

Arithmetic Operations

- Bash has built in support for arithmetic integer operations, similar to C operators
 - Either **let** or **\$(...)**
- **let a=16+5**
- **let "n=\$n-1"**
- **echo \$((a*b))** # no quote for * is necessary
- **echo \$((a--))**
- **echo \$((a**3))** # exponentiation

Arithmetic -- FP

- **N=`echo "scale=3; 13 / 2" |bc`**
- **echo \$N**

- **n=`bc << EOF**
 - › **scale=3**
 - › **13/2**
 - › **EOF`**
- **echo \$n**

Set

- **set** command re-sets positional parameters (arguments)
- **set apple banana cherry**
- **echo \$1, \$2, \$3**
- **set `date`**
- **echo \$1, \$2**
- **a='hello world!'**
- **set \$a** vs. **set "\$a"**

Testing

- To test various conditions, we use test command:
- for string = , !=, -z string (length is zero) -n string (length is non zero,), and string (not null).

```
#!/bin/sh
test $1 != Tom
echo $?

#!/bin/sh
[ $1 != Tom ]
echo $?
```

Testing

- [**-d file**] is identical to **test -d file**
- int1 -eq int2
- int1 -ne int2
- int1 -gt int2
- int1 -lt int2
- int1 -le int2

Testing

- [**-d file**] - if it is a directory ?
- [**-f file**] - a regular file ?
- [**-r file**] - the file readable ?
- [**-w file**] - the file writable ?
- [**-x file**] - the file executable ?
- [**-s file**] - the file has non-zero size ?
- [**-L file**] - a symbolic link ?
- [**-u file**] - the file has **suid** bit ?

Logical Operators

- **-a** logical AND
- **-o** logical OR
- **!** logical NOT
- **[-w res.txt -a -w score.txt]**
- **[-x op1 -o -x op2]**
- **[! -d Tmp]**
- The Bash extended test operator **[[...]]** allows
- usage of **&&,||,>,<** in an expression.
- **[[\$a>\$b]]**

Test Subtleties

- The following is bad practice:
- **[\$var = rightvalue] && echo OK**
- **[\$OSTYPE = "linux"] && echo**
- **Running in Linux**
- Why?

Test Subtleties

- What if **"\$var"** is blank? After substitution we get:
- **[= rightvalue] && echo OK**
- which is a "test" syntax error!

Test Subtleties

- An old sh programmer's trick:
- ["X\$var" = "Xrightvalue"] && echo OK
- ["X\$OSTYPE" = "Xlinux"] && echo
- **Running in Linux**
- Protects against unusual variable values

Test Subtleties

- [-d \$dir] || mkdir \$dir
- creates the directory \$dir if it does not already exist

Testing

- **a=010**
- **b=10**
- [\$a = \$b]
- # FALSE as two different strings
- [\$a -eq \$b]
- # TRUE as two numbers

Example

```
read marks
if [ $marks -ge 80 ]; then
    grade=A
elif [ $marks -ge 70 ]; then
    grade=B
elif [ $marks -ge 60 ]; then
    grade=C
else
    grade=D
fi
echo $grade
```

grade = c ?????? testing? Note
that there is no compilation

Example

```
myprogram < data.in > result.out
if [ -s result.out ]; then
    echo "Output generated !!"
else
    echo "empty output!!"
fi
```

Example

```
for SCRIPT in /path/to/scripts/dir/*
do
    if [ -f $SCRIPT -a -x $SCRIPT ]
    then
        $SCRIPT
    fi
done
```

Testing

```
• #!/usr/bin/bash
• # cookbook filename: strvsnum
• #
• # the old string vs. numeric
  comparison dilemma
• #
• VAR1=" 05 "
• VAR2="5"
• printf "%s" "do they -eq as equal? "
```

```
• if [ "$VAR1" -eq "$VAR2" ]
• then
• echo YES
• else
• echo NO
• fi
• printf "%s" "do they = as equal? "
```

```
• if [ "$VAR1" = "$VAR2" ]
• then
• echo YES
• else
• echo NO
• fi
```

```
$ bash strvsnum
do they -eq as equal? YES
do they = as equal? NO
$
```

Testing– Dealing with failures

- You want to do 2 commands, but the second one depends on the success of the first `cd testdir` then `rm *.dat`
- One solution is to use the double ampersand operator (run the second only if the first succeed)
- `$cd testdir 2>/dev/null && rm *.dat`
OR
- `cd testdir`
- `if(($?)); then rm *.dat; fi`

Arithmetic
expansion

Testing– Dealing with failures

- Another way is to set the `-e` option exit the first time encounter error (i.e. a non zero exit status) from any command in the script except while loops and if statement
- `set -e`
- `cd testdir`
- `rm *.dat`

Conditions

- We want to do more than just execute other programs
- *command1 && command2*
- executes *command1* - if *command1* has an exit status of 0 (true), then *command2* is executed
- Like C, this is short-circuiting

Conditions

- We have an 'or' operator as well:
- *command1 || command2*
- executes *command1* - if *command1* has an exit status of non-zero (false), then *command2* is executed
- Note that there is no space between the characters in '||' and '&&'

Testing— Dealing with failures

- Another way of doing it
- `cmd || printf "%b" "cmd failed\n"` How does that work?
- `cmd || { printf "%b" "cmd failed\n"; exit;}`

Looping

- for *variable* in *list_of_items*
- do
- *command1*
- *command2*
- ...
- *last_command*
- done

Looping

- for filename in *
- do
- echo \$filename
- done
- for filename in *.doc
- do
- echo "Copying \$filename to \$filename.bak"
- cp \$filename \$filename.bak
- done

Looping

- for i in 1 2 3 4 5 6 7 8 9 10
- do
- echo -n "...\$i"
- done
- echo # Clean up for next shell prompt

Looping

```
# Counts by looping for a fixed number of times
# Note do on same line requires semicolon.
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo -n "...$i"
done
echo # Output newline
```

```
Counts by looping for a fixed number of times
# Note do on same line requires semicolon.
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo -n "...$i"
done
sleep 5
echo # Output newline
```

Looping

```
# Counts backwards
for i in 10 9 8 7 6 5 4 3 2 1
do
    echo -n "...$i"
done
echo # Output new line
echo "Blast off!"
$ sh counter2
...10...9...8...7...6...5...4...3...2...1
Blast off!
```

Looping

```
for x in hello there world
do
    echo next word is $x
done
```

- Output:
- next word is hello
- next word is there
- next word is world

Looping

- Use outputs of another program as the word list using back-quotes
- for user in `cut -d: -f1 < /etc/passwd`
- do
- echo -n "\$user:"
- finger \$user
- done

Looping

```
# C-language-like for loop.  
# Must be run with bash.  
max=10  
for ((i=1; i <= max ; i++))  
do  
    echo -n "$i..."  
done  
echo
```

Redirection

- What is that?
- ls /fred > /dev/null 2> /dev/null

While

- Example: reads every line from stdin
- `while read l`
- `do`
- `echo line: $l`
- `done`

While

- Reads from a file instead
- `while read line`
- `do`
- `echo line: $line`
- `done <file`
- Or run in subshell
- `(while read line; do echo line:$line; done) <file`

Until

- Same as “while” but test is inverted
- `until test-cmd`
- `do`
- `command`
- ...
- `done`
- Execute ‘command’ while ‘test-cmd’ is false

Command line argument

- Special form of 'for' that implicitly loops over all
- command line arguments to script
- for x
- do
- ... x loops over all arguments ...
- done

Break and Continue

- Usual loop "escape" mechanisms:
- **break** [*n*]
- **continue** [*n*]
- The optional integer n specifies how many levels
- should be "broken" or "continued".

case ... esac

- Like a "switch" statement
- **case** *string* **in**
- *expr1*)
- *command* ;;
- *expr2*)
- *command* ;;
- ...
- **esac**

case ... esac

- Note that each case must end with ';;'
- Unlike C: no 'fall-through'
- Take first matching case then skip to bottom
- Expressions are special - wildcards plus '|', where
- '|' - means "or"
- "hello|goodbye" matches
- "hello" or "goodbye"

Example

- Mycal program In class discussion

```
#!/cs/local/bin/sh

case $# in
0) set `date`; m=$2; y=$6;;
1) m=$1; set `date`; y=$6;;
2) m=$1; y=$2;;
esac

case $m in
jan*)Jan*) m=1;;
feb*)Feb*) m=2;;
mar*)Mar*) m=3;;
apr*)Apr*) m=4;;
may*)May*) m=5;;
jun*)Jun*) m=6;;
jul*)Jul*) m=7;;
aug*)Aug*) m=8;;
sep*)Sep*) m=9;;
oct*)Oct*) m=10;;
nov*)Nov*) m=11;;
dec*)Dec*) m=12;;
[1-9][101112] ;;
*) y=$m;m="";;
esac

/usr/bin/cal $m $y
```

```
sh-4.1$ sh_case.sh
March 2016
Su Mo Tu We Th Fr Sa
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

sh-4.1$ sh_case.sh march
March 2016
Su Mo Tu We Th Fr Sa
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

sh-4.1$ sh_case.sh march 2016
March 2016
Su Mo Tu We Th Fr Sa
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Shift

- Built-in command:
- **shift** "moves" all arguments down 1
- **\$2** is moved into **\$1**, **\$3** is moved into **\$2**, etc.
- **\$1** is thrown away

Shift

- `#!/bin/sh`
- `while [ $# -gt 0 ]; do`
- `echo "processing arg: $1"`
- `shift`
- `done`

Shell Functions

- We can also write functions in the shell
- `name() {`
- `commands ...`
- `}`
- Parentheses `()` are always empty
- Note: no return type, no arguments declared
- Call via:
- `name [arg] [arg] ...`

Shell Functions

- Shell functions can take any number of arguments
- and return an exit status
- Arguments use the command-line argument syntax
- (\$1, \$2, ... etc)
- • **shift** works here as well!
- Return the exit status of the function

Shell Functions

```
allfiles() {  
# true if all args are files  
for x in $*; do  
    if [ ! -f $x ]; then  
        return 1  
    fi  
done  
return 0  
}
```

Shell Functions

- A function must be defined before we can use it
- We call it like any other command
- **allfiles file1 dir1 file2**
- No parentheses in function call
- • Effectively creates a new (temporary) shell
- command

Shell Functions

Beware - there are no local variables!

```
x=      # set x to blank
```

```
f1() {
```

```
x="hello"
```

```
}
```

```
f1 ; echo $x # call f1()
```

Output : "hello"

Solution: sub-shell (**f1**) or bash "**local**"

```
local x="hello"
```
