CST8177 – Linux II

More Scripting

Topics

- Output with printf
- Input
  - from a file
  - from a command arguments
  - from the command read

Is stdin a terminal?

- A script can test whether or not standard input is a terminal

```
[ -t 0 ]
```

- What about standard output, and standard error? (hint, 0, 1, 2)
- For example, if we redirect the output of one command into our script , then stdin is not a terminal

The command that does nothing

- Occasionally you'll see a command called, :
- The command is actually colon, ":"
- `: arguments`
- That command expands its arguments and does nothing with them, resulting in a 0 exit status

Testing your scripts

- when you finish a script, you need to run it to verify correct operation
- you're expecting certain things from your script on certain runs
  - example: it expects arguments and you supply none – it should print an error message
  - example: you supply the wrong number of arguments – it should print an error message
- run your script with good input, and bad
  - check that operation is correct for good and bad
  - testing should "cover" all lines of code: every line of the script runs at least once during all your testing

Scripts and the checking program

- test your script incrementally as you build it up into its final form
- you need to be able to determine whether your script is behaving as you intended
- use -x and/or -v to "watch" it execute:
  - sh -x -u myscript.sh
  - -x: print each statement before executing

Refresher on quoting

- http://teaching.idallen.com/cst8207/14w/notes/320_shell_variables.html
- http://teaching.idallen.com/cst8207/14w/notes/440_quotes.html

- You want variables to be inside double quotes, for three main reasons:
  1. globbing characters inside the variable will not be used to match filenames when double qoutes are used
  2. if the variable is empty, without double quotes it vanishes completely, and that's normally not what we want
  3. Spaces inside the variable will not split the value into words if double quotes surround the variable

Variables with null values

- If a variable has a null value, as in

```
myvar=
# both of the following result in an error, because myvar is empty
if [ $myvar = something ] ; then echo yes; fi
# after variable expansion the above becomes (error)
if [ = something ] ; then echo yes; fi
```

- If we put the same variable in double quotes:

```
myvar=
# both of the following do not result in an error (or any output)
if [ "$myvar" = something ] ; then echo yes ; fi
# after variable expansion the above becomes same as
if [ "" = something ] ; then echo yes ; fi
```

printf for script output

- formatted printing (man printf)
- `printf format arguments [...]`
  - `format` is a string of characters containing
    - plain characters: copied straight to output
    - escape sequences:
      - \n : newline
      - \t : tab
      - \a : bell (try it!)
    - format specifications with three parts
      - `flag`
      - `width`
      - `precision`

printf format specifications

- flags:
  - -: left justify rather than right justify
  - +: always display the sign of a number
  - <space>: minus sign if negative, nothing if positive
  - 0: pad with 0's instead of spaces
- width:
  - if the output has fewer than width characters, pad it with spaces on the left (see flags above) so it occupies width characters
- precision:
  - an optional dot and digit string specifying the maximum number of characters (or number of digits after the decimal point for 'e' and 'f' -see below)

printf format specifications (cont'd)

- format: a character specifying the format of the output
  - d: decimal integer
  - f: floating point number
  - s: string (if precision is 0 or missing, all characters are printed, otherwise limited by precision)

printf examples

```
printf "hello\tthere"
```

- \t gets replaced with tab

```
printf "hello %s there" you
```

- prints "hello you there"

```
printf "hello %10s there" you
```

- prints "hello   you there"

- See the file "printf_examples.txt" for more examples

case statement

```
case test-string in
    pattern-1)
        command1
        command2
        ;;
    pattern-2)
        command3
        command4
        ;;
    *)
        command5
        ;;
esac
```

case statement continued

- the patterns are globbing patterns matched to the test-string
- So we tend to use the * pattern as a catchall, if all other matches fail, but that's not required
- case statement exit status is the exit status of the last command in the matching block, or 0 if no blocks match

case statement continued

- We can use the vertical bar to specify alternative patterns:

```
case "$character" in
    a|A)
        echo "The character is A or a"
        ;;
    [bB])
        echo "the character is B or b"
        ;;
    *)
        echo "The character is not A or a or B or b"
        ;;
esac
```

Some Special References

$BASH	the name used to invoke this instance of bash (/bin/sh if we use "#!/bin/sh" at top of script)
$$	the PID of this shell
$-	"**sh**" options currently set
$?	return code from the just-previous command
$!	the PID of the most recent background job

Signals and the TRAP statement

- Various signals can be trapped and your own script code executed instead of the system's normal code. Although there are up to 64 signals available, we will consider only a few of them:
- **SIGHUP** (signal 1 or **HUP**: hang up) is issued for a remote connection when the connection is lost or terminated; it's also used to tap a daemon on the shoulder, to re-read its config files.
- **SIGINT** (signal 2 or **INT**) is the keyboard interrupt signal given by **Control-C**.
- **SIGKILL** (signal 9 or **KILL**) cannot be ignored or trapped.
- **SIGTERM** (signal 15 or **TERM**) is the default signal used by kill(1) and killall(1).

Signal-like events and TRAP

- The **EXIT** event (also "signal" 0) occurs upon exit from the current shell.
- The **DEBUG** event takes place before every simple command, **for** command, **case** command, **select** command, and before the first command in a function. See also the description of **extdebug** for the **shopt** built-in for details of its effect.
- The **ERR** event takes place for each simple command with a non-zero exit status, subject to these conditions: it is not executed if the failed command is part of a **while**, **until**, or **if** condition expression, or in a **&&** or **||** list, or if the command's return value is being inverted via **!**. See also **errexit** for details.
- The **RETURN** event occurs each time a shell function or a script executed with the **.** (that's a dot) or **source** built-in returns to its caller.

Signals and the TRAP statement

- You can set a trap:

  trap 'statement; statement; …' event-list

- The trap statement list is read by the shell twice, first when it's set (it's set once only, before it is to be used, and stays active until you clear it).
- It's read a second time when it's executed.
- If you enclose the statement in single quotes, substitutions only take place at execution time.
- If you use double quotes, substitutions will take place upon both readings.
- If statement is omitted, the signals (use **-** (dash)) for all) are reset to the default.
- If statement is a null (empty) string, the signals specified will be ignored.

Signals and the TRAP statement

- To set a trap for **SIGINT**:

  trap 'statement; statement; …' INT

- To turn it off again:

  trap INT

- To prevent any **SIGINT** handling (ignore signals):

  trap " " INT

- Be cautious in trapping **SIGINT**: how will you stop a run-away script?
- To see what traps are set (you can see traps for specific events by listing the names or numbers):

  trap -p

- To list the names for signals **1** to **SIGRTMAX**:

  trap -l # that's an ell, not a one

Trap Sample Script

```
#!/bin/bash
count=0

# set trap to echo, then turn itself off
trap 'echo -e \\nSIGINT ignored in $count; \
     trap - sigint' sigint

# loop for a while
while (( count < 10 )); do
    (( count++ ))
    read -p "$count loop again? " response
done

# if loop ends, display count
echo loop count $count
exit 0
```

```
System Prompt$ ./traptest
1 loop again?
2 loop again?
3 loop again?
4 loop again? y
5 loop again? n
6 loop again?
7 loop again? q
8 loop again? help
9 loop again? ^C
SIGINT ignored in 9

10 loop again? q
11 loop again? y
12 loop again? n
13 loop again? ^C
System Prompt$
```

Functions in bash

- You will learn that functions are exceptionally useful, and it's good to see them in bash.
- A function is a group of regular shell-script statements is a self-contained package.
- You define a function as:

```
function somename () {
    statement
    statement
 ...
}
```

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

bash Functions

In general:

[**function**] name () compound-command [redirection]

Any parameters you pass to a function will be positional parameters inside that function

Functions

- Function scope is from the point the function is defined to the end of the file (that is, it must be defined before you can use it). Generally, that means that all functions precede the main body of the script.
- As a result, previously-written functions are often included in a script using the **source** (also **.** (dot)) statement near the top of a script.
- You can define **local** variables to be used only inside the function, while your normal variables from outside the function can always be used.
- If you wish, you can pass arguments into a function as positional parameters (**$1** and so on; this is by far the recommended approach).

Functions

- You may have noticed that traps behave like a special form of function. They are called (or invoked) by an event and consist of a collection of command statements. This is not an accident.
- To unset (delete or remove) a function:

  unset -f functionname
- To list defined function names (note: my system seems to have over 400 functions, of which I have only defined 4 of my own):

  declare -F | less
- To list functions and definitions:

  declare -f [functionname]

A Simple Sample

The **rot13** script is an implementation of the Caesar code message encryption. It simply rotates the message 13 characters through the alphabet, retaining case. No numbers or punctuation characters are affected.

```
rot13 ()
{
   echo "$*" | \
                tr '[a-mA-Mn-zN-Z]' '[n-zN-Za-mA-M]'
   return 0
}
```

As you can see, **rot13** accepts command-line arguments which it passes via **echo** through a translate (**tr**) command that will print the result on **stdout**.

Entering **rot13 sheesh** produces **furrfu** on **stdout**, while the reversed **rot13 fuurfu** displays **sheesh**.

When to write a function

There are a lot of scripting situations where writing a short function of your own is a good idea.

Some of these include:

- Some common activity that will be used frequently
- Part of a larger script that will be repeated at least 2 or 3 times, perhaps slightly differently each time
- An uncommon activity used only once in a while, but you don't want to have to remember the details
- A tricky bit of logic – write once, use over and over, even if its not often
- A part of a large script that will only be used once
- The "Lego block" approach to scripting – develop functions that can be "plugged together" to form a complete script with a little "glue"