Shell Programming Topics

- Creating Shell Scripts
- Globbing
- Aliases, Variables/Arguments, and Expressions

Shell Programming Topics

- Shells, data, and debugging
- Structuring control flow
- Exit status

Shell Programming Topics

- Not (just) globbing: regular expressions
  - `grep, awk, perl` all also support regular expressions

Advantages of shell scripts

- Can very easily automate a group of tasks, especially those with i/o that are related
- Can very easily leverage powerful Unix tools

Disadvantages of shell scripts

- Shell scripts execute slowly.
- Advanced programming techniques aren't a feature of shell programming. Abstraction and encapsulation are poorly supported.

What shell to use

- For programming, most people have preferred `sh` and its derivatives such as `bash`.
- We will use `bash` for programming, although we will also talk about `csh` when appropriate in command shells.

What shell to use

In the past, many people have preferred `csh` and `tcsh` as command line shells; however, it appears that `bash` is now preferred since its support for command line editing is quite strong and it also is quite useful for shell programming.

What shell to use

There is also program `busybox` which is also worth knowing about. It is a shell — and a lot more. The binary itself includes many other programs such as `head, tail, ps, top, find, crontab,` and tar as built-ins.

Finding more information

- `man bash`
- `man {alias, bg, bind, break, builtin, cd, command, compgen, ...}`
- `info bash`
- Google `bash`

Creating a script

By convention, we use an extension of `.sh` for shell scripts.
The first line needs to be

```
#!/bin/bash
#!/bin/sh
#!/bin/csh
#!/sbin/bash
```

Creating a script

Now you should put some comments:

```
# 2008 09 06 - original version by rdl
# 2008 09 07 - updated ``text'' by rdl
#
# this shell program is used to confabulate
# the obfuscated
#
```

Using `echo`

- The program (and builtin) `echo` is useful for sending a given string or strings to stdout.

```
$ echo a b c
a b c
$ echo "a b c"
a b c
$ echo "$SHELL a b c"
/bin/bash a b c
```

Using `echo`

- The program (and builtin) `echo` is useful for sending a given string or strings to stdout.

```
$ echo "$SHELL a b c"
/bin/bash a b c
$ echo $SHELL a b c
/bin/bash a b c
$ echo '$SHELL a b c'
$SHELL a b c
```

Shell variables

- Do not have to be declared: just use them. (If you want to, you can declare them with `declare`; generally only useful to make variables read-only.)
- Can be assigned a value, or can just have a blank value
- Can dereferenced with a "$"

Shell variables

Examples:

```
$ a=b
$ b=$a
$ echo "a = $a , b = $b"
a = b , b = b
```

reading values from the command line

From the man page for bash:

> ``One line is read from the standard input, . . . and the first word is assigned to the first name, the second word to the second name, and so on, with leftover words and their intervening separators assigned to the last name. If there are fewer words read from the input stream than names, the remaining names are assigned empty values. The characters in IFS are used to split the line into words.''

`read` example

```
$ read a b c d e f
apple beta cherry delta eta figs and more
$ echo "$a - $b - \
 $c - $d - $e - $f"
apple - beta - cherry - delta - eta - figs and more
```

`read` example

It is also good to note that you can also specify that items are to go into an array rather than just individually named variables with the `-a ARRAYNAME` option.

For example:

```
$ read -a arr
a b c d e f g h
$ for i in 0 1 2 3 4 5 6 7
> do
> echo ${arr[$i]}   # note the odd syntax
> done
a
b
c
d
e
f
g
```

Command line parameters

- When you call a shell script, command line parameters are automatically setup with $1, $2, etc...

```
$ ./Script1.sh abc def ghi
first 3 args: 'abc' 'def' 'ghi'
```

- $0 refers to the name of the command (the first item)

More on command line arguments

- $# refers to the number of command line arguments.
- $@ refers to the all of the command lines arguments in one string.

Example:

```
$ ./Script2.sh abc def ghi jkl
There are 4 arguments: abc def ghi jkl
```

Debugging tips

- The options `-x` and `-v` are very helpful. You can either add them to the initial `#!` line, or you can call the shell at the command line:
- bash -xv Script1.sh abc def

Debugging tips example

```
$ bash -xv Script1.sh ls asd asdf asdf
#!/bin/bash

# 2006 09 06 - Small test script

echo "first 3 args: '$1' '$2' '$3'"
+ echo 'first 3 args: '\"ls'\" '\"asd'\" '\"asdf'\"
first 3 args: 'ls' 'asd' 'asdf'
echo "cmd: '$0'"
+ echo 'cmd: '\"Script1.sh'\"'
cmd: 'Script1.sh'
$ bash -x Script1.sh ls asd asdf asdf
+ echo 'first 3 args: '\"ls'\" '\"asd'\" '\"asdf'\"
first 3 args: 'ls' 'asd' 'asdf'
+ echo 'cmd: '\"Script1.sh'\"'
cmd: 'Script1.sh'
```

Testing

- You can test with square brackets:

```
$ [ $ -e /etc/hosts $ ] $
```

- You can also test with `test`:

```
test -e /etc/hosts
```

Testing

Example:

```
$ if test -e /etc/hosts
> then
> echo exists
> fi
exists
$ if [ -e /etc/hosts ]
> then
> echo exists
> fi
exists
```

File testing conditions

You can readily check various file status items:

```
[ -d DIR ]              #  True if
                        #  directory DIR exists.
[ -e SOMETHING ]        #  True if
                        #  file or directory
                        #  SOMETHING exists.
```

File testing conditions

```
[ -f FILE ]          #  True if regular
                     #  file FILE exists.
[ -r SOMETHING ]     #  True if file or
                     #  directory SOMETHING
                     #  exists and is readable.
[ -s SOMETHING ]     #  True if file or
                     #  directory SOMETHING
                     #  exists and
                     #  has a size greater than zero.
[ -x SOMETHING ]     #  True if file or directory
                     #  SOMETHING exists and
                     #  is ``executable" by this user
```

Numeric tests

You can readily check various numeric values:

```
[ 0 -eq 1 ]    # equality
[ 1 -ne 1 ]    # inequality
[ 1 -lt 1 ]    # less than
[ 1 -gt 1 ]    # greater than
[ 1 -le 1 ]    # less than or equal
[ 1 -ge 0 ]    # great than or equal
```

String tests

You can readily check various numeric values:

```
[ -z STRING ]          # is the string STRING zero leng
[ -n STRING ]          # is the string STRING non-zero
[ STR1 == STR2 ]       # ``bash'' equality;
                       # POSIX prefers ``=''
[ STR1 != STR2 ]       # inequality
[ STR1 < STR2 ]        # less than
[ STR1 > STR2 ]        # greater than
```

Note that it is a very good idea to "" quote any string variables; otherwise, the corresponding blank in `if [ $var1 != "today" ]` becomes `if [ != "today" ]`!

exit

- You can explicitly exit a shell with `exit`, which can take an argument which will give the exit status of the process. (If you don't specify the optional value, the exit status for the whole shell will take the value of the last command to execute.)

```
$ bash
$ exit 3
exit
$ echo $?
3
```

if / then

- We can write `if / then` statements like:

```
if condition
then
    [ ... statements ... ]
fi
```

Quoting

- Single quotes stop any globbing or variable expansion within them, and create a single token (i.e., whitespace within the quotes is not treated as a separator character.)
- Double quotes allow globbing and variable expansion within them, and create a single token (i.e., whitespace within the quotes is not treated as a separator character.)
- You can use the backslash to quote any single character.

Quoting examples

```
animal=”horse”
echo $animal      #prints: horse
echo ’$animal’    #prints: $animal
echo ``$animal” #prints: horse
cost=2000
echo ’cost: $cost’      #prints: cost: $cost
echo ``chost: $cost”  #prints: cost: 2000
echo ``cost: \$cost”  #prints: cost: $cost
echo ``cost: \$$cost” #prints: cost: $2000
```

Multiple conditions

```
[ $1 -eq $2 ] && [ -e /etc/hosts ]
[ $1 -eq $2 ] || [ -d /etc ]
```

General if/then/else

```
if condition
then
   [ ... statements ... ]
elif condition
then
   [ ... statements ... ]
[ ... more elifs ... ]
else
   [ ... statements ... ]
fi
```