More Linux Commands

EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

Repetition * ? + summary

Regex	Meaning
a*	0 or more a
a?	0 or 1 a
a+	1 or more a

ab*c matches ac abc abbc abbbc abbbbc ....

ab?c matches ac abc

ab+c matches abc abbc abbbc abbbbc

- Don't get confused with filename wildcard *

  ls ba* ba followed by 0 or more any char -- anything

  ls a*.c a followed by 0 or more any char – anything, then .c

Regular Expression Summary

Pattern	Maning	Example
c	Non-special, matches itself	'tom'
^	Start of line	'^ab'
$	End of line	'ab$'
.	Any single character	'.nodes'
[…]	Any single character in []	'[tT]he'
[^…]	Any single character not in []	'[^tT]he'
R*	Zero or more occurrences of R	'e*'
R?	Zero or one occurrences of R (egrep)	'e?'
R+	One or more occurrences of R (egrep)	'e+'
R1R2	R1 followed by R2	'[st][fe]'
R1|R2	R1 or R2 (egrep)	'the|The'

anchored

repetition

Regular Expressions: Repetition Ranges, Subexpressions

- Some examples

Regular Expression	Matches
"a*"	ZERO or more 'a'
"ba*"	b, ba, baa, baaa, baaaa, ...
"[ab]*"	Ø, a, ab, aaa, ababb, bbb, ... zero or more characters, each character an 'a' or 'b'
"[^0-9]*"	Ø, A, ABC, zw$nn, ... zero or more characters, no character a digit
"a*b*"	Ø, a, aaa, aaab, abbb, b, bbb, ... zero or more 'a', followed by zero or more 'b'

- Don't get confused with filename wildcard *

  ls a*.c a followed by 0 or more any char -- anything

  ls ba*.c

Removing Duplicate Lines: uniq

- The uniq utility displays a file with all of its identical adjacent lines replaced by a single occurrence of the repeated line.
- Here's an example of the use of the uniq utility:

```
$ cat  animals          # look at the test file.
cat  snake
monkey snake
dolphin elephant
dolphin elephant
goat elephant
pig pig
pig pig
monkey pig
pig pig
```

```
$ uniq  animals    # filter out duplicate adjacent lines.
cat  snake
monkey snake
dolphin elephant
goat elephant
pig pig
monkey pig
pig pig
```

sort

- sorts a file in ascending or descending order based on one or more fields.
- Individual fields are ordered lexicographically, which means that corresponding characters are compared based on their ASCII values.

-t field separator (default is blank or tab)
-r descending instead of ascending

-n numeric sort
-f ignore case
-M month sort (3 letter month abbreviation)
-k key sort on column

sort Example

```
$ cat  data.txt
John   Smith      1222  26  Apr   1956
Tony   Jones      1012  20  Mar  1950
John   Duncan     2       20  Jan  1966
Larry   Jones     3223  20  Dec 1946
Lisa    Sue       1222  4   Jul    1980
```

```
$ sort data.txt    # cat data.txt | sort
John  Duncan     2       20  Jan  1966
John  Smith      1222  26  Apr  1956
Larry Jones      3223  20  Dec 1946
Lisa   Sue       1222  4    Jul   1980
Tony Jones       1012  20  Mar 1950
```

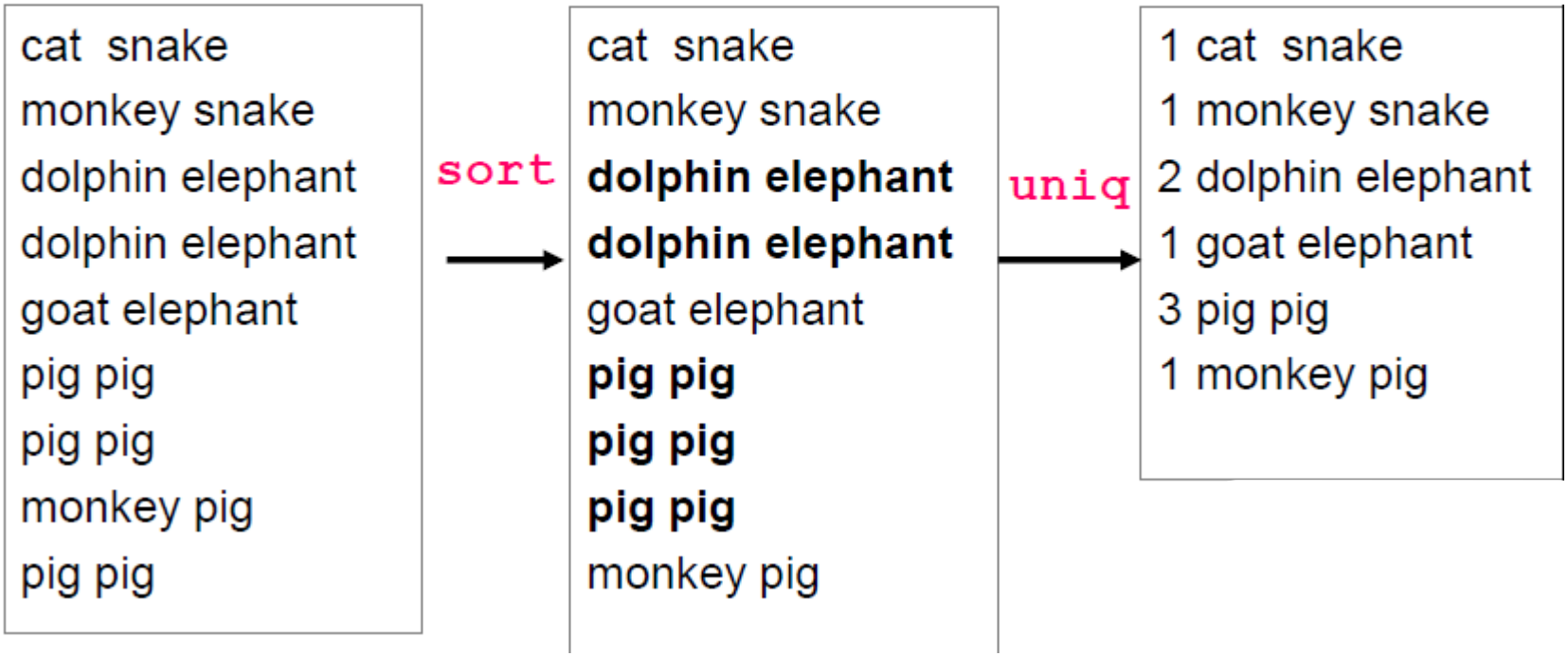
```
$ sort -r data.txt    # descending
Tony Jones       1012  20 Mar 1950
Lisa   Sue       1222  4  Jul   1980
Larry Jones      3223  20 Dec 1946
John  Smith      1222  26 Apr  1956
John  Duncan     2       20 Jan  1966
```

Whole lines are ordered lexicographically

sort+ uniq

- uniq is a little limited but we can combine it with sort

```
sort | uniq -c
```



Comparing Files: cmp, diff

- There are two utilities that allow you to compare the contents of two files:

- cmp, which finds the first byte that differs between two files
- diff, which displays all of the differences and similarities between two files

Comparing Files: cmp, diff

- There are two utilities that allow you to compare the contents of two files:

- cmp, which finds the first byte that differs between two files
- diff, which displays all of the differences and similarities between two files

---

- Testing for sameness: cmp

- The cmp utility determines whether two files are the same.

```
$ cat lady1                                  # look at the first test file.
Lady of the night,
I hold you close to me,
And all those loving words you say are right.

$ cat lady2                                  # look at the second test file.
Lady of the night,
I hold you close to me,
And everything you say to me is right.

$ cmp lady1  lady2                           # files differ.
lady1 lady2 differ: char 48, line 3
$ _
```

File Differences: diff

- The **diff** utility compares two files and displays a list of editing changes that would convert the first file into the second file.

```
$ diff lady1 lady2        # compare lady1 and lady2.
3c3
< And all those loving words you say are right.
…
> And everything you say to me is right.
$ _
```

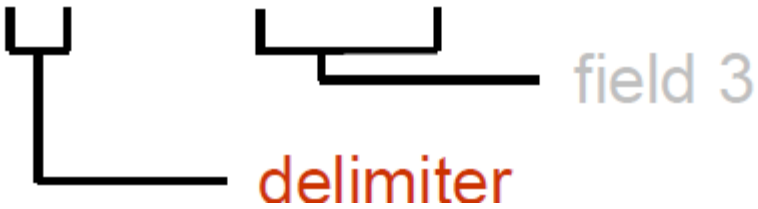
```
$ gcc yourCode;
$ a.out > yourOutput;
$ cmp yourOutput sampleOutput;
```

cut deal with fields (columns)

-d -f

- Used to split lines of a file
- A line is split into fields
- Fields are separated by delimiters/separators
- A common case where a delimiter is a space:
  - Default is tab, (not " ") need to set it for blank
    -d " "
  - cut -f3 -d" "

- **hello there world**



```
$ cat data.txt          # assuming tab as delimiter
John        Smith       1222        26     Apr  1956
Tony        Jones       1012        20     Mar  1950
John        Duncan      1111        20     Jan  1966
Larry        Jones      1223        20     Dec  1946
Lisa        Sue         1222        15     Jul  1980

$ cut -f 1  data.txt    # show field 1,  tab as delimiter
John
Tony
John
Larry
Lisa

$ cut -f 1,3  data.txt

$ cut -f 1-3  data.txt
```

```
$ cat data.txt          # assuming tab as delimiter
John        Smith       1222        26     Apr  1956
Tony        Jones       1012        20     Mar  1950
John        Duncan      1111        20     Jan  1966
Larry        Jones      1223        20     Dec  1946
Lisa        Sue         1222        15     Jul  1980

$ cut -f 1  data.txt     # show field 1,  tab as delimiter
John
Tony
John
Larry
Lisa

$ cut -f 1,3  data.txt
John    1222
Tony    101
John    1111
Larry    1223
Lisa    1222

$ cut -f 1-3  data.txt
John    Smith    1222
Tony    Jones    101
John    Duncan   1111
Larry   Jones    1223
Lisa    Sue      1222
```

find Utility

find pathList expression

- finds files starting at pathList
- finds files descending from there

- Allows you to perform certain actions
  - e.g., copying (cp), renaming (mv), deleting (rm) the files

"Find all the c files and make a backup of them/rename to .bak"

```
find . -name "*.c"  -exec mv {} {}.bak \;
```

"Find all the Java class files and delete them"

```
find . -name "*.class"  -exec rm {} \;
```

find Utility

- -name *pattern*
  True if file's name matches *pattern*, which include shell metacharacters * ? [ ]

- -mtime *count*
  True if the content of the file has been modified within *count* days

- -atime *count*
  True if the file has been accessed within *count* days

- -ctime *count*
  True if the contents of the file have been modified within *count* days or any of its file attributes have been modified

- -exec *command*
  True if the exit code = 0 from executing the command.
  - *command* must be terminated by \;
  - If {} is specified as a command line argument, it is replaced by the file name currently matched

find example

- $ find / -name x.c # search for file x.c in the entire file system
- $ find . -mtime 14 # lists files modified in the last 14 days in current and subdirectories
- $ find . -name '*.bak' # "*.bak" search for all bak files
- $ find . -name 'a?.c' # "a?.c" find all a?.c

  a1.c
  a2.c a2.c.bak
  a3.c a3.c

find example

- $ find . -name '*.bak' -exec rm {} \;
  # remove all files that end with .bak
- $ find . -name 'a2.c' -exec cp {} a2.c.bak \;
  # find a2.c and make a copy called a2.c.bak
- $ find . -name 'a?.c' -exec mv {} {}.bak \;
  # find ax.c and then rename it to ax.c.bak
- $ find . -name '*.c' -exec cp {} {}.2019W \;
  # find all c files xx.c and then cp it to xx.c.2019W

Processes

- Each command/utility involves a process
  - ls, cd, pwd, gedit, gcc …
  - Unix can execute many processes simultaneously.

- When a process ends, there is a return value aka exit code associated with the process outcome
  - a non-negative integer. ≥ 0

    0 means success

    > 0 represents various kinds of failure

    Opposite to C

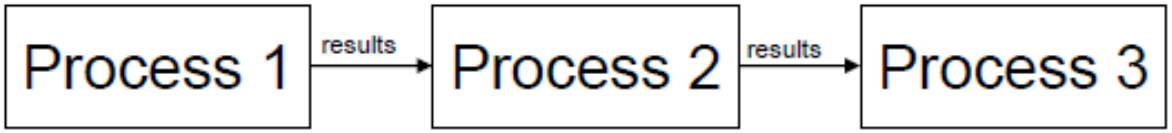
    

  - The return value is passed to the parent process

    Stored in system variable $? (Usually used in shell script)

Process communication: Unix Pipes

- A special mechanism called a "pipe" built into the heart of UNIX to support cascading utilities.
- A pipe allows a user to specify that the output of one process is to be used as the input to another process.
- Two or more processes may be connected in this fashion, resulting in a "pipeline" of data flowing from the first process through to the last.



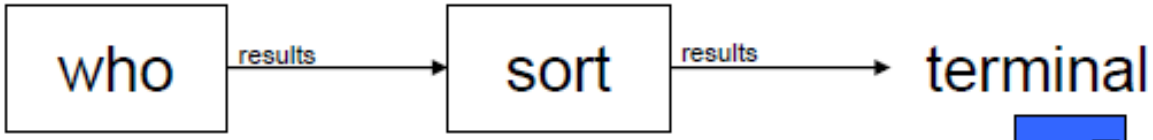
Pipeline Example

- A utility called **who** outputs an unsorted list of current users. Another utility called **sort** outputs a sorted version of its input.

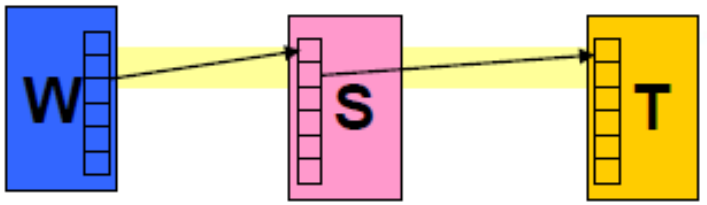
  $ who

  $ sort input.txt or sort < input.txt

- These two utilities may be connected together with a "pipe" so that the output from **who** passes directly into **sort**, resulting in a sorted list of users. | does this job.



$ who | sort



Shell Variables

- Different types of variables
  - Environment variables: HOME, PATH…
  - Parameter variables: $1, $2, …
  - User-defined variables: student, file, x, ..
- Set PATH variable

  PATH=$PATH:.:~/eecs2031/bin

  export PATH # make PATH an environment variable, which is inherited by all subprocesses
- Example:

  [indigo 301] % bash #run bash

  [indigo 301]$ x=1

  [indigo 301]$ echo $x

Shell parameter variables

- If your script is invoked with parameters, shell sets **parameter variables**
  - $#: number of parameters
  - $0, $1, …: command/script name, first/second parameters
  - $*, $@: Represents all command-line arguments at once. They can be used to pass command-line arguments to a program being run by a script or function.

Shell parameter variables (2)

- **“$*”**: all command-line arguments **as a single string**. Equivalent to "$1$2 …".
  - printf "The arguments were %s\n" "$*"

- **"$@"**: all command-line arguments **as separate, individual strings**. Equivalent to "$1" "$2" ….
  - best way to pass arguments on to another program, since it preserves any whitespace embedded within each argument.
  - lpr "$@" #print each

Positional Parameters

Positional Parameter	What It References
$0	References the name of the script
$#	Holds the value of the number of positional parameters
$*	Lists all of the positional parameters
$@	Means the same as $*, except when enclosed in double quotes
"$*"	Expands to a single argument (e.g., "**$1 $2 $3**")
"$@"	Expands to separate arguments (e.g., "**$1**" "**$2**" "**$3**")
$1 .. ${10}	References individual positional parameters
set	Command to reset the script arguments

Set command

- **set** command, a shell builtin command
  - display current variables, “set”
  - **set shell options**, “set –f”, “set –n” ..
  - set position parameters (no options),

    [indigo 301] $ set Hello world;

    [indigo 301] $ echo $1, $2

    Hello world

- Combine command substitution and set command

  [indigo 301]$ set `who am i`

  [indigo 301] $ echo Welcome, $1! You logged in from $5.

  [indigo 301] $ set `date`

  [indigo 301] $ echo The year is $6

  The year is 2023

Set command

When we run the *set* command without any arguments, it returns a list of all shell settings

```
indigo 7 $ set
BASH=/cs/local/bin/bash
BASHOPTS=cmdhist:complete_fullquote:expand_aliases:extquote:force_fignore:hostco
mplete:interactive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="4" [2]="20" [3]="1" [4]="release" [5]="x86_64-redhat
-linux-gnu")
BASH_VERSION='4.4.20(1)-release'
COLUMNS=80
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/20800/bus
DEBUGINFOD_URLS=https://debuginfod.centos.org/
DIRSTACK=()
EDITOR=vi
EUID=20800
GROUP=faculty
GROUPS=()
HISTFILE=/cs/home/wangsong/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/cs/home/wangsong
HOST=indigo
```

Set -f

- *-f* prevents us from using wildcards to search for filenames or strings.

```
indigo 16 $ set -f
indigo 17 $ ls *.txt
ls: cannot access '*.txt': No such file or directory
```

set -x

- print each command or pipeline before executing it, preceded by a special prompt (usually +)

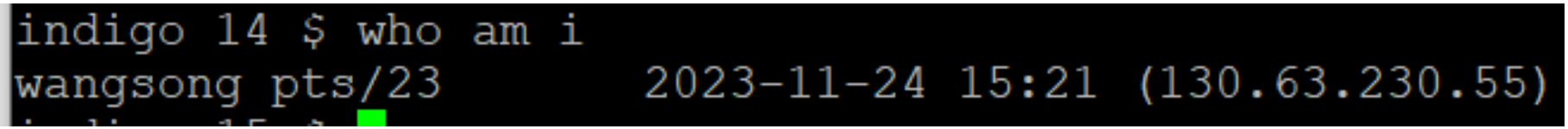
```
#!/bin/bash
set -x
n=3
while [ $n -gt 0 ]; do
    n=$[ $n-1 ]
    echo $n
    sleep 1
done
```

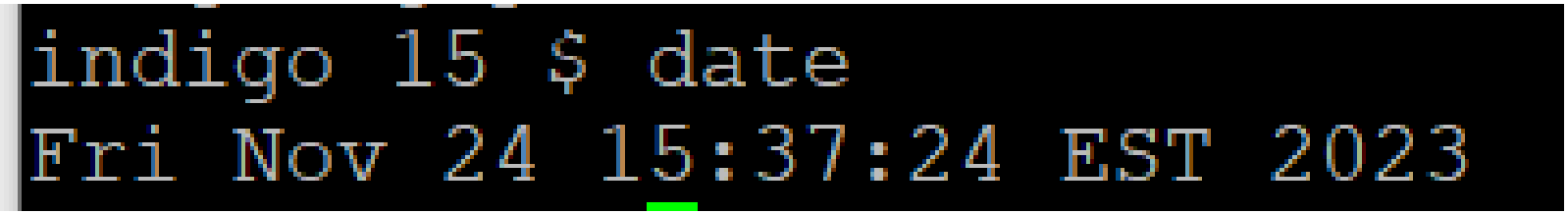
set –x

- print each command or pipeline before executing it, preceded by a special prompt (usually +)

```
#!/bin/bash
set -x
n=3
while [ $n -gt 0 ]; do
    n=$[ $n-1 ]
    echo $n
    sleep 1
done
```

```
$ bash debugging.sh
+ n=3
+ '[' 3 -gt 0 ']'
+ n=2
+ echo 2
2
+ sleep 1
+ '[' 2 -gt 0 ']'
+ n=1
+ echo 1
1
+ sleep 1
+ '[' 1 -gt 0 ']'
+ n=0
+ echo 0
0
+ sleep 1
+ '[' 0 -gt 0 ']'
```


indigo 14 $ who am i
wangsong pts/23 2023-11-24 15:21 (130.63.230.55)


indigo 15 $ date
Fri Nov 24 15:37:24 EST 2023

```
$ set -- hello "hi there" greetings #Set new positional parameters
$ echo there are $# total arguments
#Print the count
there are 3 total arguments
$ for i in $*                    #Loop over arguments individually
> do echo i is $i
> done
i is hello #Note that embedded whitespace was lost
i is hi
i is there
i is greetings
```

```
$ set -- hello "hi there" greetings #Set new positional parameters
$ for i in "$*"                    # With quotes, $* is one string
> do echo i is $i
> done
i is hello hi there greetings
```

```
$ for i in "$@" #With quotes, $@ preserves exact argument values
> do echo i is $i
> done
i is hello
i is hi there
i is greetings
$
```

User defined variables

- Declare variables by using them, e.g.,

```
[indigo 301]$ for letter in a b c
> do
> echo "Letter $letter"
> done
Letter a
Letter b
Letter c
```

Read variable value from input

```
[indigo 301] $ read timeofday
Morning
[indigo 301] $ echo Good $timeofday!
Good Morning!
[indigo 301] $ read greeting
Good morning          # don't need to quote
[indigo 301] $ echo $greeting
[indigo 301] $ Good morning
[indigo 301] $ echo "$greeting" is \$greeting
```

What will be the output ?

Command Substitution

- **Command substitution**: substitute output of a command (a string) into another context, i.e., command
- Syntax: enclose command using backquote, or $()
  - As an argument for another command
    - rm `ls *.o` ## same as rm *.o
  - To set a variable
    - time1=$(date); echo $times1 ## set the output of date to variable times
  - To be used in “for” construct

```
for file in `ls *`; do          ## for every file in current directory, do something
...
done
```

Variable's default type: string

- Variables values are stored as strings

```
[indigo 301] $ number=7+5
[indigo 301] $ echo $number
7+5

[indigo 301] $ x=2; y=3
[indigo 301] $ z1=x+y; z2=$x+$y

[indigo 301] $ echo $z1 $z2
# What will be the output?
```

Arithmetic Evaluation

- arithmetic expression:

```
[indigo 301] $ x=1
[indigo 301] $ x=$[$x+1]  ## x now has value of 2
[indigo 301] $ y=$((2*$x+16)) ## y now has value of 20
```

  - Note: spaces around operators optional
  - Complex expressions supported
- **No spaces around equals sign, as with any bash variable assignment**

Table 6-4. Arithmetic operators

Operator	Meaning	Associativity
++ --	Increment and decrement, prefix and postfix	Left to right
+ - ! ~	Unary plus and minus; logical and bitwise negation	Right to left
* / %	Multiplication, division, and remainder	Left to right
+ -	Addition and subtraction	Left to right
<< >>	Bit-shift left and right	Left to right
< <= > >=	Comparisons	Left to right
== !=	Equal and not equal	Left to right
&	Bitwise AND	Left to right
^	Bitwise Exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND (short-circuit)	Left to right
||	Logical OR (short-circuit)	Left to right
?:	Conditional expression	Right to left
= += -= *= /= %= &= ^= <<= >>= |=	Assignment operators	Right to left

From highest precedence to lowest
Relational operators (<, <=, …) produces a numeric result that acts as a truth value

Arithmetic Evaluation (2)

- Or use command expr (less efficient)

```
[indigo 301] $ x=`expr $x + 1`          # increment x by 1

[indigo 301] $ x=$(expr $x \* 2)  ## need to escape *, (, )
```

- **No spaces around equals sign, as with any bash variable assignment**

- **E.g., convert 38 F to Celsius degree**
  - **Rule: (N -32)*5/9**

Declare variable

- One can explicitly declare a variable:
  declare OPTION(s) VARIABLE=value
- Option
  - -a: variable is an array
  - -f : use function names only
  - -i: variable is to be treated as an integer; arithmetic evaluation is performed when variable is assigned a value
  - -l: when assigned a value, all upper-case characters are converted to lower-case.
  - -r Make names readonly. These names cannot then be assigned values by subsequent assignment statements or unset.
  - …

Example of numerical variable

```
[indigo 301] $ declare -i x ## x will be an integer not a
  string
[indigo 301] $ x=10
[indigo 301] $ x=x+1
[indigo 301] $ echo $x
11
[indigo 301] $ x=30
30
[indigo 301] $ x=x*2
[indigo 301] $ echo $x
60
```

Control Structures & Conditions

- Control structures in bash
  - if … then … fi
  - if … then … else … fi
  - if … then …elif … else … fi
  - for … in … do … done
  - while … do … done
  - until … do … done
  - case … in … esac
  - break, continue

- Conditions (tests): used in if structures, while, until structures, similar to boolean expression in C/C++
  - Dots shown in red are to be replaced with conditions

Example

```
echo -n "Enter your age: "
read age
if [ $age -lt 18 ]
then
        echo "You need to be 18 or older to apply for
  account"
else
         echo "Choose your preferred account
  name"
fi
```

If-statement

```
if condition
then
      commands
fi
```

```
if condition
then
  commands1
else
 commands2
fi
```

```
if condition1
then
   commands1
elif condition2
  commands2
else
  commands3
fi
```

Can be repeated…

```
case . . . in . . .
esac
```

Conditions in shell

- exit status of a command, script or shell function, e.g.,

  if **diff file1 file2 >& /dev/null # if file1 and file2 are the same**

  …

- **test** command: used to perform a variety of test, e.g., test file attributes, compares strings and numbers.

  if **test –e tmp.o** ## if there is file named test.o

  …

- **Compound condition**: combine above using ! (negation), && (and), || (or)

  if **!grep pattern myfile > /dev/null**

  …

Exit status command/script/function

- Exit Status: every command (built-in, external, or shell function) returns a small integer value when it exits, to the program invoked it.
  - Convention: command/program returns a zero when it succeeds and some other status when it fails

- How to return value from shell script?
  - exit command, syntax

  exit [exit-value]
  - Return an exit status from a shell script to its caller
  - If exit-value is not given, exit status of last command executed will be returned.
  - ? A special variable stores exit status of previous command.

Table 6-5. POSIX exit statuses

Value	Meaning
0	Command exited successfully.
> 0	Failure during redirection or word expansion (tilde, variable, command, and arithmetic expansions, as well as word splitting).
1–125	Command exited unsuccessfully. The meanings of particular exit values are defined by each individual command.
126	Command found, but file was not executable.
127	Command not found.
> 128	Command died due to receiving a signal.

test command

- Used to perform a variety of test in shell scripts, e.g., test file attributes, compares strings and numbers.
- Provide no regular output, used exclusively for its exit status
- Syntax:

test expression
[expression]

Note: space between [,] and expression …

Operator	True if ...
string	*string* is not null.
-b *file*	*file* is a block device file.
-c *file*	*file* is a character device file.
-d *file*	*file* is a directory.
-e *file*	*file* exists.
-f *file*	*file* is a regular file.
-g *file*	*file* has its setgid bit set.
-h *file*	*file* is a symbolic link.
-L *file*	*file* is a symbolic link. (Same as –h.)

`-n` *string*	*string* is non-null.
`-p` *file*	*file* is a named pipe (*FIFO* file).
`-r` *file*	*file* is readable.
`-S` *file*	*file* is a socket.
`-s` *file*	*file* is not empty.
`-t` *n*	File descriptor *n* points to a terminal.
`-u` *file*	*file* has its setuid bit set.
`-w` *file*	*file* is writable.
`-x` *file*	*file* is executable, or *file* is a directory that can be searched.
`-z` *string*	*string* is null.

s1 = *s2*	Strings *s1* and *s2* are the same.
s1 != *s2*	Strings *s1* and *s2* are not the same.
n1 -eq *n2*	Integers *n1* and *n2* are equal.
n1 -ne *n2*	Integers *n1* and *n2* are not equal.
n1 -lt *n2*	*n1* is less than *n2*.
n1 -gt *n2*	*n1* is greater than *n2*.
n1 -le *n2*	*n1* is less than or equal to *n2*.
n1 -ge *n2*	*n1* is greater than or equal to *n2*.

Numerical tests work on integers only.

Test status of file: file conditionals

- **File conditionals**: unary expressions examining status of a file

```
if test –e /etc/.bashrc   # same as if [ -e /etc/.bashrc ]
    # do something if /etc/.bashrc exists
then
      # do something else if it doesn't
fi
```

- More testing
  - **-d** file: true if the file is a directory
  - **-e** file: true if the file exists
  - **-f** file: true if the file is a regular file.
  - **-s** file: true if the file has nonzero size

if control structure

- **Single-line Syntax**

```
if  TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi
```

- **Multi-line Syntax**

```
if  TEST-COMMANDS
then
    CONSEQUENT-COMMANDS
fi
```

Testing in interactive shell

- Write a script that reads from standard input a string, and check if it’s the same as your secret password “secret”; if yes, print out “welcome!”; print out “Go away” if not.

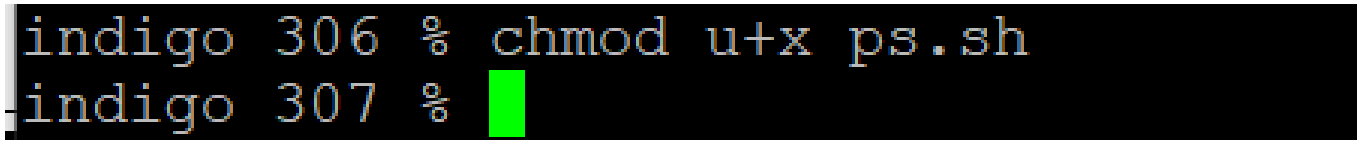
- test it out in interactive shell:

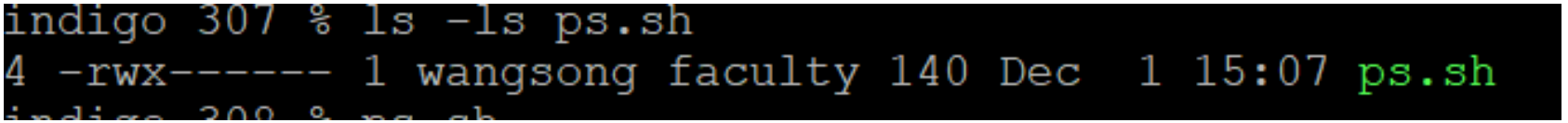
```
[indigo 301] $ read string
Secret

[indigo 301] $ if [ $string == "secret" ]; then echo
   "Welcome"; else echo "Go away"; fi

[indigo 301] $ Welcome
```

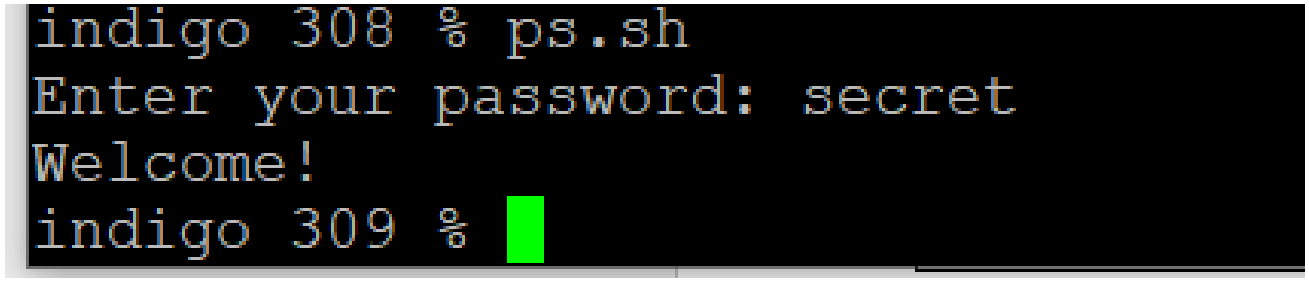
Bash Script

```
[indigo 301]$ vim ps.sh
#!/bin/bash
echo -n "Enter your password: "
read password
if [ $password == "secret" ]
then
    echo "Welcome!"
else
    echo "Go away!"
fi
```

```
indigo 304 % vim ps.sh
indigo 305 % ls -ls ps.sh
4 -rw------- 1 wangsong faculty 140 Dec  1 15:07 ps.sh
indigo 306 %
```


indigo 306 % chmod u+x ps.sh
indigo 307 %


indigo 307 % ls -ls ps.sh
4 -rwx------ 1 wangsong faculty 140 Dec 1 15:07 ps.sh


indigo 308 % ps.sh
Enter your password: secret
Welcome!
indigo 309 %

if ... then ... elif ... then ... else

```
if [[ "$op" == "+" ]]
then
        result=$(($x +
  $y))
        echo $x $op $y =
  $result
elif [[ "$op" == "-" ]]
  then
        result=$(($x - $y))
        echo $x $op $y =
  $result
```

```
elif [[ "$op" == "*" ]]
  then
        result=$(($x * $y))
        echo $x \* $y = $result
  elif [[ "$op" == "/" ]]
  then
        result=$(($x / $y))
        echo $x $op $y =
   $result
  else
        echo "Unknow operator
   $op"
fi
```

if... statements can be nested

```
#!/bin/bash
 # This script will test if we're in a leap year or not.
  year=`date +%Y` # shows the year only
  if [ $[$year % 400] -eq 0 ]; then
     echo "This is a leap year. February has 29 days."
 elif [ $[$year % 4] -eq 0 ]; then
          if [ $[$year % 100] -ne 0 ];
          then echo "This is a leap year, February has 29
 days."
          else echo "This is not a leap year. February has 28
 days."
          fi
 else echo "This is not a leap year. February has 28 days."
fi
```

Loop structure: while loop

- **Multi-line Syntax:**

```
while condition
do
   commands
done
```

- **Single-line Syntax** (useful in interactive mode)

while *condition;* do *commands;* done

  - Note: condition and commands terminated with ;

while loop

```
declare –i i=1   # an integer variable I
while [ $i -le 10 ]
 do
        echo "loop $i"
        i=i+1    # can use this since i is integer
done
```

If i is not declared as integer …

```
i=$(($i+1))
i=$[$i+1]
```

```
#!/bin/bash
echo -n "Enter your password: "
read password
if [ $password == "secret" ]
then
    echo "Welcome!"
else
    echo "Go away!"
fi
```

- **How to modify this to allow user to try until the password matches?**

```
#!/bin/bash
while test $password != "secret"    #as long as the password is not
  same as “secret”
do
  echo -n "Enter your password: "
  read password
done
echo "Welcome!”
```

- **What if we give the user at most 3 tries?**
  1. use a variable to keep track of the number of tries …
  2. modify condition …

until loop

- Tests for a condition and keeps looping as long as that condition is *false* (opposite of *while loop*).

until *condition*

do

command(s)...

done

- e.g.:

```
$until [ $passwd -eq "secret" ] ; do echo -n "Try again: "; read
  passwd; done
```

For loops

- **For loop**: iterates over a list of objects, executing loop body for each individual object in the list

```
for variable in a_list_of_objects
  do
        # do something on $variable
        commands ..
  done
```

```
e.g., for filename in lab1.cpp lab2.cpp lab3.cpp
        do
  indent $filename
        done
```

Using for loop

- Use for loop to print out 2’s power
  - Command seq: print out a sequence of number

```
#!/bin/bash
# print out 2’s powers
for a in `seq 1 10`
   do
      echo 2^$a=$((2**a))
  done
```

Note: ** is the exponent operator

case construct: branching

- **case** construct is analogus to *switch* in C/C++.

```
case "$variable" in
shellpattern1 )
command...
;;
shellpattern2)
  command ...
  ;;
shell pattern n)
command...
;;
esac
```

- Quoting variables is not mandatory
- Each pattern can contain **shell wildcard** (*,?,[a-z]), ends with a **)**
- Each condition block ends with **;;**
- If a condition tests *true*, then associated commands execute and the **case** block terminates.
- entire **case** block ends with

Calculator using case block

```
case  "$op" in
"+" )        result=$(($x + $y))
                echo $x $op $y = $result;;
"-" )        result=$(($x - $y))
                echo $x $op $y = $result;;
"*" )        result=$(($x * $y))
                echo $x \* $y = $result;;
"/" )        result=$(($x / $y))
                echo $x $op $y = $result;;
* )       echo Unknow operator $op;;
esac
```