CST8177

bash Scripting

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

bash Version Reminder

- The original version of **/bin/bash** was version 1 [GNU bash, version 1.14.7(1)], which does not support all of the features. It remains upward compatible with the original Bourne shell, **/bin/sh**.
- Fedora 14 uses GNU bash, version 4.1.7(1)-release when it's fully up-to-date. If you are using another Linux system, make sure you are using a reasonably recent version of **bash**.

Some Shell Stuff

- We've been using the bash shell from the command line all semester. However, Chapter 13 is full of details. Be sure to ***read*** and ***re-read*** it carefully, taking notes while you do. Some people even **write** or **highlight** <u>right in their book</u> - if you can believe it!
- Note, in particular, such things as:
  - **`set -x`** and **`set +x`** to turn script expansion tracing on and off;
  - **`set -v`** and **`set +v`** to toggle script line tracing;
  - **`set -n`** and **`set +n`** to toggle script execution;
  - or use **`#! /bin/bash -xvn`** to set them all on;
  - controlling prompt contents (especially, but not only, **`PS1`** and **`PS2`**);
  - built-in commands like **`alias`**, **`dirs`**, **`help`**, **`popd`**, **`pushd`**, and **`type`**;
  - And much more! It's a big chapter.

Structured Scripts

We will take advantage of some basic CS concepts:

- **Structure Theorem**: it is possible to write any computer program using only three control structures:
  - Sequence - executing one statement after another
  - Selection - choosing between two actions
  - Repetition - repeating a sequence of instructions while a condition is true.
- **Top-down development**:
  - Incorporate the control structures into a modular design
  - Start at a top level and break down a problem into a hierarchy of increasingly refined procedures
  - The scripts and functions are the modules of the program

Order of Processing

- The verb is the first token on the command line at the prompt or within a script. Since it can be many different items, it's processed in this order:
    - Aliases
    - Keywords
    - Functions
    - Built-ins
    - Executables
- What conclusions can you draw from this sequence?

Order of Processing

- The verb is the first token on the command line at the prompt or within a script. Since it can be many different items, it's processed in this order:
    - Aliases
    - Keywords
    - Functions
    - Built-ins
    - Executables
- What conclusions can you draw from this sequence?
    - You can replace a shell built-in command with an alias or a function;
    - You can also replace an executable program with an alias or a function.
    - Consider how each of these could be useful.

bash Variables (pages 810 – 831)

Normally, variables are local to the current shell, but they can be exported to the environment (**`export`** built-in) so they can be globally available to all sub-shells as well.

By convention, global names are **`UPPER CASE`** while local variables use **`mixedCase`** (also known as **`camelCase`** because there's a hump in the middle) or **`under_scores`**.

Variables ought to be declared with the **`declare`** built-in even though it's not strictly necessary. I recommend that <u>all arrays</u> and <u>key variables</u> be declared, but a loop control variable (conventionally **`i`**, **`j`**, and **`k`**) need not be.

In addition, there are some predefined variables created upon entry to a script. The command-line arguments are referred to as **$0**, which is the script name and path (you can use **$(basename $0)** and **$(dirname $0)** to separate them), **$1** to **$9** for the first 9 arguments, and **${10}** and up for more (the **{}** are required for more than 1 digit; you can also use **${1}** instead of **$1** if you want to be completely consistent).

The entire list of arguments from **$1** on up can be obtained from **$@** (space-separated) or **$*** (technically, separated by **$IFS**, the inter-field separator; it's usually a space as well). You can find out how many arguments there are from **$1** up by using **$#**.

Finally, **$?** returns the exit value of the most recently completed command or a script exit value. It can be handy for determining the success or failure of a command, but remember than **<u>any</u>** intervening command will change it.

It terms of the planning and design of your solution, you are to list your key variables in a Data Dictionary, one for each script or function that you write. You don't need to define any of the predefined variables, nor any casually used variables such as a loop control.

Name	**Type**	**Range**	**Purpose**
len	-i	0 to 100	length of string
my_str	string	N/A	the string
list	-r string	N/A	list of char

There need be no PDL for a variable, since the Data Dictionary addresses that. There are instances where you may wish to show where and how a variable is initialized, however.

Declaring bash Variables

You should declare all variables you create, using the built-in **declare** command and its options:

-a	the variable is an array
-i	the variable is an integer, not a string
-r	the variable is read-only (a constant)
-x	export the variable to the environment

You can also define read-only variables with **readonly**, and local variables (not for export) inside a function with **local**. You can assign an initial value at the time you declare a variable.

Some Variable Examples

```
declare -ai myArray             # numeric array
declare -r  pi=3.1415926        # constant float
readonly    maxsize=99          # constant integer
declare     my_string='x'       # local string
local       anotherString       # local string
declare -i  len=0               # local integer
declare -ix DEPTH=37            # global integer
```

You can use the **$** operator (or **${...}**)and the **echo** command to inspect these:

```
echo "pi is $pi and maxsize is ${maxsize}"
```

Note that the variable **pi** above is actually a <u>string</u>, and not a <u>number</u>. If you try to use it as a number, it will produce an invalid operator error for the '**.**'.

Screen Output: echo

- Page 1066 in the textbook
- A quick way to have a look at variables is to use **echo** with some useful escape sequences:

  echo [-e][-n][-E] [argument ...]

  - **-n** prevents a newline at the end of the **echo**
  - **-e** permits the use of escape codes
  - **-E** disables escape codes (default).
- The full list of escape codes is in the textbook, but the most useful probably include
  - **\a** alert
  - **\c** no newline (used at end, same as **-n**)
  - **\n** newline right here
  - **\t** tab, for formatting output

Screen Output: `printf`

- Pages 821 - 823
- You have much more control with the **`printf`** command:

  `printf format-string [argument …]`

- Notice that the **`format-string`** is required, but the list of **`arguments`** is not
- The format string uses special **`printf`** codes to format data (see printf(1) and the section of printf(3) titled "Format of the format string"), such as:
  - **`%i`** display an integer value
  - **`%s`** display a string
  - **`%c`** display a single character
- Plus the escape codes like **`\n`** and **`\t`** as with **`echo`**.

printf formats

The **format-string** can be a variable or a string, but it mostly consists of normal characters which are simply copied to **stdout**, escape sequences (which are converted and copied to **stdout**), and format specifications, each of which causes printing of the next successive argument.

In addition to the standard printf(3) formats (see_both: <**http://www.daemon-systems.org/man/printf.1.htm**l> <**http://www.gnu.org/software/bash/manual/bashref.html**>), **%b** means to expand escape sequences inside the corresponding argument, and **%q** means to quote the argument in such a way that it can be reused as shell input (for example **** becomes ****).

printf Examples

Normal **printf**

```
Prompt$ printf "%s\n" "ab\ncd"
ab\ncd
```

Quoting **printf**

```
Prompt$ printf "%q\n" "ab\ncd"
ab\\ncd
```

Backslash **printf**

```
Prompt$ printf "%b\n" "ab\ncd"
ab
cd
```

Variable **printf**

```
Prompt$ declare fred="stuff"
Prompt$ printf "%s\n" $fred
stuff
```

PDL for output

Since the PDL is explaining **WHAT** and **WHY**, in order to describe output simply use a keyword like **`PUT`**, **`WRITE`**, **`DISPLAY`**, or **`SHOW`**. Be reasonably consistent in each scripting project.

PDL: **`PUT error message to stderr`**

Script: **`echo Improper input format >> stderr`**

PDL: **`DISPLAY result of calculations`**

Script: **`printf "%10s has %i and %i\n" acct v1 rc`**

Briefly, make the PDL easy to read and to understand in its context.

Screen Input: read

- Pages 867 – 871 in the textbook
- To read user input in a script, use the **read** built-in command:

```
read [-r][-p xx][-a yy][-e] [zz ...]
```

- where
  - **xx** is a prompt string for **-p**
  - **yy** is an array name for **-a**
  - **zz** is a list of variable names.
- The **-r** option allows the input to contain backslashes, and **-e** allows **vi** editing of the input line.

read Command

`read [-r][-p xx][-a yy][-e] [zz ...]`

- If **`zz`** is a single variable, the whole line of input is placed there when **`ENTER`** is pressed.
  - If there is a second variable name, the first receives the first word and the second the rest of the line, and so on.
  - If there are enough variables, one word is placed into each one in sequence.
  - If there are not enough input words, excess variables are set to the null (also known as the empty) string, **`""`**.
  - If there are no variable names given, the special **`REPLY`** variable receives all the input.
  - The **`read`** command returns **`0`** in **`$?`** (the command exit status special variable) if it terminates normally with a newline (**`ENTER`**), or else **`$?`** is set to **`1`** if **`Control-D`** (end of file for **`stdin`**) or end-of-file (for another file) is found.

read Examples

```
read -p "Enter: " var              # prompt for var

read                               # read into REPLY

read -a tt -p 'Array> '            # prompt for tt

read -p "proceed (Y/N)? " x        # get Yes/No
```

PDL for input

PDL explains **WHAT** and **WHY**. To describe input simply, use a keyword like **GET** or **READ**. It helps the reader if you are fairly consistent in each scripting project.

PDL: **`READ array from file`**

Script: **`read -a my_array < file`**

PDL: **`GET OK to proceed?`**

Script: **`read -p "proceed (Y/N)? " x`**

As with the output, make the PDL easy to read and to understand in its context.

Numeric Operators

Most of the common C-style arithmetic operators (widely used; see also Java, C++, C#, and so on) are available in **bash**, so the integer operations **+ - * / %** can be used, as well as **+= -= *= /= %=** and both the **++** and **--** operators (see Quigley pages 884 - 885 for the complete list or search for **/^arithmetic\ evaluation** in bash(1)).

Float operations must be done in the **bc** or another calculator, or **gawk**, since **bash** does not support float directly. Thus:

```
echo `echo "scale=2; 15/4" | bc`
```

Or, using variables:

```
declare y=15; declare z=4; declare m
m=$(echo "scale=2; $y/$z" | bc)
echo $m
```

Both print the **bash** string **3.75** as the result.

Arithmetic

- Arithmetic is normally done with the **let** command:

```
declare -i x=0

let "x = x + 1"
let x=$x+1
let x+=1
```

- You can enclose an arithmetic expression in **((...))** as well, or in **$[...]** or **$((...))** to have the value returned.
- For example, these are the same. In each case, the **x** variable now holds the value that was displayed:

```
((x=x+1)); echo $x
echo $((x=x+1))
echo $[x=x+1]
```

More Arithmetic

- You may also do arithmetic in array subscripts (arrays come later), so that these examples all have the same effect on **`declare -i x=3`**
  - **`let x+=1`**
  - **`let "x = x + 1"`**
  - **`(( x += 1 ))`**
  - **`echo $[ x = x + 1 ]`**
  - **`echo $(( x += 1 ))`**
  - **`array[x++]="value"`**
- In each case, **`echo $x`** will print the result **`4`**

Arithmetic PDL

You will not normally show the actual calculation, since that's **HOW**. Instead, describe **WHAT** and/or **WHY** you're doing it. For example:

PDL: **`CALCULATE area of room`**
Script: **`let "area = len * width"`**

PDL: **`COMPUTE total price`**
Script: **`(( total = num * unit ))`**

PDL: **`COUNT next iteration of loop`**
Script: **`let i+=1`**