

CS108 Software Systems: Unix

Spring 2008

Roadmap

Shell Scripting Redux Next week:
Spring break

Simplifying Homework #5

Replacing repetitive commands generating `sort.data`:

```
data : sortcomp
  echo -en "10000\t" > sort.data
  sortcomp -hmqs 10000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo -en "\n50000\t" >> sort.data
  sortcomp -hmqs 50000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo -en "\n100000\t" >> sort.data
  sortcomp -hmqs 100000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo -en "\n500000\t" >> sort.data
  sortcomp -hmqs 500000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo -en "\n1000000\t" >> sort.data
  sortcomp -hmqs 1000000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo -en "\n5000000\t" >> sort.data
  sortcomp -hmqs 5000000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo -en "\n10000000\t" >> sort.data
  sortcomp -hmqs 10000000 | cut -d ' ' -f 3 | tr '\n' '\t' >> sort.data
  echo >> sort.data
```

Simplifying Homework #5

```
VAL1 = 10000
VAL2 = 50000
VAL3 = 100000
...

TRANSP = tr " " "\t"
TRANNL = tr "\n" " "
CUT = cut -d ' ' -f 3,7,11,15
OUTPUT = sort.data

...

data:
    echo -n $(VAL1) ' ' | $(TRANSP) > sort.data
    sortcomp -hmqs $(VAL1) | $(TRANNL) | $(CUT) | $(TRANSP) >> $(OUTPUT)
    echo -n $(VAL2) ' ' | $(TRANSP) >> sort.data
    sortcomp -hmqs $(VAL2) | $(TRANNL) | $(CUT) | $(TRANSP) >> $(OUTPUT)
    echo -n $(VAL3) ' ' | $(TRANSP) >> sort.data
    sortcomp -hmqs $(VAL3) | $(TRANNL) | $(CUT) | $(TRANSP) >> $(OUTPUT)
...

```

Simplifying Homework #5

The script will generate all the data on *stdout*.

```
#!/bin/bash

for size in 10000 50000 100000 500000 1000000 5000000 10000000; do
    echo -en "$size\t"
    sortcomp -hmq $size | cut -d ' ' -f 3 | tr '\n' '\t'
    echo
done
```

The much simplified makefile:

```
data : sortcomp
    sortcomp.sh > sort.data
```

Simplifying Homework #5

Here we use two loops.

```
#!/bin/bash

for size in 10000 50000 100000 500000 1000000 5000000 10000000; do
  echo -n $size
  for alg in h m q s; do
    echo -en '\t'
    sortcomp -$alg $size | cut -d ' ' -f 3 | tr -d '\n'
  done
  echo
done
```

Simplifying Homework #5

To improve flexibility, the array sizes are now command line arguments.

```
#!/bin/bash

for size in "$@"; do
    echo -en "$size\t"
    sortcomp -hmq $size | cut -d ' ' -f 3 | tr '\n' '\t'
    echo
done
```

```
data : sortcomp
      sortcomp.sh 10000 50000 100000 500000 1000000 5000000 > sort.data
```

Full makefile for Homework # 5

```
CXXFLAGS = -g

sortcomp : main.o sorts.o Random.o
    $(CXX) $(CXXFLAGS) -o $@ $^

data : sortcomp
    sortcomp.sh 10000 50000 100000 500000 1000000 5000000 > sort.data

graph:
    gnuplot sort.gnuplot

.PHONY : clean depend

clean :
    rm -f sortcomp *.o sort.data sort.png

depend :
    makedepend -Y *.cc
```

Our Traversal Framework

```
#!/bin/bash

helper () {
    for file in "$@"; do
        if [ -d "$file" ]; then
            helper "$file"/*
        elif [ -f "$file" ]; then
            ...
        fi
    done
}

helper "$@"
```

Putting Traversal to Work

Using our recursive file traversal framework, we can easily write a script to “secure” the permissions of an entire directory structure:

```
#!/bin/bash

helper () {
    for file in "$@"; do
        if [ -d "$file" ]; then
            chmod go-rwx "$file"
            helper "$file"/*
        elif [ -f "$file" ]; then
            chmod go-rwx "$file"
        fi
    done
}

helper "$@"
```

owen:~> `traverse.sh .`

A Few Things to Consider

1. What about “bad” command line arguments?
2. What about files that are not directories or “plain” files?
3. What about empty directories?

“Bad” Arguments

All arguments are interpreted as paths, so how will the script behave if provided non-existent paths?

“Bad” Arguments

All arguments are interpreted as paths, so how will the script behave if provided non-existent paths?

```
if [ -d "$file" ]; then
    chmod go-rwx "$file"
    helper "$file"/*
elif [ -f "$file" ]; then
    chmod go-rwx "$file"
fi
```

Non-existent paths will fail the file type checks, [-d "\$file"] and [-f "\$file"], thus no further commands will involve the “bad” path. The script need not be modified.

Traversing Odd File Types

There are several file types in addition to directories and “plain” files:

- Symbolic links
- Block and character files
- Named pipes

There are [] compatible tests for all the above file types. The only type that must be commonly considered is symbolic links, which can masquerade as directories and “plain” files.

```
odin:~> if [ -d /u ]; then echo Yes; fi
Yes
odin:~> if [ -L /u ]; then echo Yes; fi
Yes
```

Some More Built-in Tests

(See `man bash`)

- `-d file`: True if file exists and is a directory.
- `-f file`: True if file exists and is a regular file.
- `-h file`: (or `-L`) True if file exists and is a symbolic link.
- `-p file`: True if file exists and is a named pipe (FIFO).
- `-r file`: True if file exists and is readable.
- `-s file`: True if file exists and has a size greater than zero.
- `-w file`: True if file exists and is writable.
- `-x file`: True if file exists and is executable.
- `-S file`: True if file exists and is a socket.
- `file1 -ot file2`: True if file1 is older than file2.

A More Cautious Traversal

```
#!/bin/bash

helper () {
  for file in "$@"; do
    if [ -L "$file" ]; then
      continue          #why is this needed?
    elif [ -d "$file" ]; then
      chmod go-rwx "$file"
      helper "$file"/*
    elif [ -f "$file" ]; then
      chmod go-rwx "$file"
    fi
  done
}

helper "$@"
```

Traversing Empty Directories

If the script encounters an empty directory, what happens when it tries to recurse via helper `"$file"/*?`

Traversing Empty Directories

If the script encounters an empty directory, what happens when it tries to recurse via `helper "$file"/*?`

Suppose `$file` has the value `/u/mtaylor/empty`, an empty directory. Pathname expansion produces:

```
helper "$file"/*
```



```
helper /u/mtaylor/empty/*
```

The file `/u/mtaylor/empty/*` doesn't exist. Traversing empty directories is the same as traversing non-existent files, which are already handled correctly.

Another Scripting Task

The traversal framework is easily adapted to other file tasks, such as summing the disk space used by a set of files, similar to `du` or `quota` commands. To do this, we must be able to...

1. Determine the size of a file.
2. Compute a sum.

Command Substitution

There are several commands that can output the size of a file, such as `ls`, but that output must be “captured” by the script.

`$(command)`

The output of *command* is captured as a string.

```
odin:~> stat -c %s ~/www/cs108/syllabus.pdf
12693
odin:~> size=$(stat -c %s ~/www/cs108/syllabus.pdf)
odin:~> echo $size
12693
```

Arithmetic

`bash` can perform simple integer arithmetic via two sets of syntax:

- String variables, explicit arithmetic.
- Integer variables, implicit arithmetic.

Both options allow a basic set of mathematical operators:

+	Plus	-	Minus	*	Multiplication
/	Division	%	Remainder	**	Exponentiation
<<	Bit shift left	>>	Bit shift right	&	Bitwise AND
	Bitwise OR	~	Bitwise NOT	^	Bitwise XOR

String Variable Arithmetic

Any variable may have its value interpreted as a number (for better or worse) or be assigned a number.

`let variable=expression`

or

`variable=$((expression))`

```
odin:~> let x="(10 + 4) / 5 << 1"
odin:~> echo $x
4
odin:~> let y="(x << 2) / 6"
odin:~> echo $y
2
odin:~> echo $((x + y))
6
```

Integer Variable Arithmetic

Variables may be declared integer variables, and any assignment to such a variable is considered a mathematic expression.

```
declare -i variable  
variable=expression
```

```
odin:~> declare -i a  
odin:~> a="(5 * 6) % 4 + 3"  
odin:~> echo $a  
5  
odin:~> a="a << 2"  
odin:~> echo $a  
20
```

File Size Summation Script

```
#!/bin/bash

helper () {
    for file in "$@"; do
        if [ -L "$file" ]; then
            continue;
        elif [ -d "$file" ]; then
            helper "$file"/*
        elif [ -f "$file" ]; then
            totalSize+=$(stat -c %s "$file")
        fi
    done
}

declare -i totalSize=0
helper "$@"
echo $totalSize
```

Yet Another Task

Suppose we wish to compute the total amount of time a user spent logged on in the past month, as reported by the `last` command.

will	pts/0	resnet-27-120.do	Thu	Jun	2	09:23	-	11:35	(02:11)
oscreyes	pts/8	cpe-70-113-94-7.	Thu	Jun	2	06:45	-	09:36	(02:50)
oscreyes	pts/0	cpe-70-113-94-7.	Thu	Jun	2	01:54	-	07:09	(1+05:15)
pgunter	pts/0	cpe-70-113-126-8	Thu	Jun	2	01:01	-	01:02	(00:00)
jdlee	pts/8	adsl-70-243-116-	Wed	Jun	1	21:57	-	22:14	(00:17)
rposky	pts/11	cpe-70-112-217-3	Wed	Jun	1	20:10	-	20:14	(00:03)
will	pts/0	resnet-27-120.do	Wed	Jun	1	19:26	-	00:08	(04:41)

We need to process the output of `last` line by line, summing the minutes, hours, and days of time.

Line Processing in a Script

By using block redirection and the shell's `read` command it's possible to emulate a traditional line processing loop.

To execute a loop in the current shell:

```
while read var; do
    ...
done < <(commands)
```

Processing last

Here we limit ourselves to a single user's past logins as reported by `last`.

```
#!/bin/bash

while read line; do
    echo $line
done < <(last mtaylor | grep \))
```

Variable Expansion Operators

bash can perform several special variable expansions:

<code>\${#variable}</code>	Length of value in <i>variable</i> .
<code>\${variable:index}</code>	Substring of <i>variable</i> starting at <i>index</i> .
<code>\${variable:index:length}</code>	Substring of <i>variable</i> starting at <i>index</i> with length <i>length</i> .

Variable Expansion Operators

$\${variable\#pattern}$	<i>variable</i> minus the shortest prefix matching <i>pattern</i> .
$\${variable##pattern}$	<i>variable</i> minus the longest prefix matching <i>pattern</i> .
$\${variable\%pattern}$	<i>variable</i> minus the shortest suffix matching <i>pattern</i> .
$\${variable%%pattern}$	<i>variable</i> minus the longest suffix matching <i>pattern</i> .
$\${variable/pattern/string}$	Replace first, longest match of <i>pattern</i> in <i>variable</i> with <i>string</i> .

Expansion Operator Examples

Suppose `$var` has the value `(00:07)`

Expression	Result
<code>#{#var}</code>	
<code>\${var:4}</code>	
<code>\${var:1:5}</code>	
<code>\${var#*0}</code>	
<code>\${var##*0}</code>	
<code>\${var%0*}</code>	
<code>\${var%%)*}</code>	
<code>\${var/0*0/}</code>	

Expansion Operator Examples

Suppose `$var` has the value `(00:07)`

Expression	Result
<code>\${#var}</code>	<code>7</code>
<code>\${var:4}</code>	<code>07)</code>
<code>\${var:1:5}</code>	
<code>\${var#*0}</code>	
<code>\${var##*0}</code>	
<code>\${var%0*}</code>	
<code>\${var%%)*}</code>	
<code>\${var/0*0/}</code>	

Expansion Operator Examples

Suppose `$var` has the value `(00:07)`

Expression	Result
<code>\${#var}</code>	7
<code>\${var:4}</code>	07)
<code>\${var:1:5}</code>	00:07
<code>\${var#*0}</code>	
<code>\${var##*0}</code>	
<code>\${var%0*}</code>	
<code>\${var%%)*}</code>	
<code>\${var/0*0/}</code>	

Expansion Operator Examples

Suppose `$var` has the value `(00:07)`

Expression	Result
<code>\${#var}</code>	7
<code>\${var:4}</code>	07)
<code>\${var:1:5}</code>	00:07
<code>\${var#*0}</code>	0:07)
<code>\${var##*0}</code>	7)
<code>\${var%0*}</code>	(00:
<code>\${var%%)*}</code>	(
<code>\${var/0*0/}</code>	(7)

Parsing `last`

```
oscreyes pts/0          cpe-70-113-94-7. Thu Jun  2 01:54 - 07:09 (1+05:15)
```

Based on the variable expansion operators available, which primarily discard portions of data, parsing a line of `last`'s output requires the following steps:

1. Discard everything up to `(`
2. Grab the day count, if present.
3. Discard up to `+`, if a day count was present.
4. Grab the hour count.
5. Discard up to `:`
6. Grab the minute count.

Parsing last

```
oscreyes pts/0          cpe-70-113-94-7. Thu Jun  2 01:54 - 07:09 (1+05:15)
```

```
#!/bin/bash

while read line; do
    line=${line##*()}          # (1) Discard until (
    if [ ${#line} -gt 6 ]; then
        days=${line%+*}      # (2) Grab day count.
        line=${line##*+}     # (3) Discard until +
    else
        days=0
    fi
    hours=${line%:*}         # (4) Grab hour count.
    hours=${hours#0}
    line=${line#*:}          # (5) Discard until :
    minutes=${line%)*}      # (6) Grab minute count.
    minutes=${minutes#0}
    echo $days $hours $minutes
done < <(last mtaylor | grep \))
```

Total Login Time Script

```
#!/bin/bash

declare -i totalMinutes=0
while read line; do
    line=${line##* }
    if [ ${#line} -gt 6 ]; then
        totalMinutes+=24*60*${line%+*}
        line=${line#+*}
    fi
    hours=${line%:*}
    totalMinutes+=60*${hours#0}
    line=${line#*:}
    minutes=${line%)*}
    totalMinutes+=${minutes#0}
done <<(last mtaylor | grep \))
echo $totalMinutes minutes.
```

Arrays

`bash` supports one-dimensional arrays. You can assign array elements individually or all together:

```
odin:~> MyArray[0]=Hi
odin:~> MyArray[1]=There
odin:~> MyArray=(Each word becomes an element)
```

You can also access elements individually or all together:

```
odin:~> echo ${MyArray[0]}
Each
odin:~> echo ${MyArray[*]}
Each word becomes an element
```

And access the array length:

```
odin:~> echo ${#MyArray[*]}
5
```

Array Usage Script

One common usage of arrays is to capture and parse data.

```
#!/bin/bash

getSuffix () {
    if (($1 >= 11 && $1 <= 13)); then
        echo "th"
    else
        echo ${suffixes[${1%10}]}
    fi
}

suffixes=(th st nd rd th th th th th th)
data=$(date +%l:%M %p %A %e %B')
echo "It is ${data[0]} ${data[1]} on ${data[2]},\
the ${data[3]}$(getSuffix ${data[3]}) of ${data[4]}."
```

Making C Programmers Happy

We can rewrite positional parameters...

```
#!/bin/bash

for arg in "$@"; do
    echo "$arg"
done
```

As array elements...

```
#!/bin/bash

argv=("$@")
argc=${#argv[@]}
for ((i = 0; i < argc; i++)); do
    echo ${argv[$i]}
done
```

Debugging Shell Scripts

The `bash` shell can be started with a couple of options that help in debugging scripts:

- `-n` Don't execute commands, just check syntax.
- `-v` Echo commands before execution.
- `-x` Echo commands after processing.

These options can be specified as part of the script interpreter line, e.g. `#!/bin/bash -x`.