Borne Again Shell Continued

The shift operator

The shift operator increments each of the command line arguments. When it is executed,

the current $1 houses the value of $2,
the current $2 houses the value of $3, and so on

and when called 2nd time
the current $1 will house the original parameter value of $3
the current $2 will house the original parameter value of $4 etc...

Consider generating a program that will sum all the command line arguments and displaying the sum on screen using the while loop

```
#!/bin/bash
count=$#
arg=$1
sum=$1
echo –e "$arg + \c"
while [ "$count" -gt 1 ]
do
 shift #increment by one argument
 arg=$1
 sum=`expr $sum + $arg`
if test $count -eq 2
then
   echo "$arg = $sum"
else
 echo –e "$arg + \c"
fi
count=`expr $count – 1`
done
echo
exit 0
```

>> ./add.sh 3 5 2
3 + 5 + 2 = 10

Numeric Variables

ksh variables are **strings** or **integers**, depending on how they are defined

```
var=100 makes var the string '100'

integer var=100 makes var the integer 100
```

To manipulate numeric variables using C-style expressions, use either

`$(( <expression> ))` to return the value of expression

`(( <expression> ))` to return only an exit status (true or false).

Example

```
integer x=1  #declare x as an integer and assign value of 1
(( y=x*10 )) #evaluate numerical expression and this will return 1
(( x+=1 )) #evaluate numerical expression and this will return 1
echo $x $y #display the value of vars x and y
```

```
2 10
```

Within `$(( <expression> ))` and `(( <expression> ))`

you can use parentheses for grouping, the arithmetic operators +, -, *, /, %

and the relational operators <, >, <=, >=, ==, !=, &&, and ||.

Arrays

There are two ways to assign values to elements of an array

The first is the most intuitive: you can use the standard shell variable assignment syntax with the array index in brackets ([]).

For example:

```
nicknames[2]="bob"
nicknames[3]="ed"
```

Assigns the values **bob** and **ed** into the elements of the array **nicknames** with indices 2 and 3, respectively. As with regular shell variables, values assigned to array elements are treated as character strings.

Arrays

```
#!/bin/bash
nick[2]="bob"
nick[3]="fred"
echo "index 2 ${nick[2]}"
echo "index 3 ${nick[3]}"
echo "print all elements in array ${nick[*]}"
exit
```

```
>>./example.sh
index 2 bob
index 3 fred
print all elements in array bob fred
```

The second way to assign values to an array is with a variant of the **set** statement

An array variable provides a way to index a list of values:

Aname=(val1 val2 val3 ...)

creates the array *aname* (if it doesn't already exist) and assigns *val1* to *aname[0]*, *val2* to *aname[1]*, etc.

people=(Jack and Jill)

People is the array name it has three elements

Jack

and

Jill

To access values of an array use

${arrayname[element_no]}

note element_no will start from 0

The element_no can be an arithmetic expression

If you use * in place of the element_no, the value will be all elements, separated by spaces

Omitting the index is the same as specifying index 0

Example modify the addition program to store the argument list into an array using set and then access each of the elements from the array to generate the sum

```
#!/bin/bash
i=0
Test=($@)   #Test contains the list of arguments
val=${Test[0]}   #assign 1st argument to val
echo
count=$#
sum=$val
echo -e "$val + \c"
while [ "$count" -gt 1 ]
do
  ((i=i+1)) #increment the value of i by 1
  val=${Test[i]}
  sum=`expr $sum + $val`
if test $count -eq 2
then
  echo "$val = $sum"
else
  echo -e "$val + \c"
fi
((count=count-1)) # count=`expr $count - 1`
done
echo
```

Output from script when executed

```
./example.sh 2 3 4

2 + 3 + 4 = 9
```

The shell provides an operator that tells you how many elements an array has defined

${#*aname*[*]}

Note that you need the **[*]** because the name of the array alone is interpreted as the 0th element

example

```
people=(Jack and Jill)
others=(${people[*]} and Jane and John)
others[7]=and ; others[8]=Joseph
```

Array people[0] = Jack

people[1] = and

people[2] = Jill

others[0] = people[0] = Jack

others[1] = people[1] = and

others[2] = people[2] = Jill

others[3] = and

others[4] = Jane

Array others[5] = and

others[6] = John

others[7] = and

others[8] = Joesph

Items in an array can be accessed by position (first item is at index 0).

```
echo $people      # prints first element of array others,
      ie, ${people[0]} Jack

 echo ${people[0]}      # same as above Jack

 echo ${people[1]}     # prints second element of array others

  echo ${others[$(( ${#others[*]} - 1 ))]}
       # prints last element of array others Joseph
```

To evaluate the number of characters in an array element say i. We have

```
${#variable[i]}
```

Example

echo ${#others}

This will print the number of characters in first element of array others

ie, ${others[0]} returning 4

To print the number of characters in second element of array others

echo ${#others[1]} returns 3

Another example

```
#!/bin/bash
today=($(date))
echo "${today[*]}"
echo "${#today[*]}"
echo "${today[1]} ${today[2]}, ${today[5]}"
exit 0
```

```
Output
 Mon Feb 20 07:31:58 EDT 2006
 6
 Feb 20, 2006
```

Case Selection

The basic form of the case statement is:

```
case expression in
  pattern1 )
        statements ;;
 pattern2 )
        statements ;;
. . .
esac
```

The statements corresponding to the first pattern matching the expression are executed, after which the case statement terminates.

The *expression* is usually some variable's value

The *patterns* can be plain strings, or they can be bash shell patterns using `*, ?, !, [],` etc

A pattern can consist of several patterns separated by `|` (logical or), and the pattern can also be written as (pattern).

```
case $person in
  steve)
       echo "He's on the sixth floor." ;;
  todd | markus)
       echo "He's on the fifth floor." ;;
  *)
       echo  I do not know $person. ;;
  esac
```

bash shell functions

To define a function

```
function functname {
    shell commands
}
```

You can also delete a function definition with the command **unset -f** *functname*.

When calling a function

Functname parameterlist

The parameters are then identified as $1 …$9 where they can be used as read only variables

To declare variables inside the function use

```
typeset var_list
```

Function return value used to signify its status, normally set return 1 or return 0

```
#!/bin/bash
function affi    #use keyword function followed by it name
{                #encapsulate function definition with {
  typeset reply  #declare variable reply
  while true     #loop indefinitely true --keyword
  do             #loop
    read –p “$1” reply  #read value for reply displaying 1st
argument
    case $reply in   #use case test value of reply
      y|yes) return 0;; #if y or yes return 0
      n|no)  return 1;; #if n or no  return 1
      *)  echo “please type y or n”;; #if any string print
message
    esac  #end of case
  done    #end of while
} #end of function definition
#
#while call function with argument $1
while affi 'Do you want to continue?'
do
 echo "ok"  #if affi returns 0 print ok
done
```

```
./func1.sh
Do you want to continue? yesy
please type y or n
Do you want to continue? yes
ok
Do you want to continue? nop
please type y or n
Do you want to continue? no
[root@localhost /root]#
```

```
#!/bin/bash
function fun2
{
  typeset a1 a2                #declare two var a1 a2
  echo "type in 2 numbers"
  read a1 a2                   #read from keyboard two values
  echo "$1 $2 $a1 $a2” #display values of 2 arguments $1 $2 and vrs a1,a2
  (( sum = $a1 + $a2 + $1 + $2 ))  #calculate the sum of args + vars
  echo "the sum is $sum" #print out value of sum
  if [ $sum = 0 ]          #test if sum equals 0
    then
      echo "ret 0"         #if true function returns 0
      return 0
    else
      echo "ret 1"         #if false function returns 1
      return 1
  fi
} #end of function def
#
if fun2 0 1  #call fun2 $1 = 0 $2 = 1
 then
  echo "sum = 0"   #if fun2 returns 0
 else
  echo "sum not 0" #if fun2 returns 1
fi
fun2 4 5  #call fun2 with $1=4 and $2=5
echo "The value sum after function call is $sum"
echo "end"; exit 0
```

```
./func2.sh 4 5
type in 2 numbers
0 -1
0 1 0 -1
the sum is 0
ret 0
sum = 0
type in 2 numbers
6 7
4 5 6 7
the sum is 22
ret 1
The value sum after function call is 22
end
```

String Manipulation

Converting strings to lowercase or uppercase

Use the **tr** command.

Example:
% echo "a string" | tr “[a-z]” “[A-Z]”
A STRING

Determining the length of a string

${#string }

Example:
% mystrr=hello
% echo ${#mystr}
5

Extracting a substring from a string

Syntax:
${string :position :length }

string --- character string
position --- starting position
length --- no. of chars to be copied

Example:
% mystr=hello world
% echo ${mystr:4:5}

Position h = 0 e = 1 l = 2 etc.. include blank space

so will copy 5 char from position 4

returning o wor

Removing substrings from strings

Removing the shortest match of $substring from the front of $string.

Syntax:
${string#substring}

```
numstr=123456789011121314 15
echo ${numstr#1*5}
```

Will remove the matched 1 char followed by any no. of chars to the 5 character ; displays 67890112131415

Extracts the longest match of $substring from the front of $string.

Syntax:

${string##substring}

Example

```
$ str1="123456789012345678 9abc"
$ echo ${str1##1*9}
abc
$
```

To extract the shortest match of $substring from the end of $string.

Syntax

```
${string%substring}
```

example

```
$ str1="12345678901234567890abc"
$ echo ${str1%a*c}
12345678901234567890
$
$ echo ${str1%8*5}
12345678901234567890abc
```

No match so returns original string

To extract the longest match of $substring starting from the end of $string.

synatx

${string%%substring}

Example

```
$ str1="123456789012345678900abc“
$ echo ${str1%%0*c}
123456789
$
```