CST8177

bash Scripting

Chapters 13 and 14 in Quigley's
"UNIX Shells by Example"

bash String Operators

- These are called Variable Modifiers and Variable Expansion Substrings in the textbook (Page 823 to 828)
- They are the string operators that rely upon the **`${ ... }`** form of variable expansion.
- One group is indicated by an operator following a colon **`:`** such as **`:x`** to remove the leading **`x`** characters from a string (strings are counted from **`0`**).
- The other string operator group uses **`%`** and **`#`**.
- All are of the general form **`${variable??word}`**

bash String Operators

`${variable:?word}`

`:-`	use word if variable is null or not set, else use variable
`:=`	as above, but also set variable to word
`:+`	use word if variable is set and not null, else nothing (opposite of **`:-`**)
`:?`	print word and exit from the shell if variable is null or not set
`:num`	return the substring from num to the end, counting from 0
`:num:len`	return the substring starting at num for len characters

bash String Operators

`${variable%?word}`
`${variable#?word}`

%	matches the smallest trailing portion of the value of **variable** to **word** and deletes it
%%	same as **%**, but the largest
#	same as **%**, but leading, not trailing
##	same as **%%** but leading

Typical Usages

player=${somevar:-"default"};

- If **player** is undefined it becomes "**default**"

player=${player:2}

- **player** is now "**fault**"

player=${player:3:2}

- **player** is now "**lt**"

bash String Operators

- There is also **${#variable}**, which returns the length of the string in **variable**.
- It can also be used to determine the number of active elements in an array: **${#array[*]}**.
- For **$*** or **$@**, use **$#** for the number of positional parameters.
- The array subscripts * and @ are subtly different:
- * elements in double quotes form a single token (i.e. "a b c d")
- @ elements in double quotes form a list of tokens (i.e. "a" "b" "c" "d")

bash Parameters (parms), Arguments (args), and Arrays

Pages 59, 838, 874 to 877 in the textbook

Arrays

- An array is a collection of items all of the same sort, stored in a single variable. Think, perhaps, of eggs in a 12-element array called an egg carton.
- Arrays count from 0. You will forget this, usually at the worst possible time, so try hard to remember:

Arrays count from zero!

- To declare an array: **`declare -a myArray1`**
- To initialize:

```
declare -a myArray1=(1 2 3 4 5 6)

declare -a myArray2
myArray2=(1 2 3 4 5 6)

declare -a myArray3
read -a myArray3
[stdin] 1 2 3 4 5 6 [ENTER]
```

Arrays: Use and Length

- To use:

```
${myArray[3]}      # one element
${myArray[$i]}     # one element
${myArray[*]}      # all elements
```

- To get the length:

```
declare -a myArray=( 1 4 9 16 25 )
${#myArray[*]}     # returns 5 elements
${#myArray[3]}     # returns 2 characters
```

Setting and Unsetting Arrays

```
declare -a myArray=(1 2 3 4 5 6)
myArray[3]=9        # 1 2 3 9 5 6
myArray[$i]=8       # if i = 0 then
                    # 8 2 3 9 5 6

someArray=( "a” "b” [3]="c” "d” )
   # from index=[0]: "a” "b” - "c” "d”
   # note that element 2 is undefined

unset myArray[1]  # remove a single element
unset myArray     # remove a whole array
```

Wait! There's a problem!

```
#! /bin/bash
declare -a newArray
declare -i i=0
newArray[0]=1
newArray[3]=99
echo newArray has ${#newArray[*]} elements

while (( i < 5 )); do
    printf "[%d]=%s " $i ${newArray[$i]}
    let i++
done
printf "\n"
```

On **stdout:**

```
newArray has 2 elements
[0]=1 [1]=0 [2]=0 [3]=99 [4]=0
```

How can you tell defined and undefined elements apart?

- This expression returns FALSE for a defined element:
  ((${newArray[0]:-"null"} == "null"))
- And it also returns TRUE for an undefined element:
  ((${newArray[1]:-"null"} == "null"))

For example, if we fix it and try again with:

```
if (( ${newArray[$i]:-"null"} == "null" )); then
    val="undef"
else
    val=${newArray[$i]}
fi
printf "[%d]=%s " $i $val
```

On **stdout:**

```
newArray has 2 elements
[0]=1 [1]=undef [2]=undef [3]=99 [4]=undef
```

Repaired script

```
#! /bin/bash
declare -a newArray
declare -i i=0
newArray[0]=1
newArray[3]=99
echo newArray has ${#newArray[*]} elements

while (( i < 5 )); do
    if (( ${newArray[$i]:-"null"} == "null" )); then
        val="undef"
    else
        val=${newArray[$i]}
    fi
    printf "[%d]=%s " $i $val
    let i++
done
printf "\n"
```

Command Line Arguments

- When you run a script from the command line (after turning on its execute permission with **`chmod`** after the first save, and using the explicit **`./`** directory if it's needed), each argument can be used inside your script.

Command Line Arguments

$0	the script name as entered (with the typed path)
$1 to $9	the first 9 positional arguments
$10	not argument 10, it's **$1** with a **0** appended
${10}	argument 10 and so on, more arguments
$#	the number of positional arguments
$* and $@	all positional arguments
"$*"	evaluates to **"$1 $2 $3"**
"$@"	evaluates to **"$1" "$2" "$3"**
set	set options and positional arguments (**$1** etc.); use **$-** to see set options.
set --	unset all positional arguments

Some Special References

$$	the PID of this shell
$-	"**sh**" options currently set
$?	return code from the just-previous command
$!	the PID of the most recent background job

bash Expressions

- There are two forms of logical expressions, the “old kind” (Bourne shell compatible) and the new kind.
- Bourne shell (**sh**) compatible

```
[ $a -eq $b ]
[-e "filename" ]
```

- bash version 2 and up

```
(( $a == $b ))      # numeric
[[ -e "filename" ]] # string
```

- The **((...))** form can also be used in place of the **let** command.
- Hey, what's that **-e** thingie?

Special tests

-n STRING: the length of **STRING** is nonzero
-z STRING: the length of **STRING** is zero
-d FILE: **FILE** exists and is a directory
-e FILE: **FILE** exists
-f FILE: **FILE** exists and is a regular file
-g FILE: **FILE** exists and is set-GID
-k FILE: **FILE** exists and has its sticky bit set
-L FILE: **FILE** exists and is a symbolic link
-O FILE: **FILE** exists and is owned by the effective UID
-r FILE: **FILE** exists and read permission granted
-s FILE: **FILE** exists and has a size greater than zero
-t FD: file descriptor **FD** is opened on a terminal
-u FILE: **FILE** exists and its set-UID bit is set
-w FILE: **FILE** exists and write permission granted
-x FILE: **FILE** exists and exec permission granted

See test(1) for a complete list and description.

Command Expressions

- Also have two forms.
  - The backquote (back-tick) form is supported by all shells.
  - Also supports the popular form derived from the Korn shell: **`$( ... )`**.
- The advantage of the new style is that it can more easily be nested, since no character inside the parentheses is treated in a special manner: escaping is not necessary.

```
declare -a files=($(ls $(echo $HOME)))
declare -a files=(`ls \`echo \$HOME\``)
```

Some Useful Quotations

"The only way to learn a new [scripting] language is by writing [scripts] in it."

- Brian Kernighan

"Debugging is twice as hard as writing the [script] in the first place. Therefore, if you write the [script] as cleverly as possible, you are, by definition, not smart enough to debug it."

-Brian Kernighan

Brian, along with Dennis Ritchie, were the developers of the C programming language. It in turn is the ancestor of most modern programming languages, including Java, C#, C++, and many more.

"Make everything as simple as possible, but not simpler."

- Albert Einstein

Everyone knows Albert, right? Relativity and all that?