EECS 2031 E 3.0

Software Tools

Week 10: November 13, 2018

A real example

- Am I logged in more than once?

```
sh-3.2$ who | grep jenkin | wc -l
       2
```

```
sh-3.2$ cat foo.sh
#!/bin/bash
z=$(who | grep jenkin | wc -l)
echo $z
sh-3.2$ ./foo.sh
2
```

```
#!/bin/bash
z=$(who | grep jenkin | wc -l)
if test "$z" -gt 1
then
  echo "You are logged in more than once"
elif test "$z" -eq 1
then
  echo "You are logged in only one time"
else
  echo "You are not logged in at all?????"
fi
exit 0
sh-3.2$ ./foo.sh
You are logged in more than once
sh-3.2$
```

A real example

- Lets not hard code ‘jenkin’, lets figure it out

```
sh-3.2$ who am i
jenkin       ttys000  Nov  6 18:46
sh-3.2$
```

Use ‘sed’ stream editor to delete everything from the first blank to the end

```
sh-3.2$ who am i | sed 's/ .*$//'
jenkin
sh-3.2$
```

```
#!/bin/bash
user=$(who am i|sed 's/ .*$//')
z=$(who | grep "$user" | wc -l)
if test "$z" -gt 1
then
  echo "$user is logged in more than once"
elif test "$z" -eq 1
then
  echo "$user is logged in only one time"
else
  echo "$user is not logged in at all?????"
fi
exit 0
sh-3.2$ ./foo.sh
jenkin is logged in more than once
sh-3.2$
```

Let the user specify the user to check

```
#!/bin/bash
if test $# -eq 1 ; then
  user=$1
elif test $# -eq 0 ; then
  user=$(who am i|sed 's/ .*$//')
else
  echo "Usage $0: [user]"
  exit 1
fi
z=$(who | grep "$user" | wc -l)
if test "$z" -gt 1
then
  echo "$user is logged in more than once"
elif test "$z" -eq 1
then
  echo "$user is logged in only one time"
else
  echo "$user is not logged in at all"
fi
exit 0
```

```
sh-3.2$ ./foo.sh
jenkin is logged in more than once
sh-3.2$ ./foo.sh mary
mary is not logged in at all
sh-3.2$ who
_mbsetupuser console  Oct 31 23:01
jenkin       console  Oct 31 23:01
jenkin       ttys000  Nov  6 18:46
sh-3.2$ ./foo.sh _mbsetupuser
_mbsetupuser is logged in only one time
sh-3.2$ ./foo.sh a b c
Usage ./foo.sh: [user]
sh-3.2$
```

Iteration: for

- Bash as while, for and until structures
- while test; do …. done

```
#!/bin/bash
z=1
while test $z -le 10; do
  echo $z
  let "z=$z+1"
done
```

```
sh-3.2$ ./foo.sh
1
2
3
4
5
6
7
8
9
10
```

Iteration: until

- until test ; do … done

```
#!/bin/bash
z=1
until test $z -eq 11; do
  echo $z
  let "z=$z+1"
done
```

```
sh-3.2$ ./foo.sh
1
2
3
4
5
6
7
8
9
10
```

Iteration: for

- Not similar to similar structures in C
  - Similar structures in modern Java and (all) Python's
- for var in list; do … done

```
#!/bin/bash
z=1
for z in 1 2 3 4 5 6 7 8 9 10; do
  echo $z
done
```

```
./foo.sh
1
2
3
4
5
6
7
8
9
10
```

Iteration: for

- Iterate over all files in a directory?

```
#!/bin/bash
for z in `ls`; do
  echo $z
done
```

```
./foo.sh
foo.sh
hello
hello.c
output
```

Real world example

- You have a collection of folders (say 1 per assignment)
- You want to print them out in some pretty way
  - Pretty is different for different file types

```
#!/bin/bash
for z in `echo *`; do
  echo $z
  if test –d $z; then
    echo $z is a directory
  fi
done
```

Process every file, and check to see if its a directory

```
#!/bin/bash
for z in `echo *`; do
  echo $z
  if test -d $z; then
    echo $z is a directory
    cd $z
    echo *
    cd ..
  fi
done
```

And then for those directories, list their contents (but not their contents, contents)

```
#!/bin/bash
for z in `echo *`; do
  if test -d $z; then
    echo Processing assignment directory $z
    cd $z
    for q in `echo *`; do
      echo "File $q"
    done
    cd ..
  fi
done
```

```
sh-3.2$ ./foo.sh
Processing assignment directory a1
File hello.c
Processing assignment directory a2
File test2.c
Processing assignment directory a3
File test3.c
File test3.h
```

Now we have the files in those directories

Want to treat different file types differently

- Easiest way to do this is by file extension
  - .c, .h, .txt
  - Lets ignore all other file types
  - Use case statement to do this (could use if, but that's boring)

Case statement

- case expression in case1) cmd;; case2) cmd;; esac
- case1, case2 etc are patterns
  - * matches everything
  - *.c matches c files, *.h matches h files etc.

```
#!/bin/bash
for z in `echo *`; do
  if test -d $z; then
    echo Processing assignment directory $z
    cd $z
    for q in `echo *`; do
      echo "File $q"
      case $q in
      *.c)
        echo "Its a c file"
        ;;
      *.h)
        echo "Its an h file"
        ;;
      *.txt)
        echo "its a txt file"
        ;;
      *)
        echo "NO idea"
      esac
    done
    cd ..
  fi
done
```

```
./foo.sh
Processing assignment directory a1
File hello.c
Its a c file
File x
NO idea
Processing assignment directory a2
File test2.c
Its a c file
Processing assignment directory a3
File test3.c
Its a c file
File test3.h
Its an h file
```

Now what to do with the files

- Lets make them eps files
- Number them
- Put them somewhere

```
#!/bin/bash
TMPDIR=/tmp
OUTDIR=/tmp/outdir$$
rm -f $OUTDIR
mkdir $OUTDIR
fileNo=0
for z in `echo *`; do
  if test -d $z; then
    echo Processing assignment directory $z
    cd $z
    for q in `echo *`; do
      echo "File $q"
      case $q in
      *.[ch])
        echo "Its a c or h file"
        let "fileNo=$fileNo+1"
        rm -f $TMPDIR/junk$$.txt
        pr -h "$a/$q" $q >$TMPDIR/junk$$.txt
        echo $TMPDIR/junk$$.txt
        cupsfilter $TMPDIR/junk$$.txt >$OUTDIR/file$fileNo.pdf 2>/dev/null
        rm -f $TMPDIR/junk$$.txt
        ;;
      *)
        echo "NO idea"
      esac
    done
    cd ..
  fi
done
echo "There are $fileNo" files to print in "$OUTDIR"
```

Functions in Bash

- As Bash programs become larger it becomes prudent to break them down into smaller modules
- Two approaches in Bash
  - Have one script defined in terms of other scripts/ programs.
  - Have internal functions.

Scripts within scripts

- Given that Bash will execute commands defined outside of the script, you can clearly have one script ‘call’ another.

```
wanderereecsyorkuca:t jenkin$ cat a.sh
#!/bin/bash
echo "in a"
./b.sh
echo "back in a"
wanderereecsyorkuca:t jenkin$ cat b.sh
#!/bin/bash
echo "in b"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
in a
in b
back in a
```

Downside with this #1

- Spawns a new process
  - Some overhead in this

```
#!/bin/bash
echo "in a $$"
./b.sh
echo "back in a"
wanderereecsyorkuca:t jenkin$ cat b.sh
#!/bin/bash
echo "in b $$"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
in a 6953
in b 6954
back in a
```

Downside with this #2

- Shell variables belong to the process

```
wanderereecsyorkuca:t jenkin$ cat a.sh
#!/bin/bash
echo "in a $$"
a1=123
echo "In a: before a1 $a1 b1 $b1"
./b.sh
echo "In a: after a1 $a1 b1 $b1"
wanderereecsyorkuca:t jenkin$ cat b.sh
#!/bin/bash
echo "in b $$"
b1=88
echo "In b before: a1 $a1 b1 $b1"
a1=77
echo "In b after: a1 $a1 b1 $b1"
wanderereecsyorkuca:t jenkin$
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
in a 6969
In a: before a1 123 b1
in b 6970
In b before: a1  b1 88
In b after: a1 77 b1 88
In a: after a1 123 b1
```

We will come back to this nuance later.

Summary

- Separate processes
  - With all that entails
- Can ‘get around’ the variable problem (save to file, etc.) but it gets hacky/ugly quickly

Functions

- Syntax
  - function name() { … }
  - The parenthesis are optional

```
wanderereecsyorkuca:t jenkin$ cat c.sh
#!/bin/bash
function c() {
  echo "now in function c"
}

echo "about to call c"
c
echo "back"
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
about to call c
now in function c
back
```

Functions: parameters

- Parameters are $1…$n
- Number is $#
- All parameters is given by $@

```
#!/bin/bash
function c {
  echo "now in function c"
  if test $# -eq 0; then
    echo "no parameters"
  else
    for z in $@; do
      echo arg $z
    done
  fi
}

echo "about to call c"
c all this and heaven too
echo "back"
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
about to call c
now in function c
arg all
arg this
arg and
arg heaven
arg too
back
```

Variables in function

- By default, global
- Keyword local to identify as local
- Beware of variable hiding

```
wanderereecsyorkuca:t jenkin$ ./c.sh
main v1 foo
main v2 bar
about to call c
now in function c
c v1 hello world
c v2 goodbye world
back
main v1 foo
main v2 goodbye world
```

```
#!/bin/bash
function c {
  echo "now in function c"
  local v1
  v1="hello world"
  v2="goodbye world"
  echo "c v1 $v1"
  echo "c v2 $v2"
}

v1="foo"
v2="bar"
echo "main v1 $v1"
echo "main v2 $v2"
echo "about to call c"
c
echo "back"
echo "main v1 $v1"
echo "main v2 $v2"
```

Return values

- Functions can have a return value
  - Exit value (#?)

```
wanderereecsyorkuca:t jenkin$ ./c.sh
going
now in function c
0
now in function c
1
```

```
#!/bin/bash
function c {
  echo "now in function c"
  if test $# -eq 0; then
    return 0
  fi
  return 1
}

echo "going"
c
echo $?
c hello
echo $?
```

Overriding commands

- If you call functions the same name as commands, then you will change the default version of that command.

```
#!/bin/bash
function ls {
  echo "what, you wanted me to do an ls?"
}

function echo {
  /bin/echo "Echo this $@"
}

date
ls
echo "hello world"
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
Thu 16 Nov 2017 11:47:34 EST
Echo this what, you wanted me to do an ls?
Echo this hello world
```

Arrays

- Infrequently used, but they exist in bash
- Can declare them explicitly
  - declare -a foo=(a b c)
- Or implicitly
  - foo=(a b c)
- ${foo[2]} - element 2
- ${foo[@]} - all of foo
- ${#foo[@]} - size of foo
- Arrays are 0 offset

```
#!/bin/bash
foo=(monday tuesday wednesday thursday friday saturday sunday)
echo "${foo[0]}"
echo "${foo[1]}"
echo "${foo[2]}"
echo "${foo[3]}"
echo "${foo[@]}"
echo "${#foo[@]}"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
monday
tuesday
wednesday
thursday
monday tuesday wednesday thursday friday saturday sunday
7
```

Arrays

- Array elements can be null (so they will not print)
  - Unset foo[2]
  - foo[0]=

```
#!/bin/bash
foo=(monday tuesday wednesday thursday friday saturday sunday)
echo "${foo[@]}"
unset foo[1]
echo "${foo[@]}"
foo[0]=
echo "${foo[@]}"
```

Arrays

- Can create arrays from arrays
- Can select parts of arrays
  - ${foo[@]:2:1}

```
#!/bin/bash
foo=(monday tuesday wednesday thursday friday saturday sunday)
echo "${foo[@]}"
foo=(${foo[@]} holiday extraday)
echo "${foo[@]}"
foo=(${foo[@]:3:2})
echo "${foo[@]}"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
monday tuesday wednesday thursday friday saturday sunday
monday tuesday wednesday thursday friday saturday sunday holiday extraday
thursday friday
```

Bash and subshells

- It is easy (too easy) to generate subshells in bash
  - Invoke another command
  - Use a programming construct which bash implements by using subshells
- In either event you need to be aware that the (sub) shell will have its own local variables that will vanish when the sub-shell is exited

Bash and sub-shells

- If you put a command in () it is executed in a subshell.

```
#!/bin/bash
x=7
echo before $x
(echo "in parenthesis $x";x=8;echo "in parenthesis $x")
echo "after $x"
```

```
wanderereecsyorkuca:t jenkin$ ./d.sh
before 7
in parenthesis 7
in parenthesis 8
after 7
```

Bash and sub-shells

- You can cause Bash to spawn subshells whenever you pipe the output of a command

```
#!/bin/bash
x=7
echo before $x
for i in 1 2 3 4 5 6 7 8; do
  let "x=$x+1"
  echo "inside $x"
done
echo "after $x"
```

```
./d.sh
before 7
inside 8
inside 9
inside 10
inside 11
inside 12
inside 13
inside 14
inside 15
after 15
```

```
wanderereecsyorkuca:t jenkin$ cat d.sh
#!/bin/bash
x=7
echo before $x
for i in 1 2 3 4 5 6 7 8; do
  let "x=$x+1"
  echo "inside $x"
done | cat >/dev/null
echo "after $x"
```

```
./d.sh
before 7
after 7
```

Bash and subshells

```
#!/bin/bash
x=7
echo before $x
for i in 1 2 3 4 5 6 7 8; do
  let "x=$x+1"
  echo "inside $x"
done >/dev/null
echo "after $x"
```

```
wanderereecsyorkuca:t jenkin$ ./d.sh
before 7
after 15
```

Summary

- Bash - a CLI (shell) based on sh before it
  - There are other shells. Bash is free so commonly used.
- Supports standard programming language constructs, untyped variables (int, string) and arrays
- Supports functions and the ability to invoke other programs (including other bash programs)
- Utilizes value of exit (very unix-friendly) to pass a single small value integer between processes
- Variables are ‘complex’ in that different programming features can lead to the spawning/use of subshells or separate processes with their own namespace.