COMP 175

System Administration and Security

Bash Continued

A web server in a shell script

- A minimalist web server implemented in bash
- Step 1: A typical beginning

```
#!/bin/bash
base=/var/www
```

A web server in a shell script

- Step 2: The inetd feeds the script with data received from the remote host, the first row being the standard HTTP request, followed by zero or more header lines.

```
read request
while /bin/true; do
  read header
  [ "$header" == $'\r' ] && break;
done
```

- Note: /bin/true -do nothing, successfully

A web server in a shell script

- Step 3: extract the URL from the request string and locate the document on the local file system

```
url="${request#GET }"
url="${url% HTTP/*}"
filename="$base$url"
```

- Step 4: send the file (if it exists) with a leading standard header:

A web server in a shell script

```
if [ -f "$filename" ]; then
  echo -e "HTTP/1.1 200 OK\r"
  echo -e "Content-Type: `/usr/bin/file -bi
  \"$filename\"`\r"
  echo -e "\r"
  cat "$filename"
  echo -e "\r"
else
  echo -e "HTTP/1.1 404 Not Found\r"
  echo -e "Content-Type: text/html\r"
  echo -e "\r"
  echo -e "404 Not Found\r"
  echo -e "Not Found
           The requested resource was not found\r"
  echo -e "\r"
fi
```

A web server in a shell script

- Add this to /etc/inetd.conf (on a server)

```
www stream tcp nowait nobody /usr/local/bin/webd webd
```

- where webd is the name you gave to the script
- Restart the internet daemon

```
/etc/init.d/inetd restart
```

- Make the directory /var/www,
- place some HTML files in it
- load a web browser and try the URL:

```
http://localhost/filename.html
```

The usual disclaimers and caveats apply.

Disclaimer

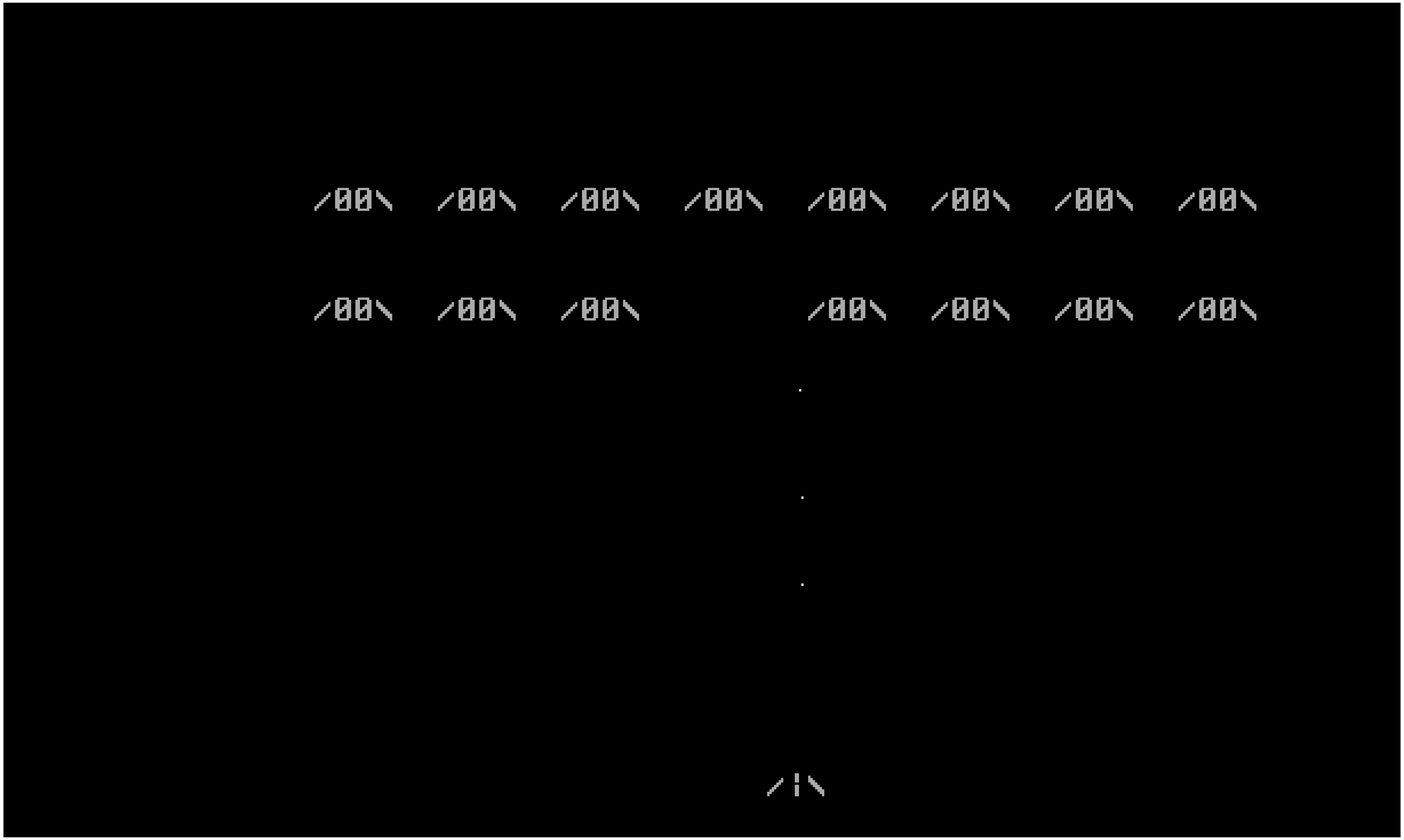
Batteries not included. Contents may settle during shipment. Use only as directed. No other warranty expressed or implied. Do not use while operating a motor vehicle or heavy equipment. Apply only to affected area. May be too intense for some viewers. For recreational use only. Do not disturb. All models over 18 years of age. If condition persists, consult your physician. No user-serviceable parts inside. Freshest if eaten before date on carton. Subject to change without notice. Times approximate. Simulated picture. As seen on TV. One size fits all. Action figures sold separately.

bash

- Scripts are good for repetitive tasks
- Shell code good for ad-hoc queries, tasks
- unique feature in bash is built in help system
- `help` - returns list of bash commands & options
- `help command` returns description of command

- Bash can even be used for games

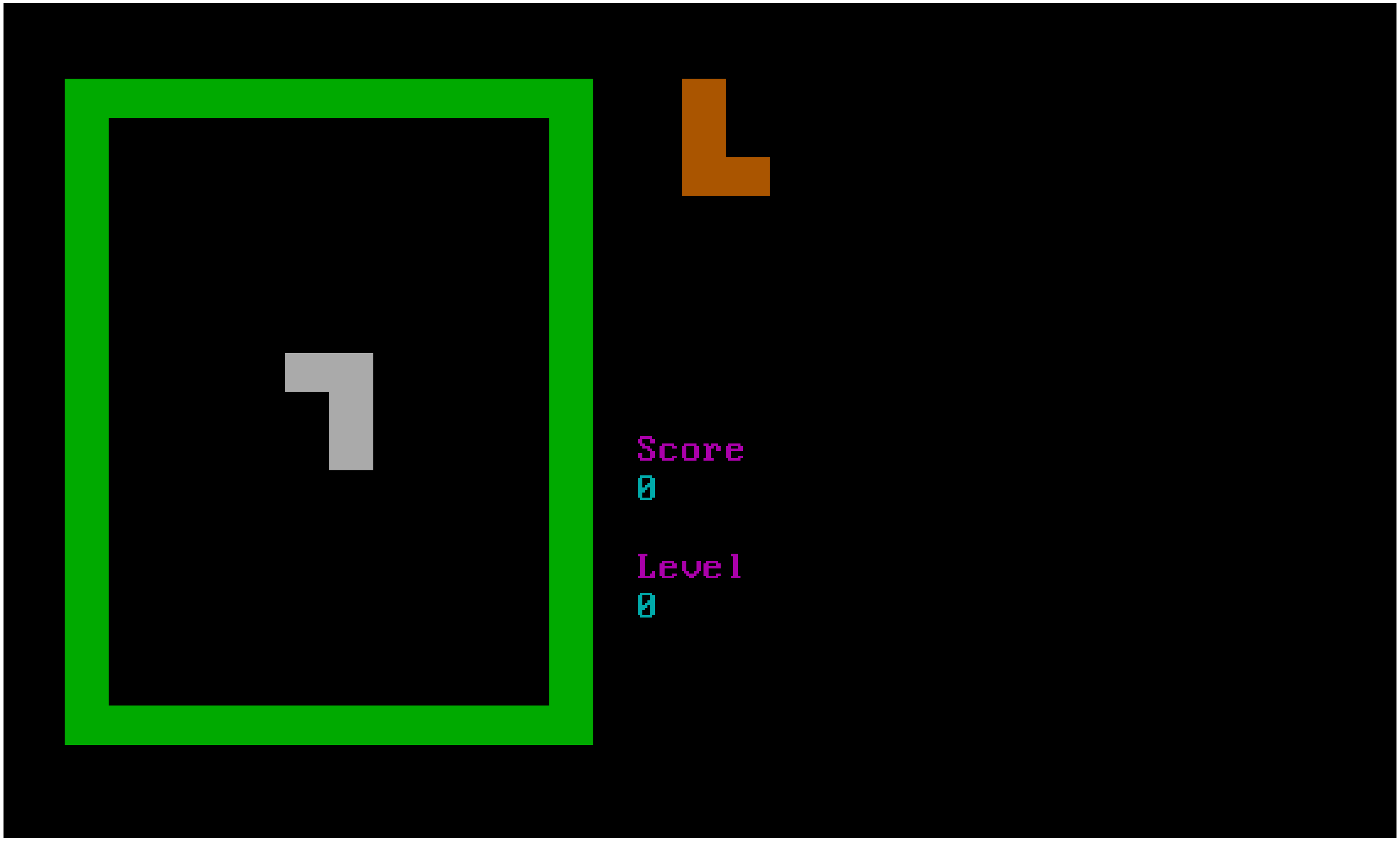
bash invaders



```
/00\  /00\  /00\  /00\  /00\  /00\  /00\  /00\


/00\  /00\  /00\        /00\  /00\  /00\  /00\

                        .


                        .


                        .




                      /¡\
```

(Google for source)

bash tetris



```
Score
0

Level
0
```

A CLOSER LOOK

WHEN I STARTED PROGRAMMING, WE DIDN'T HAVE ANY OF THESE SISSY "ICONS" AND "WINDOWS."
ALL WE HAD WERE ZEROS AND ONES -- AND SOMETIMES WE DIDN'T EVEN HAVE ONES.
I WROTE AN ENTIRE DATABASE PROGRAM USING ONLY ZEROS.
YOU HAD ZEROS? WE HAD TO USE THE LETTER "O."
S.Adams
© 1992 United Feature Syndicate, Inc.

Finding the Program

- #!/bin/bash
- echo ‘Hello, world’

- $ echo $PATH

  /usr/local/bin:/usr/bin:/bin .....
- $ which echo

  /usr/bin/echo

Bash Startup Files

1. /etc/profile – Executed automatically at login
2. Execute at login first file found in this list:
   a) ~/.bash_profile
   b) ~/.bash_login
   c) ~/.profile (same as in Korn shell)
3. ~/.bashrc – Executed by every shell at login
4. ~/.bash_logout – Executed at logout

Use bash for both programming and interactive use

executable scripts

- The first line in your script must be
  - hashbang and absolute path to the shell
  - no shebang? default shell

```
#!/bin/bash
```

- Convention has script filenames end with .sh
- Make the script executable

```
$chmod u+x mfile.sh


$ ls –lg filename.sh
-rwxr--r–- users 104 Oct 5 22:12 mfile.sh
```

Execution

- To execute the program

```
$ mfile.sh
bash: mfile.sh: command not found
```

- $PATH environment variable holds the location where all commands are stored

- Specify path of mfile.sh - absolute or relative

```
$./mfile.sh
```

Continuing Lines

```
echo This \
Is \
  A \
Very \
Long \
 Command Line
```

- This Is A Very Long Command Line

List of Commands

`cmd1; cmd2;` execute sequentially

`cmd1 &` execute asynchronously[ⓘ]

`cmd1 && cmd2` execute cmd2 if cmd1 has exit(0)

`cmd1 || cmd2` execute cmd2 only if cmd1 has non-zero exit status

[ⓘ] Background or non-blocking I/O - permits other processing to continue

Quoting

- Quoting allows you to distinguish between the literal value of symbol and symbols used as code
- A way to distinguish between the literal symbol and the symbol's use as a metacharacter or wild card characters
- To do this you must use of the following symbols:
  - Backslash (\)
  - Single quote (')
  - Double quote (")

Backslash

- A backslash is also called the escape character
- It preserves only the character immediately following it

- Example:
  - to create a file named "tools>", enter:
  - `% touch tools\>`

Single Quote

- A single quote is used to protect the literal meaning of metacharacters.
- It protects all characters within the single quotes
- The only character it cannot protect is itself
- A single quote cannot occur with other single quotes even if preceded by a backslash
- Examples:

```
% echo 'Joe said 'Have fun''
Joe said Have fun
% echo 'Joe said "Have fun"'
Joe said "Have fun"
```

Double Quotes

- Double quotes protect all symbols and characters within the double quotes.
- Double quotes will not protect literal symbols: $ (dollar sign), ! (history event), and \ (backslash).
- Example:

```
% echo "I've been studying"
I've been studying
```

Quoting and Comments

'something': preserve literally

"something": allow $ variable expansion

$’C-escaped’: e.g., $’\a’

# comment

- Redirecting output

1>	redirect standard output (stdout)
2>	redirect standard error (stderr)
2>&1	redirect stderr into stdout

Variables

- Variables - just like any programming language
- Values are always stored as strings
- Mathematical operators in the shell language that will convert variables to numbers for calculations
- No need to declare a variable, just assigning a value to its reference will create it
- Local variables
  - Set by user within script or on the shell
- Environmental variables
  - Set by OS for use in current shell

Shell Variables

- ${N} = shell Nth parameter
- $$ = process ID
- $? = exit status

- Standard environment variables include:
  - HOME = home directory
  - PATH = list of directories to search
  - TERM = type of terminal (vt100, ...)
  - TZ = timezone (e.g., US/Eastern)
  - LOGNAME = user name

Variables and Expressions

- Variables are placeholders for the value
- shell does variable substitution
- $var or ${var} is value of variable
- assignment with var=value
- no space before or after!
  - Also, let “x = 17” or let “b = b + 10”
- uninitialized variables have no value
- variables are untyped
  - interpreted based on context

Export Command

- Export command puts variable into environment and accessible to child processes.

```
$ x=hello
$ bash          # Run a child shell
$ echo $x       # Nothing in x
$ exit          # Return to parent
$ export x
$ bash
$ echo $x
hello           # It's there
```

- Child can't modify parents original value

Expansion

Biggest difference to traditional languages

- shell substitutes and executes
- mix variables and code
- run-time code generation

For bash:

- brace expansion
- tilde expansion
- parameter and variable expansion
- command substitution
- arithmetic expansion
- word splitting
- filename expansion

Brace Expansion

- Expand comma-separated list of strings into separate words:

```
bash$ echo a{d,c,b}e
ade ace abe
```

- Useful for generating list of filenames:

```
mkdir /usr/local/{old,new,dist,bugs}
```

Tilde Expansion

- ~ expands to $HOME
- e.g.,
- ~/foo   /usr/home/foo
- ~hgs/src   /home/hgs/src

Filename Expansion

- Any word containing *?[is considered a pattern
- * matches any string
- ? matches any single character
- […] matches any of the enclosed characters

Quotes

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.
- Using double quotes to show a string of characters will allow any variables in the quotes to be resolved
- Using single quotes to show a string of characters will not allow variable resolution

Quoting

```
echo '$USER'
$USER
echo "$USER"
mmaxwell
echo "\""
"
echo '\"'
\"
```

Exit Status

- 0 is True

```
ls non-existant_file
echo $?
1
echo $?
0
```

Exit Status: exit

```
cat > test.sh                 <<_TEST_
exit 3
% chmod +x test.sh
% ./test.sh
% echo $?
3
```

File Attribute Operators

-d file	file exists and is a directory
-e file	file exists
-f file	file exists and is not a directory
-r file	you have read permissions on file
-s file	file exists and is not empty

file1 –nt file2	file1 is newer than file2
file1 –ot file2	file1 is older than file2

File Testing Example

```
#!/bin/bash
echo "Enter a filename: "
read filename
if [ ! –r "$filename" ]
 then
   echo "File is not read-able"
 exit 1
fi
```

Expressions

- An expression can be: String comparison, Numeric comparison, File operators and Logical operators and it is represented by [expression]:
- String Comparisons:

=	compare if two strings are equal
!=	compare if two strings are not equal
-n	evaluate if string length is greater than zero
-z	evaluate if string length is equal to zero

- Examples:

[s1 = s2]	(true if s1 same as s2, else false)
[s1 != s2]	(true if s1 not same as s2, else false)
[s1]	(true if s1 is not empty, else false)
[-n s1]	(true if s1 has a length greater then 0, else false)
[-z s2]	(true if s2 has a length of 0, otherwise false)

Expressions

- Number Comparisons:

-eq	compare if two numbers are equal
-ge	compare if one number is greater than or equal to a number
-le	compare if one number is less than or equal to a number
-ne	compare if two numbers are not equal
-gt	compare if one number is greater than another number
-lt	compare if one number is less than another number

- Examples:

[n1 -eq n2]	(true if n1 same as n2, else false)
[n1 -ge n2]	(true if n1greater then or equal to n2, else false)
[n1 -le n2]	(true if n1 less then or equal to n2, else false)
[n1 -ne n2]	(true if n1 is not same as n2, else false)
[n1 -gt n2]	(true if n1 greater then n2, else false)
[n1 -lt n2]	(true if n1 less then n2, else false)

Expressions

- Compound Logical Expressions

!	not	
&&	and	must be enclosed within [[ ]]
||	or	must be enclosed within [[ ]]

Examples:

```
if [ ! "$Years" -lt 20 ]; then
if [[ "$Status" = "H" && "$Shift" = 3 ]]
```

Flow Control

- if/else
- for
- case
- select
- while and until

Logic: test

```
% test 1 -lt 10
% echo $?
0

% test 1 = 10
% echo $?
1
```

Conditional Tests

- [] condition test
- -f file exists and is not a dir

```
[ -f /etc/passwd ]
echo $?
0

[ ! –f /etc/passwd ]
echo $?
1
```

Logic: if

```
if [ $USER –eq "mmaxwell" ]
then
   :
# "elif" a contraction of "else if":
elif something-else
then
   :
else
then
   :
fi
```

Logic: if

```
# see if a file exists
if [ -e /etc/passwd ]
then
  echo "/etc/passwd exists"
else
  echo "/etc/passwd not found!"
fi
```

Logic: for

```
$ for i in 1 2 3
> do
> echo $i
> done
1
2
3
```

Logic: for

```
$ for i in /*
> do
> echo "Listing $i:"
> ls -l $i
> read
> done
```

Logic: C-style for

```
LIMIT=10
for (( a=1       ;
       a<=LIMIT ;
       a++      ))
do
  echo -n “$a ”
done
```

Logic: while

```
a=0; LIMIT=10
while [ "$a" -lt "$LIMIT" ]
do
  echo -n "$a"
  a=$(( a + 1 ))
done
```

```
0 1 2 3 4 5 6 7 8 9
```

Break and Continue

Interrupt for, while or until loop

- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - resumes iteration of an enclosing
    - for
    - while
    - until
    - select

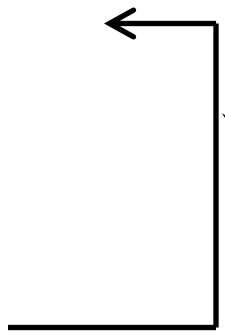
Break Command

```
while [ condition ]
do
    cmd-1
    break
    cmd-n
done
echo "done"
```

This iteration is over and there are no more iterations

Continue Command

```
while [ condition ]
do
    cmd-1
    continue
    cmd-n
done
echo "done"
```



This iteration is over; do the next iteration

Reading

- read inputs everything up to the newline

```
echo -n "Type: "
read ln
echo "You entered: $ln"
```

...a more useful example – next slide

Cursor keys

```
#!/bin/bash
ESC=$(echo -en "\033")              # define ESC
while :;do                          # infinite loop
# read quietly three characters of served input
read -s -n3 key 2>/dev/null >&2
  # if A is the result, print up
  if [ "$key" = "$ESC[A" ];then echo up;fi
  #  B dn
  if [ "$key" = "$ESC[B" ];then echo dn;fi
  #  C ri
  if [ "$key" = "$ESC[C" ];then echo ri;fi
  #  D  le
  if [ "$key" = "$ESC[D" ];then echo le;fi
done
```

Functions

- Bash has functions
  - Somewhat limited implementation
- Functions make scripts easier to maintain
- Function is a subroutine, code block, etc.
- Called by name
- Must be defined prior to being called
- May not be empty
- Can have strange names _ :
  - Obfuscation

cards.sh

```
#!/bin/bash
# Count how many elements.
Suites="Clubs Diamonds Hearts Spades"
Denominations="2 3 4 5 6 7 8 9 10 Jack Queen King Ace"
# Read into array variable.
suite=($Suites)
denomination=($Denominations)
# Count how many elements.
num_suites=${#suite[*]}
num_denominations=${#denomination[*]}
echo -n "${denomination[$((RANDOM%num_denominations))]} of "
echo ${suite[$((RANDOM%num_suites))]}
exit 0
```

```
Queen of Clubs
```

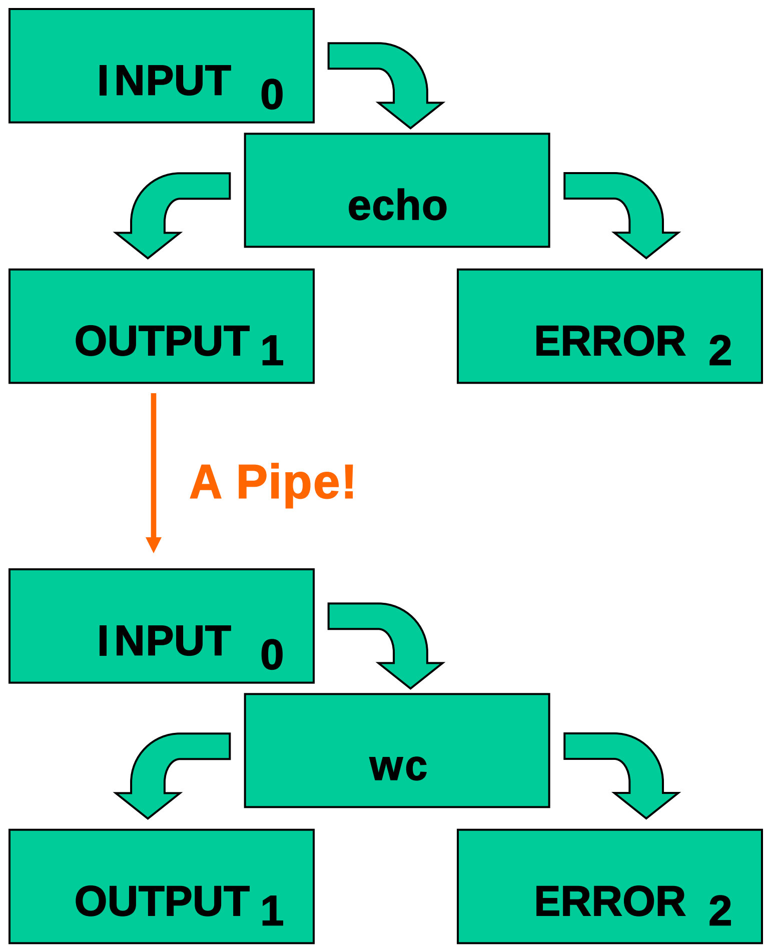
Change all filenames to lowercase

```
#!/bin/bash
# Traverse all files in directory
for filename in *
do
  fname='basename $filename'       # strip path
   n='echo $fname | tr A-Z a-z'    # to lower
  if [ "$fname" != "$n" ]
  then
    mv $fname $n
  fi
done
exit 0
```

Connecting Tools Together

Lots of Little Tools

```
echo “Hello” | \
  wc -c
```



Additional Resources

- Advanced Bash-Scripting Guide

  http://tldp.org/LDP/abs/html/

- GNU Bash

  http://www.gnu.org/s/bash/