Table of Contents

Shell Script

A text file containing shell commands and/or shell language statements is a shell script.

- A comment in a shell script is indicated by beginning the line with a "#".
- Anything you type on the command line can be put into a text file and made into a shell script and vice versa.
- In a shell script, a shebang interpreter directive tells Linux which command interpreter to use. Such as:
  - #!/bin/bash
  - #!/bin/tcsh
  - #!/usr/bin/perl
- Alternatively, you can use /usr/bin/env to search the user's PATH for the correct interpreter:
  - #!/usr/bin/env bash

Example:

```
$ cat >whoson
#!/bin/bash #shows the date and who is on
date
echo "who is on?"
who
CTRL-D
```

This will defer to PATH to find the location of bash. This can be useful when you aren't certain of the interpreter's location or if the user prefers to use a different binary than the installed one.

When debugging, it's useful to set the set -x command at the beginning of your script (after shebang). This makes all commands print to stdout as they're executed.

->Lecture:

At this time the professor is saying the file extension of genwords.py and whoson.bash are meaningless, proceeds to remove the extensions of both and run them as: ./genwords 5 and ./whoson
Both programs run as normal which asks the question: How does Linux know one is in python and the other in bash?
genwords has the shebang line #!/usr/bin/python3

1. The shebang has to be on the first line
2. The OS sees it's an executable, a text file, then looks at the first line to see how to run this.
3. Takes the interpreter then adds to it the name of your script:
   /usr/bin/python3 /home/ssilvestro/genwords

Shell Variables

- Similar to variables in other languages since they represent other values. The names must begin with a letter or underscore, but can contain letters, numbers, and underscores.
  - Valid:
    line, name1, name2, first_name
  - Invalid:
    first-name, last.name, 1char
- The value of a variable is referenced by preceding it with a $
- There cannot be any spaces around the '=' when initializing a variable. e.g VAR=value is valid, but VAR = value doesn't work.
- The shell is indifferent to the types of variables as they're all stored as strings, it moreso depends on what context you're using it in. This doesn't mean we can just add numbers to strings like in:

```
$ x="hello"
$ expr $x + 1
expr: non-numeric argument
```

- But there is no syntactic difference between these;

```
my_msg="Hello"
my_shortmsg=hi
my_num=1
my_pi=3.142
other_pi="3.142"
mix=abc123
```

- Special characters must still be properly escaped

The parameters to a shell script are referenced by a '$' followed by a positional number. The name of the command is $0. $1 is the first parameter, $2 is the second parameter, and so on. The number of parameters is $#, $@ represents the list version of the parameters. Variable names are case sensitive.

Example:

```
$ cat >echoName
echo "Number of parameters is $#"
echo "Full name is $1 $2"
name=$1
echo "First name is $name"
$ chmod u+x echoName
$ ./echoName joe mama
Number of parameters is 2
Full Name is joe mama
First name is joe
```

Setting Variable Values

Shell scripts provide assignment statements which can give shell variables values during the execution of the shell.

bash/dash:

```
targetVariable=value
```

Example:

```
#assigning some variables via bash script
$ cat > assignVar
#!/bin/bash
first=bruce
last=wayne
echo $first $last
$ chmod u+x assignVar
$ ./assignVar
bruce wayne
$ echo $first $last
```

The result after the last line is literally nothing because `first="bruce"` and `last="wayne"` are only valid during the execution of the script.

After the shell script ‘assignVar’ executes, the values of first and last are only set within the scope of the script itself. Once the script is finished executing, the variables aren’t accessible in the shell where the script was run.

You would need to source the script instead of running it as a subprocess:

```
$ source assignVar
bruce wayne
$ echo $first
bruce
$ echo $last
wayne
```

($ **.** **assignVar** is another valid option)

Assigning Math Expressions in bash

4 Forms:

1. let targetVariable=expression
2. let targetVariable=$((expression))
3. targetVariable=$((expression))
4. ((targetVariable=expression))

Note: The double parenthesis allow spacing around operators within the double parenthesis.

The **let** command causes the variables to undergo expansion.

Example:

```
# shell script for simple math taking 2 parameters
$ vi simpMath
================================================
#!/bin/bash
let sum=$1+$2
let product=$(($1*$2))
diff=$(($1-$2))
((quotient = $1/$2))
echo “sum is $sum, product is $product”
echo “diff is $diff, quotient is $quotient”
================================================
$ chmod u+x simpMath
$ ./simpMath 3 2
sum is 5, product is 6
diff is 1, quotient is 1
```

Floating Point

Note that floating point values don’t work in either bash or tcsh. If necessary, pipe the expression into a utility such as **bc** (basic calculator).

Example:

```
$ echo $((3/2))
1
$ echo “3/2” | bc -l
1.50000000000000000000
```

```
#What's the result of this script?
$ cat > massign
#!/bin/bash
one=two
two=one
let $two=2
let one=2
echo “one is $one”
echo “two is $two”
$ ./massign
one is 2
two is one
```

Some Important Variables

There are several important environment variables (global) for your login session:

- PATH - List of directories to search to find commands. This is initially established by a Systems Administrator having root privileges.
- HOME - This is where your home directory exists. The shell uses this for the value of '~'
- PWD - This is the current directory.

```
$ echo $PATH #shows the contents of PATH
/home/kate/bin: /home/kate/.local/bin:..etc
```

Showing Env Variables

You can see all of your current environment variables by using the printenv command.

Example:

```
$ printenv | more
USER=ssilvestro
LOGNAME=ssilvestro
HOME=/home/ssilvestro
PATH=/usr/local/sbin:...etc
--more--
```

Linux Book

When the Linux kernel executes a program, it provides it with an environment (a list of strings in the form of name-value pairs/an array of strings). This is a key part that is used to pass configuration information, settings, and other data.
This list is called the command execution environment (or just environment). The name-value pairs are in the form *name=value*.

When bash is invoked, it scans the environment given by the kernel/parent process and creates shell parameters for each name-value pair, which are referred to as environment variables.

Example:
If the environment contains HOME=/home/user, bash creates a variable named HOME and assigns it the value /home/user.

Environment variables are often called global variables as well because they're accessible throughout the shell session and by any child processes spawned by the shell.
They're also called exported variables since for a shell variable to become an environment variable (and inherited by child processes), it has to be exported using the export command.

Example:

```
A_VAR="Hi"
export A_VAR
#to remove use: unset A_VAR
```

Parent-Child -> When a process/shell spawns a child process, the child inherits a copy of the parent's environment (all environment variables, changes in the child don't affect the parent's environment, and changes in the parent after the child is spawned don't affect the child.)

Parent:

```
PARENT_VAR="parent_value"
export PARENT_VAR
echo "Parent: PARENT_VAR=$PARENT_VAR"
bash #spawns child shell
```

Child:

```
echo "Child: PARENT_VAR=$PARENT_VAR"
CHILD_VAR="child_value"
export CHILD_VAR
echo "Child: CHILD_VAR=$CHILD_VAR"
```

Back to Parent:

```
echo "Parent: CHILD_VAR=$CHILD_VAR"
#will be empty: parent cannot see child's changes
```

If two processes are unrelated (e.g 2 separate terminal sessions), they don't share environment variables even if those variables share the same name.
Variables are local, meaning they're specific to a process.

Example:

```
$ cat extest1
cheese=american
echo "$0 1: $cheese"
./subtest
echo "$0 2: $cheese"
$ cat subtest
echo "$0 1: $cheese"
cheese=swiss
echo "$0 2: $cheese"
$ ./extest1
./extest1 1: american
./subtest 1: #displays nothing b/c not declared or in the environment.
./subtest 2: swiss
./extest1 2: american
```

The subtest script never receives the value of cheese from extest1, and extest1 never loses the value: cheese is a shell variable, not an environment variable.

```
$ cat extest2
export cheese=american
echo "$0 1: $cheese"
./subtest
echo "$0 2: $cheese"
$ ./extest2
./extest2 1: american
./subtest 1: american
./subtest 2: swiss
./extest2 2: american
```

You can export a variable without/before assigning a value to it. You do not need to export an already-exported variable after you change its value.

'env' runs a program as a child of the current shell, allowing you to modify the environment the current shell exports to the newly created process.

env [options] [-] [name=value] . . .[command-line]

Setting PATH

You can set your path to include your bin (make sure your ~/bin exists, if not mkdir it.)

bash:

- To have it set each time you start bash, change your ~/.bashrc
- To change PATH to search ~/bin after the other directories, use: PATH=$PATH:$HOME/bin

```
#hidden files:
$ ls -al ~/.*rc
-rwx--x--x 1 ssilvestro faculty 441 Jul 14 2015 /home/ssilvestro
/.cshrc
-rw-r--r-- 1 ssilvestro faculty 43 May 31 2016 /home/ssilvestro /.dmrc
-rw------- 1 ssilvestro faculty 97 Oct 17 2016 /home/ssilvestro
/.vimrc

#create ~/bin
$ mkdir ~/bin
```

Scope of Shell Variables

By default, shell variables are only known to the script and do not propagate to any script invoked from that script.

Scope is covered above, this will include (repeated) examples from the slides.

Example:

```
$ cat >extest1 #creating the script extest1
echo “$0 0: $cheese”
cheese=american
echo “$0 1: $cheese”
./subtest1
echo “$0 2: $cheese”
$ cat >subtest1 #another script created
echo “$0 0: $cheese”
cheese=swiss
echo “$0 1: $cheese”
$ ./extest1
./extest1 0:
./extest1 1: american
./subtest1 0:
./subtest1 1: swiss
./extest1 2: american
```

Note that cheese didn’t propagate to subtest1, and the change to cheese in subtest1 isn’t known in extest1

Exporting Variables

Explained above, you can propagate shell variables to executed shell scripts by exporting the variables. Exporting a variable makes it an environment variable, which is a global variable for shells. The variable will be available to any sub-shells.

```
$ cp extest1 extest2 #copies extest1 to extest2
$ vi extest2
 ⌜
  echo “$0 0: $cheese”
  cheese=american
  export cheese
  echo “$0 1: $cheese”
  ./subtest1
  echo “$0 2: $cheese”
                        ⌟
$ ./extest2
./extest2 0:
./extest2 1: american
./subtest1 0:
./subtest1 1: swiss
./extest2 2: american
#run extest1 again ?
$ ./extest1
./extest1 0:
./extest1 1: american
./subtest1 0:
./subtest1 1: swiss
./extest1 2: american

#create another version of extest2
$ cp extest2 extest3
$ vi extest3
 ⌜
  echo “$0 0: $cheese”
  cheese=american
  export cheese
  echo “$0 1: $cheese”
  source ./subtest1
  echo “$0 2: $cheese”
                        ⌟
$ ./extest3
./extest3 0:          #pre-source: blank
./extest3 1: american #pre-source: american
./extest3 0: american #pre-source: american
./extest3 1: swiss    #pre-source: swiss
./extest3 2: swiss    #pre-source: american
```

Getting an invoked shell script to set variables in the current shell can be done (depending on the shell) by using either "." or source. This causes the invoked shell script to execute as part of the current process instead of a new process. Changes to variables in the invoked script will affect the current shell.

Exit Status

In addition to commands writing output to stdout, commands have an exit status:

0	success
non-zero	command failed

This can be tested to see whether something that was invoked actually worked. Your shell commands can return a failure by executing: `exit n`
To show a failure, the value of n will typically be 1 for most scripts.
You can get the last exit status by accessing the special value `$?`

Example:

```
$ cat xxx
cat: xxx: No such file or directory
$ echo $?
1
```

The exit status is also known as a condition code or return code ($status under tcsh).

- A nonzero exit status is interpreted as false and means the command failed.
- A zero is interpreted as true and indicates success.

Job Sequences

Within shell scripts we can specify job sequences which are an easy way to link two commands based on the execution status.

- cmd1 args && cmd2 args #logical and
  - #cmd2 only executes if cmd1 returns 0
- cmd1 args || cmd2 args #logical or
  - #cmd2 only executes if cmd1 returns non-zero
- cmd1 args; cmd2 args
  - #cmd2 executes regardless of cmd1's return value

The approach of using && and || is like short circuiting in many languages.

- If it's necessary for cmd2 to be multiple commands, surround them in parenthesis.

Example:

```
$ mkdir ~/cs3423/Jobs #create a backup directory, if it fails show an error
$ mkdir “Backup`date+%Y%m%d`”||\ #\ is used to continue the command
? echo “creation of Backup directory failed”
#Question mark is a shell prompt for continuation
#The name of the backup directory will be “Backup” followed by the current date

$ mkdir “Backup`date+%Y%m%d`”||\
? echo “creation of Backup directory failed”
mkdir: cannot create directory Backup20170713: File exists
creation of Backup directory failed

$ mkdir "Backup`date +%Y%m%d`" && cp -r ~/cs2123/* "Backup`date +%Y%m%d`"
mkdir: cannot create directory Backup20170713: File exists
#creates a Backup directory, if it’s successful, copy the contents
#of a folder to it. cp -r will recursively create/copy any subdirectories

#removes the backup directory
$ mdir Backup20170713

#make the backup copy only if creating the directory is ok
$ mkdir "Backup`date +%Y%m%d`" && cp -r ~/cs2123/* "Backup`date +%Y%m%d`"
# Create the backup directory and copy the files. If either of those
# failed, show an error message.
$ ( mkdir "Backup`date +%Y%m%d`" && \
? cp -r ~/cs2123/* "Backup`date +%Y%m%d`" ) || echo "backup failed";
mkdir: cannot create directory Backup20170713: File exists
backup failed
# Remove the backup directory (whatever its name is) and try that again
```

Reading input from stdin (works in bash and dash)

The read built-in command reads from stdin. Syntax:

`read variable` -> reads the input into the specified variable until linefeed

`read var1 var2` -> reads the first word into var1, the second word into var2, and so on. read looks for a linefeed if there are less words in the input than read variables, it doesn’t populate the other variables. If there are more words than variables. The remaining words are placed in the last variable.

`read -p “prompt message” variableList` -> This shows the prompt message and reads the words into the variables represented by variableList. Those variables and input work the same as without the -p and prompt.

`read:` accepts user input (reads information from stdin and stores it as a variable). stdin/stdout are connected to your terminal. When a program reads from a terminal, it receives keystrokes from your keyboard; when it writes to a terminal, characters are displayed on your monitor.

- using read, scripts can accept input from the user and store that input in variables.
- input > refers to any information that your program receives/reads. Input can come from several different places such as: command-line arguments, environment variables, files, anything else a File Descriptor can point to (pipes, terminals, sockets).
- output > refers to any information that your program produces/writes. It can also go to: files, anything else a File Descriptor can point to, /command-line arguments/ or /environment variables passed/ to some other program.

Example:

```
$ cat read1
echo -n “Go ahead: “ #-n suppresses newline letting you enter text on the same line
read firstline
echo “You entered: $firstline”
$ ./read1
Go ahead: This is a line.
You entered: This is a line.
```

```
$ cat read1a
read -p “Go ahead: “ firstline #-p prompts user for input on same line
echo “You entered: $firstline”
$ ./read1a
Go ahead: My line.
You entered: My line.
```

```
$ cat read1_no_quote
read -p “Go ahead: “ firstline
echo You entered: $firstline
$ ./read1_no_quote
Go ahead: * #the shell expands the asterisk into a list of files in the working directory
You entered: read1 read1_no_quote script.1
$ ls
read1       read1_no_quote          script.1
```

```
$ ./read1 #with quotation marks it would be:
Go ahead: *
You entered: *
```

When you don’t specify a variable to receive read’s input, bash puts the input into the variable named REPLY.

```
$ cat read1b
read -p “Go ahead: “
echo “You entered: $REPLY”
```

```
$ cat read2
read -p “Enter a command: “ cmd
$cmd #reads user response and assigns it to the variable cmd
echo “Thanks”      #it then attempts to execute the command line that results
                   #from the expansion of the cmd variable
$ ./read2
Enter a command: echo Please display this message.
Please display this message.
Thanks.
$ ./read2
Enter a command: who
max   pts/4        2013-06-17 07:50 (:0.0)
sam   pts/12       2013-06-17 11:54 (guava)
Thanks
$ ./read2
Enter a command: xxx
./read2: line 2: xxx: command not found
Thanks
```

If cmd doesn’t expand into a valid command line, the shell gives an error.

```
$ cat read3
read -p “Enter something: “ word1 word2 word3
echo “Word 1 is: $word1”
echo “Word 2 is: $word2”
echo “Word 3 is: $word3”
$ ./read3
Enter something: this is something
Word 1 is: this
Word 2 is: is
Word 3 is: something
$ ./read3
Enter something: this is something else, really.
Word 1 is: this
Word 2 is: is
Word 3 is: something else, really.
```

When more words than variables are entered, read assigns all leftover words to the last variable.

Example:

```
$ vi simpRead
```

```
read -p “Enter some words.” line
echo “line = $line”
read -p “Enter your first and last name:” first last
echo “first = $first, last = $last”
read -p “Enter first name only, but read asked for 2:” first2 last2
echo “first2 = $first2, last2 = $last2”
read -p “Enter more than 2 words:” one other
echo “one = $one, other = $other”
```

```
Enter some words: one two three
line = one two three
Enter your first and last name: joe mama
first = joe, last = mama
Enter first name only, but read asked for 2:joe
first2 = joe, last2 =
Enter more than 2 words: joe and bazooka joe
one = joe, other = and bazooka joe
```

Read Options

`-a var` -> (array) Assigns each word of input to an element of array var.

`-d delim` -> (delimiter) Uses delim to terminate the input instead of newline.

`-e` -> (Readline) If input is coming from a keyboard, uses Readline Library to get input. Provides editing capabilities. (e.g. while typing “foo” if you stop and hit CTRL-A, CTRL-K, then CTRL-Y, those won’t show on the terminal. Without -e it shows as “foo^A^K^Y”.

`-n num` -> (number of characters) Reads num characters and returns.

`-p prompt` -> (prompt) Displays prompt on standard error without terminating newline before reading input. Displays prompt only when input comes from the keyboard.

`-s` -> (silent) Doesn’t echo characters.

`-un` -> (file descriptor) Uses the int n as teh file descriptor that read takes its input from. (‘read -u4 arg1 arg2’ is equivalent to ‘read arg1 arg2 <&4’.

Reading Text from stdin until EOF (in bash)

read returns a success exit status if it reads a line of input.

```
while command; do
      do something with it
done
```

Note that you can tell the while loop to use a different file for stdin by specifying < and a fileName after the done.

```
while read line; do
      do something with it
done < fileName
```

Example:

```
#copying the file mySentences from /usr/local/courses/ssilvestro/cs3423/shell to your directory
$ vi rloop
 ⌜
  #!/bin/bash
  count=0
  while read line; do
   echo $line
   let count+=1
  done
  echo “$count lines”
                          ⌟
$ chmod +x rloop #Invoke rloop using mySentences for stdi
$ ./rloop <mySentences
Scooby Doo shook with fear when he saw the ghost. Shaggy ran and hid
in the Mystery van. Freddie tried to act brave to impress Daphne, but
she was lovinly watching Scooby Doo. Velma was studying the foot
prints in the mud.
Velma said, "this is the same mud that we saw on the stairs at UTSA!"
6 lines
```

```
$ cat names
Alice Jones
Robert Smith
Alice Paulson
Henry Rollins
$ while read first rest
do
echo $rest, $first
done < names
Jones, Alice
Smith, Robert
Paulson, Alice
Rollins, Henry
```

Flow Control Statements in bash

The while, if, case, and for statements are used for flow control. The various shell dialects have different syntax for flow control.

while

```
while condCommand; do
      doSomething
done
```

if

```
if condCommand; then
      doSomething
fi
```

if-else

```
if condCommand; then
      doSomething
else
      doSomethingWhenNotTrue
fi
```

if-elif-else

```
if condCommand; then
      doSomething
elif condCommand2; then
      doSomething2
. . .
else
      doSomethingN
fi
```

Example:

```
#if statement
$ cat 1f1
「
    read -p “word 1: “ word1
    read -p “word 2: “ word2
     if test “$word1” = “$word2”
      then
      echo “Match”
     f1
  echo “End of program.”
                              」
$ ./1f1
word 1: peach
word 2: peach
Match
End of program.
```

```
#while loop
┌
  x=1;
  while [ $x -le 5 ] #-le is <=
  do
    echo “Welcome $x”
    x=$(( $x + 1 ))
  done
                                ┘
$ ./whiletest
Welcome 1
Welcome 2
Welcome 3
Welcome 4
Welcome 5
```

```
#if-else
┌
  echo -n “Enter a number: “
  read VAR
    if [[ $VAR -gt 10 ]]
      echo “The variable is greater than 10.”
    else
      echo “The variable is equal or less than 10.”
    fi
                                                        ┘
$ ./elsetest
Enter a number: 42
The variable is greater than 10.
```

```
#if-elif-else
┌
  echo -n “Please enter a whole number: “
  read VAR
  echo Your number is $VAR
    if [ $VAR -gt 100 ]
      then
      echo “It’s greater than 100”
    elif [ $VAR -lt 100 ]
      then
      echo “It’s less than 100”
    else
      echo “It’s exactly 100”
    fi
echo Bye!
                                              ┘
$ ./eliftest
Please enter a whole number: 49
Your number is 49
It’s less than 100
Bye!
```

```
#checking arguments
$ cat chkargs
「
   if test $# -eq 0
      then
      echo “You must supply at least one argument.”
      exit 1
   fi
   echo “Program running.”
                                                    」

#script that sums integers from 1 to n
$ vi simpWhile
「
    if [$3 -lt 1]; then
    echo “usage: simpWhile number”
    echo “    where number is a number”
    exit 1
    fi
  sum=0
  index=$1
    while [ $index -gt 0 ]; do
    let sum=sum+index
    let index=index-1
    done
  echo $sum
                                          」
$ ./simpWhile 7
$ chmod +x simpWhile
$ ./simpWhile 4
10
```

Conditional Commands

while and if statements use conditions, which can be commands. There’s a special command, test, which can be used to test conditions:

```
if test $var -gt 100; then
```

The test command returns an exit status of 0 to represent that the condition is true if the variable is greater than 100.
You can also add single brackets around the command.

```
if [ $var -gt 100 ]; then
```

bash-only scripts can use double brackets which are handled by bash instead of invoking the test command. Problems associated with using “>” or “<” for tests are avoided when using the double brackets.

```
if [[ $var -gt 100 ]]; then
```

Numeric Comparisons of Two Operands

```
op1 -gt op2      Greater Than
op1 -lt op2      Less Than
op1 -eq op2      Equal To
op1 -ne op2      Not Equal To
op1 -le op2      Less Than or Equal To
op1 -ge op2      Greater Than or Equal To
```

String Comparisons of Two Operands

```
op1 > op2
op1 < op2
op1 == op2
op1 != op2
op1 <= op2
op1 >= op2
```

File Checks

```
-d filename      Exists and is a directory
-e filename      Exists
-f filename      Exists and is a file, not a directory
-r filename      Exists and is readable
-s filename      Exists and has a size > 0 bytes
-w filename      Exists and is writable
-x filename      Exists and is executable
```

Warning!

This doesn't do what you expect:

```
if [ $varA > $varB ]; then

if test $varA > $varB; then
```

With test and the single bracket expressions, the '>' is interpreted as redirection of output to the file named as the value of $varB. Step outside and God smites you.

The problem can be avoided with one of these approaches:

- if [$varA \> $varB]; then

- if [[$varA > $varB]]; then

case statement

The case statement is very powerful. It has multiple patterns for matching values. Syntax:

```
case “variableRef” in
      pattern1)
        doSomething1
       ;;
      pattern2)
        doSomething2
       ;;
      . . .
      *)
        doSomethingDefault
       ;;
      esac
```

Each pattern can include:

simpleValue -> This is just a string to match. (e.g Jan, Feb)

* -> This matches anything. By itself, this is used for the default case.

alt1|alt2 -> This specifies alternatives (e.g dog|cat)

[list] -> This specifies a list of possible values. (e.g [Jj]an,[Ff]eb). You can also use hyphen for a range of values.

? -> This matches any single character.

Example:

```
read -p “Enter A, B, or C: “ letter
case “$letter” in
A)
   echo “You entered A”
   ;;
B)
   echo “You entered B”
   ;;
C)
   echo “You entered C”
   ;;
*)
   echo “You did not enter A, B, or C”
   ;;
esac
$ ./case1
Enter A, B, or C: B
You entered B
```

Example:

```
$ vi exCase
┌
  #!/bin/bash
  read -p "Enter Month (MMM) DayOfMonth and Year:" month day year
  # convert the alpha month to a numeric
  case "$month" in
  [Jj]an) mon=1;;
  [Ff]eb) mon=2;;
  [Mm]ar) mon=3;;
  [Aa]pr) mon=4;;
  [Mm]ay) mon=5;;
  [Jj]un) mon=6;;
  [Jj]ul) mon=7;;
  [Aa]ug) mon=8;;
  [Ss]ep) mon=9;;
  [Oo]ct) mon=10;;
  [Nn]ov) mon=11;;
  [Dd]ec) mon=12;;
  *) echo "Bad month value = '$month'"
  exit 1;;
  esac
  echo "date=$mon/$day/$year"
                                                    ┘
```

☆ You can use | to separate alternative choices:

Example:

```
a|A)
   echo “You entered A”
   ;;
b|B)
   echo “You entered B”
c|C)
   echo “You entered C”
   ;;
. . .
```

☆ Special characters in echo:

```
\a    -> alert
\b    -> backspace
\c    -> suppress trailing newline
\f    -> formfeed
\n    -> newline
\r    -> return
\t    -> horizontal tab
\v    -> vertical tab
\\    -> backslash
\nnn  -> character with ASCII octal code nnn; if nnn not valid, displays string literally
```

for statement

The for statement iterates over a list. Syntax options:

```
for var in "$@"; do
   doSomething
done
```

```
for var in valueList, do
   doSomething
done
```

```
for var in $(command argos); do
   doSomething
done
```

Another C-style for-loop utilizing a special syntax of the arithmetic evaluation operator:

```
for ((x=0;x<10;x++)); do doSomething, done
```

Example:

```
$ cat dirfiles #lists the names of the dir. files in the working
directory and using test to determine which are dir. files
for i in *
do
   if [ -d "$i" ]
     then
     echo "$i"
   fi
done
```

Example:

```
$ vi simpFor
echo "1. show the list of files"
for file in "$@"; do
   echo "$file"
done

echo "2. show the list of fruit"
   for fruit in orange apple grape; do
   echo "$fruit"
done

echo "3. show the list of files from a command"
for file in $(ls -a *); do
   echo "$file"
done
```

Break and continue

The break and continue statements can be used within for, while, and until statements.
break exits the loop. continue continues with the next iteration skipping
the remaining statements within the current iteration.

Example:

```
#!/bin/bash
#loop until one of the arguments is not a valid file
for file in "$@"; do
 if [ ! -r "$file" ]; then
 break
 fi
cat $file
done
```

---

prompted input loop until EOF with multiple prompts

Suppose you want to prompt a user for input or terminate with CTRL-D in a loop. If there are multiple prompts (like a menu), you may need to do a while that itself doesn't have a terminating condition. When the EOF is encountered, break the loop.

Example:

```
$ vi showMenu.bash
```

```
#!/bin/bash
  go=0
   while [ $go ]; do
   echo "Enter your choice or CTRL-D"
   echo "A - I want do get an A"
   echo "B - I want to get a B"
   echo "F - I give up"
 if ! read ans; then
 #got EOF
 break
 fi
case "$ans" in
A) echo "you got an A"
break
;;
B) echo "you can do better than a B"
;;
F) echo "keep trying"
;;
*) echo "we think you can type better than that"
esac
done
```

Reading several variables from multiple lines in a file

Sometimes it is necessary to read several variables from multiple lines of text in a single file. We saw earlier how we can read multiple lines of text from a file using a while loop. This is different since we need to do multiple reads, but have each populate different variables.

Example:

```
$ vi first.bash
#!/bin/bash
 read -p "Enter the filename:" filename
#we want to read multiple lines from that file, populating multiple
#variables per line
  if [ ! -r $filename ]; then
   echo "could not read that file"
   exit 1
  fi
bash second.bash < $filename
```

```
$ vi second.bash
  #!/bin/bash
read studentId studentMajor
read studentName
echo "Student: $studentId $studentName"
echo "Major: $studentMajor"
echo "Courses:"
while read courseNr courseGrade; do
     echo “$courseNr $courseGrade”
done
```

```
$ bash first.bash
Enter the filename:111
Student: 111 Sally Mander
Major: BIO
Courses:
BIO3233 A
BIO3343 B
BIO1111 A
MAT1214 C
```

Functions

As with many languages, most shells offer the ability to create functions. In Bash, the following syntax creates a function which can be used later:

```
my_function ()
{
# function body
echo "hello world"
}
```

To invoke the function, use the name without the parentheses
If you want the function body to execute in its own subshell (i.e., not have it affect the current shell's environment), replace the curly brackets with parentheses.

```
my_function2 ()
(
# function body
echo "hello world"
)
```

As with any command, you may use redirection with function invocations.

Example:

```
$ vi functions.bash
```

```
#!/bin/bash
cd_courses ()
{
 cd ~/courses
   echo "I'm executing this from within `pwd` "
}
cd_courses
echo "Now I'm executing this from within `pwd` "
```

```
$ cd ~
$ bash functions.bash
I'm executing this from within /home/ssilvestro/courses
Now I'm executing this from within /home/ssilvestro/courses
$ vi functionsSubshell.bash
```

```
#!/bin/bash
 cd_courses ()
 (
  cd ~/courses
  echo "I'm executing this from within `pwd` "
 )
cd_courses
echo "Now I'm executing this from within `pwd` "
```

```
$ cd ~
$ bash functionsSubshell.bash
I'm executing this from within /home/ssilvestro/courses
Now I'm executing this from within /home/ssilvestro
```