Introduction to the Bourne Again SHell (bash)

Overview

- Bash basics, variables, looping and redirection.
- grep (text searching)
- awk (text processing)
- sed (text processing/stream editor)
- bc (CLI calculator)
- Shell scripts
- A real life example.
- Other languages of interest

Some required files

```
wget http://ceclnx01.cec.miamioh.edu/~taylors6/acm_bash_demo.tar.xz
tar xf acm_bash_demo.tar.xz
cd acm_bash_demo
```

bash

- Let's first make sure everyone is using bash.

```
echo $SHELL
        /bin/bash
```

- What happened here?
  - The bash built-in command `echo` takes input from it's argument list, and prints it to standard-out.
  - Why do we not see “`$SHELL`” printed?
- If you aren't using bash, you will need to run the following:

```
/bin/bash --login
```

A bit more about variables

- Variables are not strictly typed.
- By default, variables are treated as strings.

```
a=1
echo $a
      1
a=$a+2
echo $a
      1+2
```

A bit more about variables (2)

- We can tell bash that it should treat our variable like an integer.

```
a=1
echo $a
      1
a=$((a+2))
echo $a
      3
```

Quotes, and the for loop

- Using the “for var in list” syntax.
  - var is the name of the iterating variable
  - list is a whitespace delimited list of text elements.

```
for file in `ls`; do
echo 'file name: $file'
done
```

Quotes, and the for loop

- Using the “for var in list” syntax.
  - var is the name of the iterating variable
  - list is a whitespace delimited list of text elements.

```
for file in `ls`; do
echo 'file name: $file'
done
```

- Uh-oh, this didn't work. What went wrong?

Quotes in bash

- There are 3 types of quote pairs:
  - The double, forward quote "
  - The single, forward quote '
  - The single, back quote `
- Back quotes tell bash to execute whatever lies inside as if it were typed in the command prompt directly, and paste the output where the back quotes were.
- In general, single and double quotes are interchangeable in bash, and are used for grouping strings.
  - Single quotes don't permit variable evaluation, while double quotes do

Redirection

- Most of the utilities you will encounter operate on text streams.
  - stdin: Standard input
  - stdout: Standard output
  - stderr: Standard error (second output stream)
- By default, your typing is standard in, and standard out/err is what is displayed on the screen.
- But, what if we want to read/write a file instead?
- What about hooking the standard out of one program to the standard in of another?

Redirection (2)

- cat is a utility which simply reads from stdin and writes to stdout.

```
cat < readme.txt
```

```
echo "Here's some text" > text.txt
```

- Bash provides us with several stream redirection operators
  - < pulls data from a file and puts it on stdin of a program
  - > pulls data from stdout of a program, and puts it in a file
  - | (pipe) pulls data from stdout of the left hand side program, and puts it on stdin of the right hand side program

```
cat < readme.txt | cat > readme_copy.txt
```

Redirection (3)

- In the previous example, | (pipe) is pretty useless, however, it will become much more useful as we learn some of the other UNIX utilities.
- There are a lot more redirection operators that have varying degrees of usefulness.
- What happens to readme_copy.txt if you run again:

```
cat < readme.txt > readme_copy.txt
```

- What happens if you instead run:

```
cat < readme.txt >> readme_copy.txt
```

Redirection (4)

- >> - pulls data from standard out and appends it to the end of a file
- 2> – 2 is standard error, this redirects standard error to a file, instead of standard out.
- 2>&1 - & is a stream concatenate. This is a special operator that redirects the standard error of a program to the standard out.
- > /dev/null – This redirects the standard output to a character device sink that goes nowhere. Useful if you don't want to see the output of your program

Some other UNIX commands:

- Bash is great, but there is only so much that can be done with built in functions
- Let's look at some other powerful commands
- grep
- awk
- sed
- bc

Grep

- Stands for “Globally search a Regular Expression and Print”
- Basically, we use this as a search filter, printing if the string we're searching for (an argument to grep) is found in the standard input to grep.
- We have the option to use regular expressions, but looking for plain text works just fine too.

```
ls | grep txt
```

```
ps aux | grep uniqueID
```

Awk

- Named for it's creators.
- Awk is a text processing and scripting language on its own.
  - Awk operates on a line-by-line basis.
  - Each line will be treated as whitespace delimited fields
- For simplicity, we will only be learning about print/printf.

Awk (2)

- Let's look at the data fields from bash's list command:

```
ls -lh
```

```
-rw-rw-r-- 1 steve steve  57K Mar 12 23:02 album.html
```

- There appear to be nine fields, which awk will number as $1 through $9 in our script.

Awk (2)

- Let's look at the data fields from bash's list command:

```
ls -lh
```

```
-rw-rw-r-- 1 steve steve  57K Mar 12 23:02 album.html
```

- There appear to be nine fields, which awk will number as $1 through $9 in our script.
- We want to print just the name of the file, and the size.
  - The following awk program should do the trick

```
'{ print $9 " " $5; }'
```

- Let's pipe the output from ls to the awk program:

```
ls -lh | awk '{ print $9 " " $5; }'
```

Awk (3)

- That output is alright, but I think we can do better.
- Let's use awk with a formatter to make cleaner code, and prettier output

```
ls -lh | awk '{ printf "File %s is %s bytes.\n", $9, $5; }'
```

- Notice, we have an anomaly on the very first line. This is fixable, but requires a new command

Tail & Awk (4)

- Tail will help us select which lines get output.

```
ls -lh | tail -n +2 | awk '{ printf "File %s is %s bytes.\n", $9, $5; }'
```

- By piping through tail with the “-n +2” argument, we are telling tail we want only lines beginning at line 2 and later.
  - If we instead asked for “-n 2”, we would get just the last 2 lines.
- You might also look up head, which is the complement of tail.

Sed

- Sed stands for Stream EDitor.
- Like grep, we can use regular expressions with it.
- This adds complexity, so I will try to keep things as minimalistic as possible, although they will still look awful.
- For 99% of the sed scripts I write, I am only using it for substituting one string for another.

```
cat < example.c
```

Sed (2)

- Let's look at example.c. It prints my name, but what if we want it to print yours?
- Can we do this “find & replace” without opening a heavy weight text editor?

```
sed -i 's/Steve/Your Name/' example.c
```

- Take a look and make sure it worked.

Sed (3)

- Remember all those fancy comments from example.c?
- Those are c++ style comments, which have been accepted into the c99/c11 standard.
- But what if we don't have a c99 compiler?
- Try to compile the code:

  ```
  gcc -ansi example.c
  ```

- Look at all the errors... but what's wrong?

Sed (4)

- '//' has no meaning to an old c89/K&R compiler. If you want comments, they're gonna be c-style /* and */ pairs.
- Surely you can't expect me to change every line from // to /* */?
- We want the stuff after // to go between the /* and */ pair now.
- This isn't a simple find and replace operation like last time.

Sed (5) and RegEx

- We can use sed, of course!
- We need to introduce two confusing concepts though:
  - The Regular Expression
  - The match/backreference

```
sed -i 's|//\(.*\)|/* \1 */|' example.c
```

Sed (6) and RegEx

- We can use sed, of course!
- We need to introduce two confusing concepts though:
  - The Regular Expression
  - The match/backreference

```
sed -i 's|//\(.*\)|/* \1 */|' example.c
```

  - s – This is the command, substitute
  - | - This is the delimiter, since we have '/' as a character we're searching for
  - .* - This is the regular expression, means everything, repeated
  - \(, \) and \1 – These set up the match in part 2, and store it for later use as the backreference in part 3

Sed (6), gcc and program execution

- Let's try to compile the fixed version

  ```
  gcc -ansi example.c
  ```

- And execute:

  ```
  ./a.out
  ```

BC (basic calculator)

- Bash can do some basic math with integers.
- If we want floating-point, we need a new tool.
- Let's start an interactive session with bc and load the standard math library.

  ```
  bc -l
  ```

- Let's ask it a simple question:

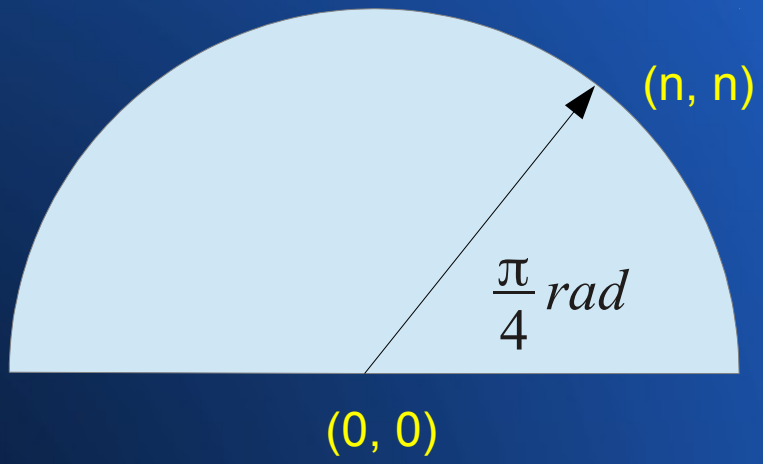
  ```
  4*a(1)
  ```

BC (2)

- Look familiar?
- So, what did we do?

BC (3)

- The function a() is taking the arc-tangent of the number in parentheses.



BC (4)

- Other functions in the BC math library
  - `a(n)` – arctangent
  - `s(n)` – sine
  - `c(n)` – cosine
  - `l(n)` – natural logarithm
  - `e(n)` – exponential
  - `sqrt(n)` – square root
  - `j(n,x)` – Bessel function order *n* of *x*

The shell script

- We can take everything we do interactively with bash, and put it into a convenient, executable file, a script.
- Let's try something simple, print your name:
- Open a text editor:

```
nano name.sh
```

- Add these two lines:

```
#!/bin/bash
echo "My name is Your_Name"
```

- Save and exit.

The shell script (2)

- First, we must make the new script executable.

  ```
  chmod +x name.sh
  ```

- Now, we can run it just like the c program.

  ```
  ./name.sh
  ```

The shell script (3)

- We can also pass command line parameters to shell scripts
- These show up as special variables inside the script
  - `$#` is the count of command line parameters, think of it like argc in a c program
  - `$1`...`$`*n* are the parameters, think of them like argv[1] through argv[*n*] in a c program.
  - `$@` is the entire list of `$1`...`$`*n*
- Try changing the second line in your script to:

```
echo "My name is $1"
```

- Now, you should run it with an argument:

```
./name.sh Steve
```

Run time performance

- Execution time is an important measure of a program's performance. To get this statistic, we can use the bash time function.
- We will be using the factors.c program for run time analysis.

```
gcc -ansi factors.c
```

- Let's examine the components of runtime.sh

runtime.sh

```
for TRY in {1..5}; do
  { time ./a.out $1 > /dev/null; } 2>&1 | \
    grep real | \
    awk '{ print $2; }' |
    sed 's/\(.*\)m\(.*\)s/ \1 * 60 + \2 /' | \
    bc >> times.txt
done
```

runtime.sh

```
TOTAL=0
TRIES=0
for TIME in `cat < times.txt`; do
  TOTAL=`echo "$TOTAL+$TIME" | bc -l`
  TRIES=$((TRIES+1))
done

echo "The average run time is " `echo "$TOTAL/
$TRIES" | bc -l`
```

Run time performance (2)

- Let's test this new script.
- Check that the script is executable:

```
ls -l runtime.sh
```

```
-rwxrwxr-x 1 steve steve 1222 Mar 13 15:35 runtime.sh
```

- If not, set the executable bit:

```
chmod +x runtime.sh
```

- And try it out with a big numer

```
./runtime.sh 12345678
```

```
The average run time is  .2578
```

References

- http://www.tldp.org/LDP/abs/html/
- http://www.grymoire.com/Unix/Sed.html
- http://www.tutorialspoint.com/unix/unix-regular-expressions.htm
- http://www.gnu.org/software/sed/manual/sed.html
- http://www.grymoire.com/Unix/Awk.html

Other important languages/editors/commands

- Perl
- Python
- CMake
- GNU Make
- emacs
- vim/ed
- cut/paste/join
- sort
- date