`bash` Functions and Arrays

Functions

- More efficients than scripts as they are kept in memory while the script has to be read in
- Help in organizing long scripts into manageable modules
- Defined by one of the two methods (no functional difference between the two forms):
  1. Method 1

     ```
     function fn
     {
         shell commands
     }
     ```

  2. Method 2

     ```
     fn()
     {
         shell commands
     }
     ```

```
#!/bin/bash

hello()
{
    if [ $# -eq 0 ]
    then
        echo "Hello World"
        return
    fi

    echo "Hello $*"
    return
}

hello
hello John
```

- A function must be defined in a script before it is being used
- The functions in the environment can be seen in alphabetical order) by the command `declare -f`
- A function can be deleted by `unset -f fn`
- Positional parameters in functions work exactly as they do in shellscripts
  - Positional parameters are local to the functions
  - `$0` stays the same inside a function because functions execute in the environment of shellscript
- Local variables
  - Variables defined inside a function are local to the function and override a variable with the same name in the calling script
  - The local variables can be declared by using the keywork `local` (a good practice)

  - The variables not defined with `local` automatically are declared with global scope

- Return values
  - The return values can be used to return a value to the calling function; however, that is not a good way to do this
    - The return value in calling function will have to be captured in `$?`
    - The range of the return value is limited to a max of 255
  - The shell way to do this is to print the value

```
product()
{
    echo $(( $1 * $2 ))
}
```

- Overriding commands
  - You can override the commands by using the keyword `command`

```
function ls
{
    command ls -l
}
```

  - Absence of the keyword `command` will make the function call itself recursively

Command Precedence

- Commands are executed in the following order of precedence
  1. Aliases
  2. Keywords (`function`) and other control statements
  3. Functions
  4. Built-in commands such as `cd` and `type`
  5. Scripts and executables, searched through `PATH`
- The exact version of command used can be found by `type`

```
type ls
type -all ls
```

Arrays in `bash`

- `bash` has two types of arrays: one-dimensional indexed arrays and associative arrays
- Any variable can be used as a 1D aray
  - Identified as `var[index]`

Indexed arrays

- Index array need not be declared though they can be using the command `declare -a`
- You can also declare arrays of any size by

```
declare -a color[3]
```

with the elements indexed from 0 to 2

- In reality, the index above is ignored
- You can simply add more elements to it by assigning an element with a new index

- Accessed by an index
  - Index starts at 0
  - Index can be any positive integer, up to 599147937791
  - Arithmetic expressions are supported in index
- Values are assigned by

```
var[index]=value
```

- Examples

```
color[1]="red"
color[2]="green"
color[0]="blue"
```

  - Values need not be assigned in any specific order
- Another way to assign values, known as *compound statement*, is

```
color=([2]=green [1]=red [0]=blue)
```

- If you specify the elements in order, you can specify them as

```
color=(red blue green)
```

- The above values can be accessed by

```
for i in 0 1 2
do
    print ${color[$i]}
done
```

  - You must use the curly braces to access array elements; otherwise, you'll just get the first array element and the subscript
- If you specify an index at some point in compound assignment, the values get assigned in consecutive locations from that point on

```
color=(red [2]=blue green)
```

  - This array has four elements, with index 1 element null
  - Reassigning to an existing array with a compound statement will lose existing values in array

    ```
    color=([5]=violet yellow)
    ```
- You can see all the values by using `*` or `@`

```
echo ${color[*]}
```

  - `*` expands the array to one string with values separated by first characters of `IFS`
  - `@` expands the array to separate words

- The indices that have been assigned can be listed by `!` as

```
echo ${!color[@]}
```

- The number of elements, or the length of a specific element, can be found by `#`

```
echo ${#color}
echo ${#color[1]}
```

- List all the elements by

```
for i in ${!color[@]}
do
    echo ${color[$i]}
done
```

- Use of double quotes
  - Use of double quotes around the variable makes the array elements appear as one

```
for i in ${color[*]}
do
    echo ${i}
done
```

```
for i in "${color[*]}"
do
    echo ${i}
done
```

- Deleting array elements
  - Any array element can be deleted by using the command `unset`

```
unset color[1]
echo ${color[*]}
echo ${!color[*]}
echo ${#color}
```

- Extracting range of indices
  - You can extract `n` elements starting at index `m` from an array by

```
${array[*]:m:n}
echo ${colors[*]:3:2}
```

- Search and replace an element

```
echo ${colors[*]/red/gold}
```

- You can also use the pattern to remove an item

```
echo ${colors[*]/red/}
```

- Reading an array from a file

```
var=$( cat file )
echo $var
echo ${#var}
echo ${!var[*]}
var=($( cat file ))
echo $var
echo ${#var}
echo ${!var[*]}
```

Associative arrays

- The index can be any arbitrary string
- Associative arrays *must be* declared with `typeset -A` or `declare -A`
- Examples

```
declare -A shade
shade[apple]=red
shade[banana]=yellow
shade[grape]=purple
```

- The above values can be accessed by

```
for i in apple banana grape
do
    print ${shade[$i]}
done
```

- You can access all elements by using $\star$ for the index, for both indexed as well as associative arrays
- Other features of indexed arrays are available as well

```
for i in ${!shade[@]}
do
    printf "%-10s%-10s\n" $i ${shade[$i]}
done
```