COSC 3750

Shell Scripts

Kim Buckner

University of Wyoming

Jan. 26, 2023

Last time

- Discussed various utilities.
- Text utilities.
- Process control utilities.
- File utilities

Introduction

- Shell scripting can be simple or complex.
- Best to start with the simple and work up.
- Usually use either Bourne shell or the C shell.

Bourne shell scripts

- I think these are most effective and portable.
- Shell scripts are a text file that is (usually) executable.
- The lines are equivalent to typing at the command prompt.
- If you want the file to be **correctly** executed, it needs a special line.

(more . . .)

- Begin the file with a line like this

  `#!/bin/sh`

- Every modern shell understands this syntax and executes the listed program.
- Then passes the rest of the file to the program as input.
- This works with many things other than shells, like gawk, sed, and perl.

(more . . .)

- The *pound-bang* should always be
  - at the left margin and
  - the first line in the file.
- Always, always use the full path, never just "sh". Security issue.
- In general, comments begin with a # and continue to the end of the line.
- Good editors can do syntax highlighting.

Then . . .

- After that first line, just type in the commands, standard system utilities, programs you have written, or shell functions and shell syntax.
- **Make no assumptions**, shell programming does NOT rely on some programming language like C.
- Most of the syntax is old and slightly cumbersome, with quirks.

Variables

- Variables follow pretty standard naming rules.
- No data type, all are treated as strings
- Declaration and assignment are done at the same time in bash/sh.
- `var=value`
- Referenced with a dollar sign:

```
if [ $x == "bill" ]; then
```

Bourne shell

- For the Bourne shell, arguments to the script are readily available inside the script.
- The first argument is $1, the second $2, etc. thru $9
- The entire command line is $*.
- The number of arguments is $#.
- $0 is the name used to invoke the script.

Shift

- This is a command that "shifts" the arguments.
- The easy version is just `shift`.
- That moves $2 to $1, $3 to $2 and so on. The original $1 is LOST.
- Can do `shift n` where n is an integer and n arguments are shifted, not just 1.

Quotation marks

- There are often problems with strings with spaces or special characters.
- Using quotation marks around a string makes the shell treat it as a single word.
- Double quotation (quote) marks allow variables to be expanded within the quotes.
- Single quotation (apostrophe or tick) marks remove **all** special meaning from characters like the dollar sign.

Back quotes (backtick)

- Not interchangeable with the other types of quotation marks.
- Everything within the back quotes is treated as a command.
- It is executed in a subshell and the standard output of that is substituted within the current script as a string.

Syntax

- Simple but the rules are a little clunky.
- Everything must be delimited with whitespace.
- For instance `if[-z "$x"]; then` is a syntax error.
- Must be `if [ -z "$x" ]; then`.

Flow control constructs

- There are several different ones, the most common are `if`, `for` and `case`.
- There are others and you might need them but I usually don't.
- These are `while` and `until`.

if

- `if` *list*; `then` *list*; `fi`
- Can add any number of `elif` *list*; `then` *list*
- And can have a single `else` *list*;
- Here, *list* is a list of commands.

(more . . .)

- Most commonly, the *list* immediately after the `if` or `elif` is a "test".
- This is actually a command. By using the brackets (square), you get the shell's version.
- All commands have **some** exit status, an integer value. This is NOT printed to standard output or standard error.

Test

- In C and C++, this is the value returned by main().
- The basic syntax of the test is
  `[ var op var ]`
- There are a large number of operators, some numeric, some string, and some file system.
- For instance `[ -x "$f" ]` would test the string in $f and if it was the name of an executable file then the test would be true.

(more . . .)

- Of course that means that "test" returns **0**.
- The IXes are odd to many because when a program executes **correctly**, the program signifies this by returning 0.
- Anything else is an error (incorrect operation).

(more . . .)

- Another example is `[ $# -lt 5 ]`.
- This is actually an arithmetic test but the "$<$" operator is reserved for **string** operations.

(more . . .)

- You can test multiple things at once, as in C
- `[ $# -gt 2 -a "$1" == "-q" ]`
- Make sure that you test such things thoroughly before deleting files.

for I

- **Bash** supports versions that I am not sure are portable. Stick with the basics.
- `for` *name [in list]*; `do` *list*; `done`
- *name* is an identifier that is used as the loop control.
- When the name first occurs, do NOT use dollar sign.

for II

- The loop repeats, each time assigning the next one of *list* to *name* for use in the loop.
- If the optional '`in` *list*' is omitted, the script's **arguments** are used.

Example

```
for i
do
  echo $i
done
```

Oddities

- The *list* of strings is assumed to be whitespace delimited.
- This can be a problem; all of a sudden, too many strings.
- If the `do` is not on a separate line, must be preceded by a semicolon.

case

```
case word in
(pattern)
    list ;;
⋮
esac
```

(more . . .)

- Each pattern is compared to the *word*.
- The patterns can be multiple patterns using | as "or".
- Each '`pattern` *list*' set MUST end with ;; or ;& or ;;&
- ;; is equivalent to break. ;& is the fall through and ;;& is continue testing patterns.
- The last two may not be portable. (*bash* peculiar)