

**This code segment reads and echos the contents of a file...however, it does not print lines that are over 80 characters long.**