# UNX510/DPS918 - Unix BASH Shell Scripting

## Lecture 10 - Arrays; Option Parsing

### Arrays

* one-dimensional, index starts at zero
* may be declared implicitly, or using declare -a
* elements may be initialized separately:
* array[0]=hi
* array[1]=hello
* array[2]="how are you"
* multiple elements may be initialized at once: array=(hi hello "how are you")
* indices may also be specified: array=([0]=hi [1]=hello [2]="how are you")
* arrays may be sparse: array=(hi [10]=hello [20]="how are you")
* array elements are referenced as follows: ${array[$index]}
* if "$index" is negative, array is indexed from the end, so "${array[-1]}" refers to the last element
* the number of elements in the array is: ${#array[\*]} or ${#array[@]}
* all array values may be referenced at once: ${array[\*]} or ${array[@]}
* all array indices may be referenced at once: ${!array[\*]} or ${!array[@]}
* some examples:
* ==> b[0]="First element"
* ==> b[1000]="Second element"
* ==> echo ${#b[\*]}
* 2
* ==> echo ${b[\*]}
* First element Second element
* ==> echo ${!b[\*]}
* 0 1000
* ==> echo ${b[1000]}
* Second element
* ==> echo ${b[-1]}
* Second element
* ==> echo ${b[-1000]}
* ==> echo ${b[-1001]}
* First element
* ==> echo ${b[-1002]}
* -bash: b: bad array subscript

==> \_

* array elements or entire arrays can be removed by using unset, for example:
* ==> echo ${#b[\*]}
* 2
* ==> unset b[0]
* ==> echo ${#b[\*]}
* 1
* ==> unset b
* ==> echo ${#b[\*]}
* 0

==> \_

* arrays may be processed within a "for" loop, for example:
* ==> zoo=(lion tiger "bear, oh my")
* ==> for animal in ${zoo[\*]}; do echo $animal; done
* lion
* tiger
* bear,
* oh
* my
* ==> for animal in "${zoo[\*]}"; do echo $animal; done
* lion tiger bear, oh my
* ==> for animal in "${zoo[@]}"; do echo $animal; done
* lion
* tiger
* bear, oh my
* ==> for animal\_num in ${!zoo[@]}; do echo $animal\_num; done
* 0
* 1
* 2
* ==> for animal\_num in ${!zoo[@]}; do echo ${zoo[$animal\_num]}; done
* lion
* tiger
* bear, oh my

==> \_

* here is an example of using arrays to work with filenames which may contain special characters:
* ==> touch file{1..5} 'file 6'
* ==> ls
* file 6 file1 file2 file3 file4 file5
* ==> files=$(ls file\*)
* ==> echo $files
* file 6 file1 file2 file3 file4 file5
* ==> rm $files
* rm: cannot remove `file': No such file or directory
* rm: cannot remove `6': No such file or directory
* ==> ls
* file 6
* ==> touch file{1..5}
* ==> ls
* file 6 file1 file2 file3 file4 file5
* ==> files=(file\*)
* ==> echo ${files[@]}
* file 6 file1 file2 file3 file4 file5
* ==> rm "${files[@]}"
* ==> ls

==> \_

### Associative Arrays

* one-dimensional, indices are strings called "keys"
* must be declared explicitly using declare -A
* elements may be initialized separately:
* array[frog]=tadpole
* array[bear]=cub
* array[duck]="ugly duckling"
* multiple elements may be initialized at once: array=([frog]=tadpole [bear]=cub [duck]="ugly duckling")
* array elements are referenced as follows: ${array[$key]}
* the number of elements in the array is: ${#array[\*]} or ${#array[@]}
* all array values may be referenced at once: ${array[\*]} or ${array[@]}
  + order is unpredictable
* all array keys may be referenced at once: ${!array[\*]} or ${!array[@]}
  + order is unpredictable
* an example of an associative array:
* ==> declare -A numbers
* ==> numbers[zero]=0
* ==> numbers[one]=1
* ==> echo ${#numbers[\*]}
* 2
* ==> numbers=([zero]=0 [one]=1 [three]=3 [four]=4 [five]=5 [six]=6 [seven]=7 [eight]=8)
* ==> echo ${#numbers[\*]}
* 8
* ==> echo ${numbers[\*]}
* 4 7 1 5 0 6 8 3
* ==> echo ${!numbers[\*]}
* four seven one five zero six eight three
* ==> echo ${numbers[three]}
* 3

==> \_

* associative arrays may be processed with the "for" loop, for example:
* ==> declare -A numbers
* ==> numbers=([zero]=0 [one]=1 [two]=2 [three]=3 [four]=4 [five]=5 [six]=6 [seven]=7 [eight]=8)
* ==> for number in ${!numbers[\*]}; do echo $number; done
* four
* seven
* one
* five
* zero
* six
* two
* eight
* three
* ==> for number in ${numbers[\*]}; do echo $number; done
* 4
* 7
* 1
* 5
* 0
* 6
* 2
* 8
* 3
* ==> for number in ${!numbers[\*]}; do echo "element $number is ${numbers[$number]}"; done
* element four is 4
* element seven is 7
* element one is 1
* element five is 5
* element zero is 0
* element six is 6
* element two is 2
* element eight is 8
* element three is 3

==> \_

* of course the keys may be sorted however we want, for example:
* ==> cat sort\_alpha
* declare -A numbers
* numbers=([zero]=0 [one]=1 [two]=2 [three]=3 [four]=4 [five]=5 [six]=6 [seven]=7 [eight]=8)
* for number in $(echo ${!numbers[\*]} | tr ' ' '\n' | sort)
* do
* echo "element $number is ${numbers[$number]}"
* done
* ==> sort\_alpha
* element eight is 8
* element five is 5
* element four is 4
* element one is 1
* element seven is 7
* element six is 6
* element three is 3
* element two is 2
* element zero is 0

==> \_

### Option Parsing

### "getopts" shell command

* usually used in a loop, will find the next option if there is one
* returns exit status of 0 if successful in finding an option, exit status of 1 otherwise
* a simple example: getopts abc opt
  + will find options -a, -b, and -c, in one or more option strings
* the next option found will be placed into the specified variable, "opt" in this case, without the "-" sign
* an example:
* ==> cat options
* while getopts abc opt
* do
* case $opt in
* a) aflag=1 ;;
* b) bflag=1 ;;
* c) cflag=1 ;;
* esac
* done
* if [ "$aflag" = 1 ]
* then echo "You entered option -a."
* fi
* if [ "$bflag" = 1 ]
* then echo "You entered option -b."
* fi
* if [ "$cflag" = 1 ]
* then echo "You entered option -c."
* fi
* ==> options a
* ==> options -a
* You entered option -a.
* ==> options -b -bxa
* ./options: illegal option -- x
* You entered option -a.
* You entered option -b.
* ==> options -b -- -a
* You entered option -b.

==> \_

* note that -- ends the option strings, anything following can still be accessed as an argument

### Keyword Options

* in the following example, the colon after "b" indicates that the -b option takes a keyword, placed into special variable "OPTARG":
* ==> cat options
* while getopts ab:c opt
* do
* case $opt in
* a) aflag=1 ;;
* b) bflag=$OPTARG ;;
* c) cflag=1 ;;
* esac
* done
* if [ "$aflag" = 1 ]
* then echo "You entered option -a."
* fi
* if [ "$bflag" != "" ]
* then echo "You entered option -b with keyword \"$bflag\"."
* fi
* if [ "$cflag" = 1 ]
* then echo "You entered option -c."
* fi
* ==> options -b xyz
* You entered option -b with keyword "xyz".
* ==> options -bxyz
* You entered option -b with keyword "xyz".
* ==> options -ab
* ./options: option requires an argument -- b
* You entered option -a.
* ==> options -cba
* You entered option -b with keyword "a".
* You entered option -c.

==> \_

### Handling Option Errors

* in the following example, the colon at the beginning of the option list puts "getopts" into quiet mode, with no default error messages:
* ==> cat options
* while getopts :ab:c opt
* do
* case $opt in
* a) aflag=1 ;;
* b) bflag=$OPTARG ;;
* c) cflag=1 ;;
* esac
* done
* if [ "$aflag" = 1 ]
* then echo "You entered option -a."
* fi
* if [ "$bflag" != "" ]
* then echo "You entered option -b with keyword \"$bflag\"."
* fi
* if [ "$cflag" = 1 ]
* then echo "You entered option -c."
* fi
* ==> options -x
* ==> options -b

==> \_

* preferable to default error messages, we can create our own:
  + if an invalid option is present, the variable "opt" will get character "?", and "OPTARG" will get invalid option
  + if a required keyword is not present, the variable "opt" will get character ":", and "OPTARG" will get option
* in the following example, note that the "?" must be escaped, otherwise it would be seen as a pattern matching any character:
* ==> cat options
* while getopts :ab:c opt
* do
* case $opt in
* a) aflag=1 ;;
* b) bflag=$OPTARG ;;
* c) cflag=1 ;;
* :) echo "$(basename $0): option -$OPTARG requires a value" >&2
* exit 2 ;;
* \?) echo "$(basename $0): unknown option -$OPTARG" >&2
* exit 2 ;;
* esac
* done
* if [ "$aflag" = 1 ]
* then echo "You entered option -a."
* fi
* if [ "$bflag" != "" ]
* then echo "You entered option -b with keyword \"$bflag\"."
* fi
* if [ "$cflag" = 1 ]
* then echo "You entered option -c."
* fi
* ==> options -x
* options: unknown option -x
* ==> options -b
* options: option -b requires a value
* ==> options -axb
* options: unknown option -x

==> \_

* special variable "OPTIND" will contain position of first non-option argument, so that options may be shifted away:
* ==> cat options
* while getopts :ab:c opt
* do
* case $opt in
* a) aflag=1 ;;
* b) bflag=$OPTARG ;;
* c) cflag=1 ;;
* :) echo "$(basename $0): option -$OPTARG requires a value" >&2
* exit 2 ;;
* \?) echo "$(basename $0): unknown option -$OPTARG" >&2
* exit 2 ;;
* esac
* done
* echo "Before shift, \$# is $#, \$\* is $\*"
* shift $((OPTIND - 1))
* echo "After shift, \$# is $#, \$\* is $\*"
* if [ "$aflag" = 1 ]
* then echo "You entered option -a."
* fi
* if [ "$bflag" != "" ]
* then echo "You entered option -b with keyword \"$bflag\"."
* fi
* if [ "$cflag" = 1 ]
* then echo "You entered option -c."
* fi
* ==> options -abxyz -cab abc lion tiger bear
* Before shift, $# is 6, $\* is -abxyz -cab abc lion tiger bear
* After shift, $# is 3, $\* is lion tiger bear
* You entered option -a.
* You entered option -b with keyword "abc".
* You entered option -c.

==> \_

### Shortcuts For Multiple Options

* in complex scripts, there may be many options:
* while getopts :abek:moprt opt
* do
* case $opt in
* a) aflag=1 ;;
* b) bflag=1 ;;
* e) eflag=1 ;;
* k) karg=$OPTARG ;;
* m) mflag=1 ;;
* o) oflag=1 ;;
* p) pflag=1 ;;
* r) reverse=-r ;;
* t) tflag=1 ;;
* :) echo "$(basename $0): option -$OPTARG requires a value" >&2
* exit 2 ;;
* \?) echo "$(basename $0): unknown option -$OPTARG" >&2
* exit 2 ;;
* esac

done

* some of the repetition can be eliminated using eval :
* while getopts :abek:moprt opt
* do
* case $opt in
* [abemopt]) eval ${opt}flag=1 ;;
* k) karg=$OPTARG ;;
* r) reverse=-r ;;
* :) echo "$(basename $0): option -$OPTARG requires a value" >&2
* exit 2 ;;
* \?) echo "$(basename $0): unknown option -$OPTARG" >&2
* exit 2 ;;
* esac

done

* eval constructs a command from it's arguments, which is then read and executed by the shell:
* ==> opt=a
* ==> ${opt}flag=1
* If 'aflag=1' is not a typo you can use command-not-found to lookup the package that contains it,
* like this:
* cnf aflag=1
* ==> eval ${opt}flag=1
* ==> echo $aflag
* 1

==> \_

* alternatively, shell arithmetic can be used:
* while getopts :abek:moprt opt
* do
* case $opt in
* [abemopt]) ((${opt}flag = 1)) ;;
* k) karg=$OPTARG ;;
* r) reverse=-r ;;
* :) echo "$(basename $0): option -$OPTARG requires a value" >&2
* exit 2 ;;
* \?) echo "$(basename $0): unknown option -$OPTARG" >&2
* exit 2 ;;
* esac

done

* the shell will perform the variable substitution before the arithmetic is evaluated:
* ==> opt=b
* ==> ((${opt}flag = 1))
* ==> echo $bflag
* 1

==> \_