CS 241: Systems Programming
Lecture 7. Shell Scripting 2

Fall 2019

Prof. Stephen Checkoway

Script positional parameters

```
$ ./script arg1 ... argn # or bash script arg1 ... argn
```

Special variables

- `$#` — Number of arguments
- `$0` — Name used to call the shell script (./script or script)
- `$1`, `$2`, …, `$9` — First nine arguments
- `${n}` — *n*th argument (braces needed for $n > 9$)
- `"$@"` — all arguments; expands to each argument quoted
- `"$*"` — all arguments; expands to a single quoted string

Two special builtin commands

`set --`

- Can set positional parameters (and `$#`)

  ```
  set -- arg1 arg2 … argn
  ```

`shift`

`shift n`

- Discard first *n* parameters and rename the remaining starting at `$1`
- If `n` is omitted, it's the same as `shift 1`
- Updates `$#`

Iterate over parameters

```
while [[ $# -gt 0 ]]; do
  arg="$1"
  # whatever you want to do with ${arg}
  shift
done
```

Functions

```
#!/bin/bash

num_args() {
  echo "foo called with $# arguments"
  if [[ $# -gt 0 ]]; then
    echo "  foo's first argument: $1"
  fi
}

echo "Script $0 invoked with $# arguments"
if [[ $# -gt 0 ]]; then
  echo "  $0's first argument: $1"
fi

num_args 'extra' "$@" 'args'
```

`local` creates a local variable.

What does this script print out?

A. `A`

B. `B`

C. `C`

D. The empty string

E. Nothing, it's a syntax error

```
#!/bin/bash

foo() {
  x="$1"
}
bar() {
  local x="$1"
}

x=A
foo B
bar C
echo "${x}"
```

`local` creates a local variable.

What does this script print out?

A. `A`

B. `B`

C. `C`

D. `D`

E. Nothing, it's a syntax error

```
#!/bin/bash

foo() {
  x="$1"
}
bar() {
  local x="$1"
  foo "$2"
}

x=A
foo B
bar C D
echo "${x}"
```

Lists — sequence of commands

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | ... | cmdn`

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | ... | cmdn`

- Exit value is exit value of last command in the pipeline

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | ... | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | ... | cmdn`

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | ... | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | ... | cmdn`

Lists

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | … | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | … | cmdn`

Lists

- `pipeline1 ; pipeline2 ; … ; pipelinen`
  can replace `;` with newline

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | … | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | … | cmdn`

Lists

- `pipeline1 ; pipeline2 ; … ; pipelinen`
  can replace `;` with newline
- `pipeline1 && pipeline2`
  `pipeline2` runs if and only if `pipeline1` returns 0

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | … | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | … | cmdn`

Lists

- `pipeline1 ; pipeline2 ; … ; pipelinen`
  can replace `;` with newline
- `pipeline1 && pipeline2`
  `pipeline2` runs if and only if `pipeline1` returns 0
- `pipeline1 || pipeline2`
  `pipeline2` runs if and only if `pipeline1` doesn't return 0

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | … | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | … | cmdn`

Lists

- `pipeline1 ; pipeline2 ; … ; pipelinen`
  can replace `;` with newline
- `pipeline1 && pipeline2`
  `pipeline2` runs if and only if `pipeline1` returns 0
- `pipeline1 || pipeline2`
  `pipeline2` runs if and only if `pipeline1` doesn't return 0
- `pipeline &`
  runs `pipeline` in the background

When writing a script, we often want to change directories with cd. If the directory doesn't exist, the script should exit with an error.

Which construct should we use?

A. `cd "${dir}" && exit 0`

B. `cd "${dir}" || exit 0`

C. `cd "${dir}" && exit 1`

D. `cd "${dir}" || exit 1`

E. `cd "${dir}" && exit 2`

Arrays

Assign values at numeric indices

- `arr[0]=foo`
- `arr[1]=bar`

Assign multiple values at once

- `arr=(foo bar)`
- `txt_files=(*.txt) # pathname expansion/globbing`

Append (multiple values) to an array

- `arr+=(qux asdf)`

Arrays

Access an element

- `${arr[0]}`
- `${arr[1]}`
- `n=42`
  `${arr[n]}`

Access all elements

- `"${arr[@]}"` # expands to each element quoted by itself
- `"${arr[*]}"` # expands to one quoted word containing all elements

Array length

- `${#arr[@]}`

If `arr` is the two element array

```
arr=('foo bar' baz)
```

how should we print each element of `arr`?

A.
```
for elem in ${arr}; do
  echo "${elem}"
done
```

B.
```
for elem in "${arr}"; do
  echo "${elem}"
done
```

C.
```
for elem in "${arr[*]}"; do
  echo "${elem}"
done
```

D.
```
for elem in "${arr[@]}"; do
  echo "${elem}"
done
```

E.
```
for (( n=0 ; n < ${#arr[@]}; n+=1 )); do
  echo "$arr[n]"
done
```

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-07.html

Grab a laptop and a partner and try to get as much of that done as you can!