Shell Scripting

and a peek at cron

scripts

- If you find yourself in need of typing in a set of related commands over and over again, then you probably want to write a script of commands only once, and run it many times
- Scripts are a sequence of command lines just like the ones we've been creating and running all through this semester
- But it gets even better: we can use looping and conditional constructs, and more!
- And better still: cron can run our scripts regularly if we want!

Scripting is programming:

- Variables
- Input for programs
- Output from our programs
- Control constructs:
  - looping (e.g. for statements)
  - conditionals (e.g. if statements)

bash Variables

Normally, variables are local to the current shell, but they can be exported to the environment (**export** command) so they can be globally available to all sub-shells as well. By convention, global names are UPPER CASE while local variables use mixedCase or under_scores.

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 (`const`)
`-x`	export the variable

You can also define read-only variables with **readonly**, and local variables inside a function with **local**

Some bash Variable Examples

```
declare -ai myArray   # numeric array
declare -r pi=3.1415926   # constant
readonly e=2.7182818      # constant
declare my_string         # local
local anotherString       # local
declare -x DEPTH=37       # global
```

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

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

Program Output

- A quick way to have a look at variables is to use **`echo`** with some useful escape sequences:

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

- The option **-n** prevents a newline at the end of the echo, while **-e** permits the use of escape codes and **-E** disables escape codes. The list of escape codes is in the textbook, but the most useful are probably **\a** (alert), **\c** (no newline), **\n** (newline), and **\t** (tab).

printf

- You have a lot more control with the **printf** command:

```
printf format-string [argument …]
```

printf (cont'd)

- The **format-string** can be a variable or a string, but it 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.

printf formats

- In addition to the standard **`printf(1)`** formats (see_<http://www.ampl.com/cm/cs/what/ampl/NEW/printf.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 ****).
- Note that **`printf`** is a built-in in **`bash`** version 2, but only a command in **`/usr/bin`** in **`bash`** version 1.

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
```

Program Input

- 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 (**-p**), **yy** is an array name (**-a**), and **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

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) is the only input.

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.
- If there are no variable names given, the special **REPLY** variable receives all the input. If there are not enough input words, excess variables are set to the null string, **""**.

read Examples

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

read                # read REPLY

read a b c
  one two three four five
# c will become "three four five"
```

bash Arithmetic

- Arithmetic can be done with the **let** command:

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

- You can enclose arithmetic expressions in **((...))** as well, or in **$[...]** or **$((...))** to have the value returned.

Control: The IF Statement

The *command* is executed and its return value (the same as **$?**) is evaluated. If the return value is zero, the **then** clause is executed, otherwise it is skipped. If an **elif** clause is next, its *command* is executed and evaluated in the same manner, and so on. If none of the **then** clauses is executed, the **else** clause (if any) is executed.

```
if command
then
        commands
elif command
then
        commands
else
        commands
fi
```

"if" example

- C++ "style" if statement

```
declare -i x=1
if ((x==1))
then
   echo x is 1
fi
```

Or

```
if ((x==1)); then echo x is 1; fi
```

Command Style `if`

- In order to validate a command:

```
if grep "string" file.txt
then
   echo string found in file.txt
elif grep "something" file.txt
   echo something found instead
else
   echo neither string nor something
fi
```

Control: The CASE Statement

- The *variable* is compared with the *values* using the ***shell*** wildcards (**?** * **[…]**) and the Inclusive Or (**|**), **NOT** regular expressions. All the commands are executed for a matching *value* until the ending **;;**. If no *value* matches, then the default ***)** case is executed.

```
case variable in
value1)
        commands
        ;;
value2)
        commands
        ;;
*)
        commands
        ;;
esac
```

Case Example

```
declare var="fred"
case "$var" in
  stuff)
        echo stuff was var
        ;;
  fred)     # this could also be [fF]red or fr?d, etc.
        echo fred was var
        ;;
  *)
        echo default case
        ;;
esac
```

Loops: The FOR Statement

- The **do-done** loop is executed once for each *word* in the *list*, assigning the *word* to the *variable*. If "**in** *wordlist*" is omitted, the positional parameters from **$1** on are used as the *wordlist*.

```
for variable in wordlist
do
        commands
done
```

For Loop Example

```
for x in “abc” “def” “ghi”
do
  cp $x $x’.backup’
done
```

- If the directory contains files named **abc**, **def** and **ghi**, it will contain these file and **abc.backup**, **def.backup** and **ghi.backup.**

Loops: The WHILE and UNTIL statements

- The **do-done** loop is executed as long as *command* returns a value of zero (**$?**).

```
while command
do
        commands
done
```

- **until** is the opposite of **while**, and executes as long as *command* returns a non-zero value.

```
until command
do
        commands
done
```

Something “traditional”

```
declare -i x=0

while (( x<4 ))
do
    let x+=1
    echo $x
done
```

While Loop Advanced Example

```
declare var=“fred”
while echo $var | grep -q “^fred$“
do
   var="Barney“
   echo changing var
done
```

I/O Redirection and Subshells for Loops

- The whole of a loop right down to the done statement can be treated as a unit for redirection and background processing, so that constructs like these are possible:

```
command | loop do … done
loop do … done | command
loop do … done > filename
loop do … done &
```

The BREAK and CONTINUE statements

- **break** operates as in C/C++. That is, it exits from the current loop by executing the statement after the **done**. If the optional *number* is supplied, it breaks out of that many nested loops. Use this with care!

  `break [number]`

- **continue** is also like C/C++, sending control to the command after the **do** statement. It includes a level *number* like **break** which is just as dangerous.

  `continue [number]`

The NULL statement

- The null statement **:** does nothing at all. Use it for empty **then** clauses (for example).

```
if blah
then
  :
else
   # do something really exciting!
fi
```

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, just like **argv[ ]** in C/C++:

Command Line Arguments

`$0`	the script name as entered (with the typed path)
`$1` to `$9`	the first 9 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 positional arguments
`set --`	unset all positional parameters

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

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

- The ((...)) form can also be used in place of the let command.

Command Expressions

- Also have two forms.
  - The backquote form is supported by all shells.
  - Also supports the 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\``)
```

Functions in bash

- Functions are useful, and it's good to see them in bash. You define a function as:

```
function somename { command;
    command; …; }
```

- And you call it by using the name as if it were a normal command.

bash Functions

- A function does not need to return a result, but it can in 3 ways:
  1. set a new value into a variable defined outside the function;
  2. use **return** statement to set the value of **$?**;
  3. write the results to **stdout**.

Functions

- Function scope is from the point it's defined (that is, it must be defined before you can use it). You can define **local** variables to be used only inside the function, but your normal variables can also be used. If you prefer, you can pass arguments into a function as positional parameters (**$1** et al - this is the recommended approach).

Functions

- You will have noticed that traps look like a special form of function.
- To unset a function:

  `unset -f` ***`functionname`***
- To list defined function names:

  `declare -F`
- To list functions and definitions:

  `declare -f`

Functions

- To include pre-written functions in a script (like **#include**):

  `. filename` or `source filename`

- Where in the script would you put this? Why?

- A function is usually written without using **;** as are normal scripts:

```
function somename {
command
command
...
}
```

Shell scripts

- #! /bin/bash
- this line is usually at the top
- it means that the program after the #! should get this file as input
- in other words, bash will run this file as a script
- that way, we can invoke our script with just ./script instead of sh script

Cron (and at)

- Cron is a facility for running programs/scripts according to a schedule
  - example: every day at noon
- at is a program that runs a job (once) at a certain time

Cron

- each user can have a crontab file
- see man 5 crontab
  - there is a crontab(1) and a crontab(5)!

- cron(8) examines cron entries once every minute.

  The time and date fields are:

field	allowed values
minute	0-59
hour	0-23
day of month	1-31
month	1-12 (or names, see below)
day of week	0-7 (0 or 7 is Sun, or use names)

Crontab file (man 5 crontab)

```
# use /bin/sh to run commands, no matter what /etc/passwd says
    SHELL=/bin/sh
    # mail any output to ‘paul’, no matter whose crontab this is
    MAILTO=paul
    #
    CRON_TZ=Japan
    # run five minutes after midnight, every day
    5 0 * * *       $HOME/bin/daily.job >> $HOME/tmp/out 2>&1
    # run at 2:15pm on the first of every month -- output mailed to paul
    15 14 1 * *     $HOME/bin/monthly
    # run at 10 pm on weekdays, annoy Joe
    0 22 * * 1-5    mail -s "It’s 10pm" joe%Joe,%%Where are your kids?%
    23 0-23/2 * * * echo "run 23 minutes after midn, 2am, 4am ..., everyday"
    5 4 * * sun     echo "run at 5 after 4 every sunday"
```