ADVANCE SHELL SCRIPTING

CREATING FUNCTIONS

Creating a function

- The first format uses the keyword function, along with the function name you assign to the block of code:

```
function name {
commands
}
```

- The Second Format:

```
name() {
commands
}
```

Using functions

```
#!/bin/bash
# using a function in a script
function func1 {
echo "This is an example of a function"
}
count=1
while [ $count -le 5 ]
do
func1
count=$[ $count + 1 ]
done
echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
```

Function must have a Unique name

```
#!/bin/bash
# testing using a duplicate
    function name
function func1 {
echo "This is the first definition of
    the function name"
}
func1
function func1 {
echo "This is a repeat of the
    same function name"
}
func1
echo "This is the end of the
    script"
```

- If you redefine a function, the new definition will override the original function definition, without producing any error messages

RETURNING A VALUE

The default exit status

- By default, the exit status of a function is the exit status returned by the last command in the function. After the function executes, you use the standard $? variable to determine the exit status of the function:

```
#!/bin/bash
# testing the exit status of a function
func1() {
echo "trying to display a non-existent file"
ls -l badfile
}
echo "testing the function:"
func1
echo "The exit status is: $?"
```


Using the return command

```
#!/bin/bash
# using the return command in a function
function dbl {
read -p "Enter a value: " value
echo "doubling the value"
return $[ $value * 2 ]
}
dbl
echo "The new value is $?"
```

Remember

- You must be careful though when using this technique to return a value from a function. There are two things that can cause problems:
  - Remember to retrieve the return value as soon as the function completes.
  - Remember that an exit status can only be in the range of 0 to 255.

Using function output

- You can use this technique to retrieve any type of output from a function to assign to a variable:
  - result=`dbl`

Example

```
#!/bin/bash
# using the echo to return a value
function dbl {
read -p "Enter a value: " value
echo $[ $value * 2 ]
}
result=`dbl`
echo "The new value is $result"
```


Passing parameters to a function

- Functions can use the standard parameter environment variables to represent any parameters passed to the function on the command line.
- For example, the name of the function is defined in the $0 variable, and any parameters on the function command line are defined using the variables $1, $2, and so on.

Example

```
#!/bin/bash
# passing parameters to a function
function addem {
if [ $# -eq 0 ] || [ $# -gt 2 ]
then
echo -1
elif [ $# -eq 1 ]
then
echo $[ $1 + $1 ]
else
echo $[ $1 + $2 ]
fi
}
echo -n "Adding 10 and 15: "
value=`addem 10 15`
echo $value
echo -n "Let's try adding just one number: "
value=`addem 10`
echo $value
echo -n "Now trying adding no numbers: "
value=`addem`
echo $value
echo -n "Finally, try adding three numbers: "
value=`addem 10 15 20`
echo $value
```

A Point

- Since the function uses the special parameter environment variables for its own parameter values, it can't directly access the script parameter values from the command line of the script.

Example

```
#!/bin/bash
# trying to access script parameters inside a function
function badfunc1 {
echo $[ $1 * $2 ]
}
if [ $# -eq 2 ]
then
value=`badfunc1`
echo "The result is $value"
else
echo "Usage: badtest1 a b"
fi
```


Handling variables in a function

- Functions use two types of variables:
  - Global
  - Local

Global variables

- Global variables are variables that are valid anywhere within the shell script.
- If you define a global variable in the main section of a script, you can retrieve its value inside a function.

Example

```
#!/bin/bash
# using a global variable to pass a value
function dbl {
value=$[ $value * 2 ]
}
read -p "Enter a value: " value
dbl
echo "The new value is: $value"
```

Things can go wrong

```
#!/bin/bash
# demonstrating a bad use of variables
function func1 {
temp=$[ $value + 5 ]
result=$[ $temp * 2 ]
}
temp=4
value=6
func1
echo "The result is $result"
if [ $temp -gt $value ]
then
echo "temp is larger"
else
echo "temp is smaller"
fi
```


Local variables

- To do that, just use the local keyword in front of the variable declaration:
  - local temp
- You can also use the local keyword in an assignment statement while assigning a value to the variable:
  - local temp=$[$value + 5]

Example

```
#!/bin/bash
# demonstrating the local keyword
function func1 {
local temp=$[ $value + 5 ]
result=$[ $temp * 2 ]
}
temp=4
value=6
func1
echo "The result is $result"
if [ $temp -gt $value ]
then
echo "temp is larger"
else
echo "temp is smaller"
fi
```


Designing a Function

- Lets design a function to check for a valid IP address.
- Function definition:
  - isvalidip()

Next Step

- The first set of tests is contained in a case statement:

```
case $1 in
  "" | *[!0-9.]* | *[!0-9]) return 1 ;;
esac
```

- It checks for an empty string, invalid characters, or an address that doesn't end with a digit.

- Next we declare a local variable to set the IFS.
  - local IFS=.
- The set builtin replaces the positional parameters with its arguments. Since $IFS is a period, each element of the IP address is assigned to a different parameter.
  - set -- $1

- The final two lines check each positional parameter in turn. If it's greater than 255, it is not valid in a dotted-quad IP address. If a parameter is empty, it is replaced with the invalid value of 666.

```
[ ${1:-666} -le 255 ] && [ ${2:-666} -le 255 ] &&
[ ${3:-666} -le 255 ] && [ ${4:-666} -le 255 ]
```



```
isvalidip()
{
 case $1 in
  ""|*[!0-9.]*|*[!0-9]) return 1 ;;
 esac
 ## Change IFS to a dot, but only in this function
 local IFS=.
 set -- $1
 [ $# -eq 4 ] &&
 [ ${1:-666} -le 255 ] && [ ${2:-666} -le 255 ] &&
 [ ${3:-666} -le 255 ] && [ ${4:-666} -le 255 ]
}
for ip in 127.0.0.1 168.260.0.234 1.2.3.4 123.1OO.34.21 204.225.122.150
do
 if isvalidip "$ip"
 then
  printf "%15s: valid\n" "$ip"
 else
  printf "%15s: invalid\n" "$ip"
 fi
done
```

Print the Result

- A function's purpose may be to print information, either to the terminal or to a file.

```
uinfo( )
{
  printf "%12s: %s\n" \
    USER    "${USER:-No value assigned}" \
    PWD     "${PWD:-No value assigned}" \
    COLUMNS "${COLUMNS:-No value assigned}" \
    LINES   "${LINES:-No value assigned}" \
    SHELL   "${SHELL:-No value assigned}" \
    HOME    "${HOME:-No value assigned}" \
    TERM    "${TERM:-No value assigned}"
} > ${1:-/dev/fd/1}
```


Variable Arrays

- A really cool feature of environment variables is that they can be used as arrays.
- To set multiple values for an environment variable, just list them in parentheses, with each value separated by a space:
  - $ mytest=(one two three four five)
- Not much excitement there. If you try to display the array as a normal environment variable, you'll be disappointed:
  - $ echo $mytest
    - one

- To reference an individual array element, you must use a numerical index value, which represents its place in the array. The numeric value is enclosed in square brackets:
  - $ echo ${mytest[2]}

Displaying the entire value

- To display an entire array variable, you use the asterisk wildcard character as the index value:
  - $ echo ${mytest[*]}

Changing values

- You can also change the value of an individual index position:
  - $ mytest[2]=seven
  - $ echo ${mytest[*]}
  - one two seven four five
- You can even use the unset command to remove an individual value within the array, but be careful, as this gets tricky. Watch this example:
  - $ unset mytest[2]
  - $ echo ${mytest[*]}
- Finally, you can remove the entire array just by using the array name in the unset command:
  - $ unset mytest
  - $ echo ${mytest[*]}

Passing arrays to functions

```
#!/bin/bash
# trying to pass an array variable
function testit {
echo "The parameters are: $@"
thisarray=$1
echo "The received array is ${thisarray[*]}"
}
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
testit $myarray
```

- If you try using the array variable as a function parameter, the function only picks up the first value of the array variable.
- To solve this problem, you must disassemble the array variable into its individual values, then use the values as function parameters.

The Proper Technique

```
#!/bin/bash
# array variable to function test
function testit {
local newarray
newarray=(`echo "$@"`)
echo "The new array value is: ${newarray[*]}"
}
myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
```

```
#!/bin/bash
# adding values in an array
function addarray {
local sum=0
local newarray
newarray=(`echo "$@"`)
for value in ${newarray[*]}
do
sum=$[ $sum + $value ]
done
echo $sum
}
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=`addarray $arg1`
echo "The result is $result"
```

Returning An Array

```
#!/bin/bash
# returning an array value
function arraydblr {
local origarray
local newarray
local elements
local i
origarray=(`echo "$@"`)
newarray=(`echo "$@"`)
elements=$[ $# - 1 ]
for (( i = 0; i <= $elements; i++ ))
{
newarray[$i]=$[ ${origarray[$i]} * 2 ]
}
echo ${newarray[*]}
}
myarray=(1 2 3 4 5)
echo "The original array is:
${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=(`arraydblr $arg1`)
echo "The new array is:
${result[*]}"
```

A function to perform Bubble Sort

```
function bubblesort()
{
 n=${#data[@]}
 for i in `seq 0 $n`
 do
  for (( j=n; j > i; j-=1 ))
  do
   if [[ ${data[j-1]} > ${data[j]} ]]
then
     temp=${data[j]}
     data[j]=${data[j-1]}
     data[j-1]=$temp
     fi
  done
done
}
```

```
data=( roger oscar charlie kilo indigo
       tango )
echo "Initial state:"
for i in ${data[@]}
do
 echo "$i"
done
bubblesort
echo
echo "Final state:"
for i in ${data[@]}
do
 echo "$i"
done
```


Selecting range from arrays

- You can access a range of values from the array by the following syntax:
  - echo ${array_name[@]:Index2:Index2}
- Example
  - $ food=(apples bananas cucumbers dates eggs fajitas grapes)
    - $ echo ${food[@]:0:1}
    - $ echo ${food[@]:2:4}

Associative Arrays

- The associative array is a new feature in bash version 4.
- Associative arrays link (associate) the value and the index together, so you can associate metadata with the actual data.

Example

```
#!/bin/bash
declare -A beatles
beatles=( [singer]=John [bassist]=Paul [drummer]=Ringo
 [guitarist]=George )
for musician in singer bassist drummer guitarist
do
 echo “The ${musician} is ${beatles[$musician]}.”
Done
```

- What makes associative arrays even more useful is the ability to reference back to the name of the index.
- To do this, use the syntax
  - ${!array[@]}

Example

```
#!/bin/bash
declare -A beatles
beatles=( [singer]=John [bassist]=Paul
  [drummer]=Ringo [guitarist]=George )
for instrument in ${!beatles[@]}
do
  echo "The ${instrument} is
    ${beatles[$instrument]}"
done
```

copying an array

- Copying one array to another is simple.
- It is important for quoting and spacing that the ${array[@]} format.

```
activities=( swimming “water skiing” canoeing “white-
    water rafting” surfing )
$ for act in ${activities[@]}
> do
>   echo “Activity: $act”
> done
```

The proper way

```
hobbies=( “${activities[@]}” )
$ for hobby in “${hobbies[@]}”
> do
>   echo “Hobby: $hobby”
> done
```

Appending to an array

- Appending to an array is much the same as copying it. The simplest way to append to an array is to extend the syntax for copying an array.

```
$ hobbies=( "${activities[@]" diving )
$ for hobby in "${hobbies[@]}"
> do
>   echo "Hobby: $hobby"
> done
```


Another technique

- This notation returns the last element of the array:
  - [${#hobbies[@]}]
- Example

```
$ hobbies[${#hobbies[@]}]=rowing
$ for hobby in "${hobbies[@]}"
> do
>   echo "Hobby: $hobby"
> done
```

Combining two arrays

- The bash shell does have a builtin syntax to combine two arrays.
- With the C-like notation of +=, this method is concise and allows for very clear code.

```
$ airsports=( flying gliding parachuting )
$ activities+=("${airsports[@]}")
$ for act in "${activities[@]}"
> do
>   echo "Activity: $act"
> done
```


Recursive Functions

```
#!/bin/bash
# using recursion
function factorial {
if [ $1 -eq 1 ]
then
echo 1
else
local temp=$[ $1 - 1 ]
local result=`factorial $temp`
echo $[ $result * $1 ]
fi
}
read -p "Enter value: " value
result=`factorial $value`
echo "The factorial of $value is: $result"
```