5.14 Variables

We've already started working with variables, quite a bit. So far, though, the variables we've seen are from positional parameters and basic variables within the shell scripts, and environment variables from the shell itself. When declaring a variable, you can use the syntax:

```
variableName='someValue'
```

Depending on what falls into the place of "someValue", you can set the variable name as a string value, or a more complicated value such as something returned from an expansion of some kind.

You can additionally use the `declare` keyword to declare placeholders.

- `-i` sets integer values
- `-r` is read only (think for making constants)
- `-a` declares an array type
- `-x` declares value exported to the environment
- `-p` prints the variable's declared purpose

`declareExample.sh`

```
declare -i integerValue
integerValue=23
declare -p integerValue


someValue="I am a string"
declare -r someValue
#someValue="What if I changed?"
declare -p someValue

declare -a array
array="can I attempt to make an array without values?"
array="Another?"
declare -p array

declare -x environmentVar
environmentVar="I AM IN THE ENVIRONMENT NOW!"
declare -p environmentVar

```

5.15 Arrays

When creating an array, you can use the `declare` syntax, however you can implicitly create an array by using:

```
someArrayName=("element0", "element1", "etc")
```

To access the elements, you use the indexed values:

```
echo ${someArrayName[1]}
```

To add a new element into the array, you can use the `+=` syntax:

```
someArrayName+=("element3")
```

Let's look at an how to work with arrays:

`arrayExample.sh`

```
#!/bin/bash

emptyArray=()

emptyArray+=("someElement")
emptyArray+=("otherElement")

echo Echoing emptyArray = $emptyArray
echo Echoing all of the arrays contents: ${emptyArray[*]}
echo Getting the length of the array is with ${#emptyArray[*]}


echo You can add elements to any index of an array, however:
emptyArray[9]="what?"

echo
echo To iterate over an array, you will need to:
for ((i=0; i < ${#emptyArray[*]}; i++ ))
do
        echo element at i$i = ${emptyArray[$i]}
done
echo
echo what happened to \"what?\" ?? We need to access the 9th index:
echo emptyArray\[9\]  = ${emptyArray[9]}
```

```
echo


echo And to reset the indices of an array:
resetArray+=( "${emptyArray[@]}" )

echo empty array length is: ${#emptyArray[*]}
echo Reset array length is ${#resetArray[*]}

for singleElement in "${emptyArray[@]}"
do
        echo single element: $singleElement
done
```

The above prints out:

> Echoing emptyArray = someElement
>
> Echoing all of the arrays contents: someElement otherElement
>
> Getting the length of the array is with 2
>
> You can add elements to any index of an array, however:
>
> To iterate over an array, you will need to:
>
> element at i0 = someElement
>
> element at i1 = otherElement
>
> element at i2 =
>
> what happened to "what?" ?? We need to access the 9th index:
>
> emptyArray[9] = what?
>
> And to reset the indices of an array:
>
> empty array length is: 3
>
> Reset array length is 3
>
> single element: someElement
>
> single element: otherElement
>
> single element: what?

It's also possible to read array elements in with `read`:

`readArrayElements.sh`

```
#!/bin/bash

echo Words separated by a space are now array elements
read -a arr

for value in "${arr[@]}"
do
        echo  the value is: $value
done
```

When running this program we get to enter space delimited array elements:

```
./readArrayElements.sh
```

> Words separated by a space are now array elements
>
> i have a neat array making sentence
>
> the value is: i
>
> the value is: have
>
> the value is: a
>
> the value is: neat
>
> the value is: array
>
> the value is: making
>
> the value is: sentence

5.16 Variable Modifiers

It's possible to use modifiers on top of variables to make your code do more. These modifiers can do multiple things, from checking to see if the variable exists to grabbing specific substrings:

- `${someVariable:-defaultValue}`: This **returns** the default value if `someVariable` doesn't exist (or has no value).
- `${someVariable:=defaultValue}`: This **sets** the value of `someVariable` to `defaultValue` if `someVariable` doesn't exist (or has no value)

- `${someVariable:?errorMessage}`: If `someVariable` doesn't exist, the program exits and prints `errorMessage` to the std error

Existential Modifiers

An example of using the above four is:

`unsetVarExamples.sh`

```
1 #!/bin/bash
2
3 unsetVar1=''
4 unsetVar2=''
5
6
7 var1=${unsetVar1:-somethingElse}
8 echo var1 is $var1
9 echo
10
11 ${unsetVar1:="echo I am new!"}
12 echo unsetVar1 is now $unsetVar1
13 echo
14
15 ${unsetVar3:?"Throw an error and exit!"}
16 echo Did you \exit?
```

Notice in the above that there are two types of variables that we're using that are technically "unset" or "null". First we have the two declared variables that are empty strings. That's just as good as being unset or null. Secondly, these work just the same with variables that have never been declared whatsoever like in line 15.

Substring Modifiers

These variable modifiers allow for us to interact with specific chunks of the variables themselves:

- `${someVariable:offset:length}`: This modifier grabs a substring from `someVariable` staring from the offset and going to length (or the end if no length)
- `${someVariable#pattern}` OR `${someVariable##pattern}` deletes a pattern with matching prefix
- `${someVariable%pattern}` OR `${someVariable%%pattern}` deletes a pattern with the matching suffix
- `${someVariable/pattern/newString}` finds and replaces the substring matching `pattern` with `newString`

`substringModifiers.sh`

```
#!/bin/bash

someVariable="I am a long variable."

echo ${someVariable:7:4}

echo ${someVariable#I am a }

echo ${someVariable%iable.}

echo ${someVariable/long/way\ cool}
```

Seeing this code in action:

```
./substringModifiers.sh
```

> long
>
> long variable.
>
> I am a long var
>
> I am a way cool variable.

It is possible to apply these patterns to arrays. When using the offset/length, you'll wind up with subarray. All other methods will map over the array and apply the string manipulations to each array element.

Parsing Strings

Often times, it's likely that you'll come across a string that needs to be parsed over and split on a specific delimiter. You can do this with the internal field separator, or `IFS`. The general syntax for using `IFS` is done in conjunction with read:

```
IFS=',' #setting space as comma
read -ra arrayOfValuesFromDelimitedString <<<"$commaSeparatedString"
```

Previously we were working on a shell script, `versionBump.sh`. Now that we know how to use arrays, and we know hot to use mathematical expansions, and we know how to use sed. Let's put all of this together for a final working version bump:

```
File version is now 1 0 0
[mjlny2@delmar ch5]$ vim versionBump.sh
#!/bin/bash
RED_COLORATION='\033[0;31m' #red color



versionType=$1
fileName=$2

echo You wish to increment the $versionType version for $fileName

if [[ ! -f ./$fileName ]]; then
    printf "$fileName ${RED_COLORATION}does not exist.${NO_COLORATION}\n"
    exit
fi



currentVersion=$(sed -n 's/VERSION=//p' $fileName)
echo The current version of $fileName is $currentVersion

IFS='.'
read -ra versionArray <<< "$currentVersion"

echo Major ${versionArray[0]}
echo Minor ${versionArray[1]}
echo Patch ${versionArray[2]}

major=${versionArray[0]}
minor=${versionArray[1]}
patch=${versionArray[2]}


case $versionType in
[Mm]ajor)
    echo Major version bump:
    ((major++))
    ;;
[Mm]inor)
    echo Minor version bump:
    ((minor++))
    ;;
[Pp]atch)
    echo Patch version bump:
    ((patch++))
```

```
    ;;
*)  printf "${RED_COLORATION}Bad version bump input.${NO_COLORATION}\n"
    echo The possible input values for version type are: "Major" \| "Minor"
\| "Patch"
    exit
    ;;
esac

newVersion="$major"'.'"$minor"'.'"$patch"

sed -i "s/$currentVersion/$newVersion/" "$fileName"
echo File version is now $newVersion
```

In the above, we've split the version into three separate variables, `major`, `minor`, and `patch`, wherein we then update the version type in the case statement. From there we recreate the version string and then supply a new version string with `sed` and it's `-i` inline flag.

5.18 Functions

We've already seen some functions, but we've only used them to act as a way to quickly print a menu and keep our code mildly uncluttered. Functions act like their own independent scripts. Looking at the syntax again:

```
function someFunc () { commandlist }
```

Parameters:

You may see the function syntax and think it's similar to c-style languages that can define parameters within the parentheses but that will just cause errors. Functions take *positional parameters* just like a full shell script. Take, for example, the script `functionArguments.sh`. We can pass an argument into this script, and then have that script pass an argument of another kind to a function:

`functionArguments.sh`

```bash
1 #!/bin/bash
2 
3 function someFuncWithArgs () {
4   echo the first \function argument is $1
5   echo the second is $2
6 }
7 
8 echo The first argument to the script is $1
9 echo The second argument to the script is $2
10 
11 someFuncWithArgs neato mosquito
```

Now, when executing this script with the following parameters:

```bash
./functionArguments.sh cool beans
```

> The first argument to the script is cool
>
> The second argument to the script is beans
>
> the first function argument is neato
>
> the second is mosquito

Think of functions as their own subscripts. Even though we're using `$1` and `$2` at lines 4,5 and 8,9, depending on the context, those positional parameters are referring to 2 completely different sets of data.

Return Values

Functions all have a return value. All functions (and shell scripts for that matter) have a return value of some integer value 0-255. You can specify which integer value to return with the `return` keyword. However, it's unlikely that you'll always want to return a simple integer (also using the return value for the status for your own purposes is mildly unwise).

There are a number of ways to interact with data from functions. The first is the most convenient in that, it's exactly how we return data to the command line of any shell script: `echo`.

```bash
#!/bin/bash

function noEchoBack() {
  results='some results!'
}

function echoBack() {
  otherResults="echoBack results."
  echo "$otherResults"
```

```
}

firstFuncResults=$(noEchoBack)
secondFuncResults=$(echoBack)

echo first results are: $firstFuncResults
echo second results are: $secondFuncResults

echo but can I also see what is stored \in $results or $otherResults ?
```

You could also consider using global variables to pass yours back. However, global variables can be dangerous if you're not paying attention.

5.19 Redefining Bash Built-In Functions

It is possible to redefine bash built in functions. It's strongly recommended that you don't do this.

5.20 Example Scripts

Please take the time to go through the sample scripts!

5.21 Debugging!

When you are working on a shell script, it's likely the case that you'll run into issues. Luckily, if a command fails to execute because of a syntax error, it will automatically be printed to the shell. However, that doesn't work for logical errors. To see how your program executes, you can trace each of its steps with the `set -x` command to turn tracing on (or `bash -x someScript.sh`). Let's try it with our `versionBump.sh`

```
set -x
./versionBump.sh minor versionedProgram.sh
```

> +./versionBump.sh minor versionedProgram.sh You wish to increment the minor version for versionedProgram.sh The current version of versionedProgram.sh is 2.0.0 Major 2 Minor 0 Patch 0 Minor version bump: File version is now 2 1 0 Filename is still versionedProgram sh File version is now 2 1 0 ++ printf '\033]0;%s@%s:%s\007' mjlny2 delmar '~/2750Materials/module3/ch5'

5.22 Error and Interrupt Handling

In order to ensure that your scripts fail gracefully and can properly assess interrupts, you'll need to properly handle these details.

Error Handling

Depending on the type of error, a script may or may not halt. If your script has a syntax error, then you'll need to fix it. When regular commands fail, however, their output is logged, and the script continues, which is not always ideal. To ensure that these errors are caught, making us of the return values can be helpful.

A good example of when you'd want to gracefully fail is in a deploy script, which typically will bump the version, build the project ,and then deploy it to the cloud (or whatever service it is hosted on).

`failedVersionBump.sh`

```
#!/bin/bash

fileName='someFile.whatever'

echo Bumping $fileName version of $1 and deploying to the cloud.

./versionBump.sh $1 $fileName

echo Build Phase:
#do the building

echo Deployment Phase:
#do the doployment commands
```

In the above code, if we fail on the `versionBump.sh` command, we'll still keep executing. We want everything to run smoothly. We can instead use the return values of programs to ensure that we get to deal with failed executions:

`gacefulFailure.sh`

```
#!/bin/bash

fileName='someFile.whatever'

echo Bumping $fileName version of $1 and deploying to the cloud.

./versionBump.sh $1 $fileName
returnCode=$?

if  (( returnCode==1 ))
then
  echo Error in versioning.
  exit 1
fi

echo Build Phase:
```

```
#do the building

echo Deployment Phase:
#do the doployment commands
```

Interrupt Handling

Interrupt handling is something slightly different. While we haven't covered signals by any significant measure, we have seen the `kill` command (and we've used the `ctrl + c` command). These are signals, which we'll get into significantly more deeply in chapter 11. Here, though we can at least introduce the command `trap` which can take a specific signal and doing something with that signal. The general syntax looks like:

```
trap command signal
```

In this command we can do anything we wish. For now, let's ensure that our extremely important program `countAndWait.sh` can execute without being interrupted:

```
#!/bin/bash

trap "echo The program must complete before interruption or termination." SIGTERM SIGINT

for (( i=0; i<15; i++))
do
        echo $1
        sleep 1
done
```

Now, if we run this program and attempt to interrupt it, we'll see:

```
./countAndWait.sh
```

> the value of 0
>
> the value of 1
>
> the value of 2
>
> ^CThe program must complete before interruption or termination.
>
> the value of 3
>
> ^CThe program must complete before interruption or termination.
>
> the value of 4

> ^CThe program must complete before interruption or termination.
>
> the value of 5
>
> ^CThe program must complete before interruption or termination.
>
> the value of 6
>
> ^CThe program must complete before interruption or termination.
>
> the value of 7
>
> ^CThe program must complete before interruption or termination.
>
> the value of 8
>
> ^CThe program must complete before interruption or termination.
>
> the value of 9
>
> the value of 10
>
> ^CThe program must complete before interruption or termination.
>
> the value of 11
>
> the value of 12
>
> the value of 13
>
> the value of 14