Unix Scripting

Week10

Agenda

- Array in Shell Scripting
  - How to declare
  - How to process
- Parsing Program Arguments, Option Parsing
  - getops

Introduction to Arrays

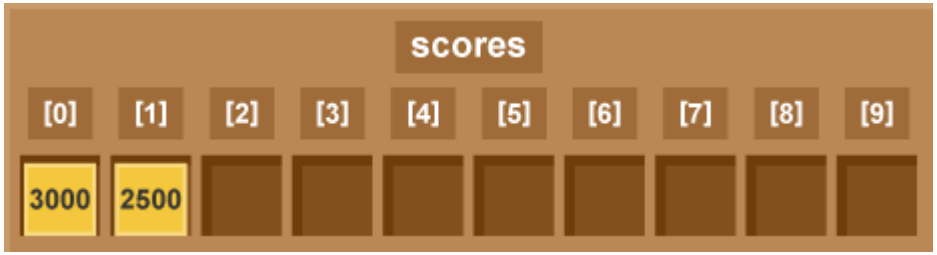
- A **variable** is a memory location that can store a value. It can be thought of as a box in which values are stored. The value held in the box can change, or vary. But each variable can only hold one item of data.
- An array is a series of memory locations – or 'boxes' – each of which holds a single item of data, but with each box sharing the same name. All data in an array must be of the same **data type**.

Array

- Arrays are like shelves

Array

- For example, imagine that a score table in a game needs to record ten scores.
  - Var Score1, Score2, …, Score10
- Instead of having ten variables, each holding a score, there could be one array that holds all the related data:
  - score(10)



Array

- An array is a variable containing multiple values.
- Using the following syntax to declare an array:
  - **ARRAY[INDEX]=value**
- Explicit declaration of an array is done using the **declare** built-in:
  - **declare -a ARRAYNAME**
- Array variables may also be created using compound assignments in this format:
  - **ARRAY=(value1 value2 ... valueN)**

How to access to an array element

- In order to refer to the content of an item in an array, use curly braces.
- **ARRAY=*(one two three)***
- To display first array element
  - ***echo ${ARRAY[0]}***
  - Array indexing always start with 0.
- To display all array elements
  - **echo ${ARRAY[*]}**

Shell Parameter Expansion

- The basic form of parameter expansion is ${*parameter*} which we used it in array dereferencing
- The basic form of parameter substitution is {parameter/pattern/string}

- ${*parameter*:*offset*:*length*}
  - This is referred to as Substring Expansion. It expands to up to *length* characters of the value of *parameter* starting at the character specified by *offset*.

Read Array from Input

- We can also read/assign values to array during the execution time using the *read* shell-builtin.
- `read -a array`
- Upon executing the above statement inside a script, it waits for some input. We need to provide the array elements separated by space (and not carriage return)

Various Operations on Arrays

- array=(apple bat cat dog elephant frog)
- #print first element
- `echo ${array[0]}`
- `echo ${array:0}`

Various Operations on Arrays

- array=(apple bat cat dog elephant frog)
- #display all elements
- `echo ${array[@]}`

Various Operations on Arrays

- `array=( apple bat cat dog elephant frog )`
- What does the following command do?
- `echo ${#array[0]}`
  - #length of first element
- `echo ${#array}`

Various Operations on Arrays

- array=(apple bat cat dog elephant frog)
- `echo ${array[@]:1}`
  - #display all elements except first one
- `echo ${array[@]:1:4}`
  - #display elements in a range

Various Operations on Arrays

- array=(apple bat cat dog elephant frog)
- `echo ${array[@]/a/A}`
  - #replacing substring

Traverse Array Elements using Loop

- To traverse through the array elements we can also use for loop.

```
for i in "${array[@]}"
   do
   #access each element as $i. . .
   done
```

Command Substitution with Arrays

- Command substitution assigns the output of a command or multiple commands into another context.
- Here in this context of arrays we can insert the output of commands as individual elements of arrays. Syntax is as follows.
  - `array=( $(command) )`
- Try this:
  - `array=($(date))`
  - `echo ${array[@]}`

Associative array

- The array that can store string value as an index or key is called associative array.
- An associative array can be declared in bash by using the **declare** keyword and the array elements can be initialized at the time of array declaration or after declaring the array variable.
  - `declare -A assocArray1`
    `assocArray1[fruit]=Mango`
    `assocArray1[bird]=Cockatail`
    `assocArray1[flower]=Rose`
    `assocArray1[animal]=Tiger`

Accessing the Associative Array

- Array elements of an associative array can be accessed individually or by using any loop.
- echo ${assocArray1[bird]}
- for key in "${!assocArray1[@]}"

```
do
    echo $key
done
echo "${!assocArray2[@]}"
```

Example

- ```
  declare -A assocArray2=
  ```

  ```
  ( [HDD]=Samsung [Monitor]=Dell [Key
  board]=A4Tech )
  ```
  ```

Parsing Program Arguments

- A common task in shell scripting is to parse command line arguments to your script.
- Bash provides the **getopts** built-in function to do just that.
  - **`getopts`** *`optstring name [arg ...]`*
- The **getopts** function takes three parameters.
  - The first is a **specification (optsting)** of which options are valid, listed as a sequence of letters
    - if the script recognizes **-a**, **-f** and **-s**, *optstring* is ***`afs`***
  - The *name* on the **getopts** command line is the name of a shell variable. Each time you invoke **getopts**, it obtains the next option from the *positional parameters* and places the option letter in the shell variable *name*.
  - The third argument to getopts is the **list of arguments and options** to be processed.

Example

```
while getopts ":ht" opt; do
   case ${opt} in
      h ) # process option h
         ;;
      t ) # process option t
         ;;
      \? ) echo "Usage: cmd [-h] [-t]"
         ;;
   esac
done
```

the string 'ht' signifies that the options -h and -t are valid.

opt will hold the value of the current option that has been parsed by getopts.

3rd argument of getopts

- When not provided, this defaults to the arguments and options provided to the application ($@). You can provide this third argument to use getopts to parse any list of arguments and options you provide.

Parsing options with arguments

- When **getopts** obtains an option from the script command line, it stores the index of the next argument to be processed in the shell variable **OPTIND**.
- When an option letter has an associated argument (indicated with a : in *optstring*), **getopts** stores the argument as a string in the shell variable **OPTARG**.
  - If an option doesn't take an argument, or **getopts** expects an argument but doesn't find one, **getopts** unsets OPTARG.

OPTIND and OPTARG

- **ENVIRONMENT VARIABLES**
  - ***OPTARG*** stores the value of the option argument found by **getopts**.
  - ***OPTIND*** contains the index of the next argument to be processed.

Examples

```
while getopts ":t:" opt; do
  case ${opt} in
    t )
      target=$OPTARG
      ;;
    \? )
      echo "Invalid option: $OPTARG" 1>&2
      ;;
    : )
      echo "Invalid option: $OPTARG requires an argument"
1>&2
      ;;
  esac
done
shift $((OPTIND -1))
```

Shifting processed options

- The variable **OPTIND** holds the number of options parsed by the last call to getopts.
- It is common practice to call the ***shift command*** at the end of your processing loop to remove options that have already been handled from **$@**.

Notes

- The special option of two dashes ("**--**") is interpreted by **getopts** as the end of options.
- By default, **getopts** will report a verbose error if it finds an unknown option or a misplaced argument. It also sets the value of *optname* to a question mark ("**?**"). It does not assign a value to **$OPTARG**.
  - If the option is valid but an expected argument is not found, *optname* is set to "**?**", **$OPTARG** is unset, and a verbose error message is printed.
- If you put a colon at the beginning of the *optstring*, getopts runs in "silent error checking mode." It will not report any verbose errors about options or arguments, and you need to perform error checking in your script.

getopt vs getopts

- There is also the external utility **getopt** , which parses long-form arguments, like "--filename" instead of the briefer "-f" form
- **getopt**’s traditional versions can’t handle empty argument strings, or arguments with embedded whitespace

Ref:

- https://sookocheff.com/post/bash/parsing-bash-script-arguments-with-shopts/#:~:text=A%20common%20task%20in%20shell,getopts%20function%20takes%20three%20parameters.
- https://www.computerhope.com/unix/bash/getopts.htm