CST8177 – Linux II

Shell Scripting

Shells

- A shell can be used in one of two ways:
  - A *command interpreter*, used interactively
  - A *programming language*, to write shell scripts (your own custom commands)

Shell scripting

- If we have a set of commands that we want to run on a regular basis, we could write a script
- A script acts as a Linux command, similarly to binary programs and shell built in commands
- In fact, check out how many scripts are in `/bin` and `/usr/bin`
  - `file /bin/* | grep 'script'`
  - `file /usr/bin/* | grep 'script'`
- As a system administrator, you can make your job easier by writing your own custom scripts to help automate tasks
- Put your scripts in `~/bin`, and they behave just like other commands (if your `PATH` contains `~/bin`)

Standard Script Header

- As we've already discussed, it's good practice to use a standard header at the top of our scripts
- You could put this in a file that you keep in a convenient place, and copy that file to be the beginnings of any new script you create
- Or, copy an existing script that already has the header

```
#!/bin/sh -u
PATH=/bin:/usr/bin ; export PATH    # add /sbin and /usr/sbin if needed
umask 022                           # use 077 for secure scripts
```

Interpreter Magic, or Shebang

- The interpreter magic, or "shebang":

```
#!/bin/sh -u
```

  - `#!` need to be the first two characters in the file, because they form a magic number that tells the kernel this is a script
  - `#!` is followed by the absolute path of the binary program that kernel will launch to interpret (that is, run) the script, `/bin/sh` in our case, and arguments can be supplied, `-u` in our case
  - The `-u` flag tells the shell to generate an error if the script tries to make use of a variable that's not set
    - That will never happen if the script is well written and tested
    - If it does happen, it's better to stop processing than continue processing garbage.

Standard Script Header (cont'd)

- Set the PATH
- The script will run the standard commands from the standard locations

```
PATH=/bin:/usr/bin ; export PATH # add /sbin and /usr/sbin if needed
```

- Set the umask
- Any files the script creates should have sane permissions, and we lean to the secure side

```
umask 022                          # use 077 for secure scripts
```

stdin, stdout,stderr

- We then follow the header with commands like the ones we type at the shell prompt.
- The stdin, stdout, stderr of the of the commands inside the script are the stdin, stdout, stderr of the script as it is run.
- When a command in your script prints output to stdout, your script will print that output to its stdout
- When a command in your script reads from stdin, your script reads from stdin

Scripting techniques

- Today we cover the following scripting topics
- Running scripts
  - arguments passed on the command line
  - ways to invoke a script
- Writing scripts
  - examining exit status
  - positional parameters and receiving arguments
  - variables
  - interacting with the user
  - the `test` program for checking things
  - control flow with if statements, looping, etc

Arguments on the command line

- we supply arguments to our script on the command line (as with any command args)
- `command` is executable and in `PATH`

```
command arg1 arg2 arg3
```

- `command.sh` is executable and in `PATH`

```
command.sh arg1 arg2 arg3
```

- `command.sh` is executable and not necessarily in `PATH`

```
./command.sh arg1 arg2 arg3
```

Arguments on the command line

- We can also invoke the script interpreter directly, with its own arguments
- We pass the file containing the script after the interpreter arguments
- The shebang line mechanism is not being used in this form

```
sh –u command.sh arg1 arg2 arg3
sh –u ./command.sh arg1 arg2 arg3
```

- The arguments seen by our script are

```
arg1 arg2 arg3
```

Quoting and arguments

`command "a b c"`
- 1 argument
  - `a b c`

`command 'a b c"' "d 'e f"`
- 2 arguments
  - `a b c"` and `d 'e f`

`command 'a ' b '"def"'`
- 3 arguments
  - `a ` and `b` and `"def"`

`command 'a b' "c 'd e' f"`
- 2 arguments
  - `a b` and `c 'd e' f`

Exit Status

- Each command finishes with an exit status
- The exit status is left in the variable `?` (`$?`)
- A non-zero exit status normally means something went wrong (`grep` is an exception)
- non-zero means "false"
- A exit status of `0` normally means everything was OK
- `0` means "true"
- `grep` returns `0` if a match occurred, `1` if not, and `2` if there was an error

Checking Exit status

- On the command line, after running a command we can use `echo $?` immediately after a command runs to check the exit status of that command

```
[tgk@kelleyt ~]$ ls
accounts  empty  rpm  test.sh
[tgk@kelleyt ~]$ echo $?
0
[tgk@kelleyt ~]$ ls nosuchfile
ls: cannot access nosuchfile: No such file or directory
[tgk@kelleyt ~]$ echo $?
2
[tgk@kelleyt ~]$
```

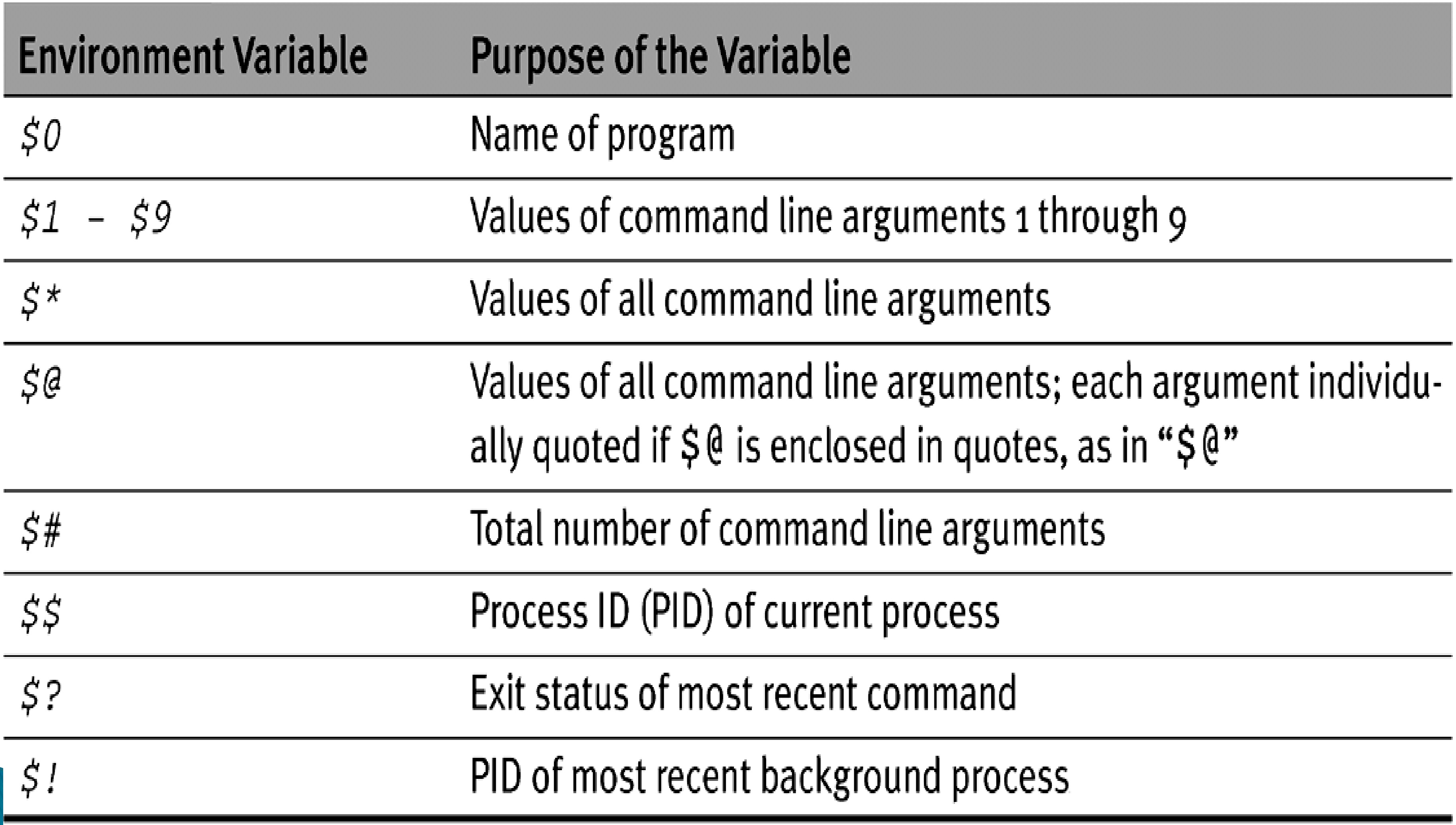
Positional Parameters

- When our script is running, the command line arguments are available as Positional Parameters
- The script accesses these through variables.
- `$#` holds the number of arguments on the command line, not counting the command itself
- `$0` is the name of the script itself
- `$1` through `$9` are the first nine arguments passed to the script on the command line
- After `$9`, there's `${10}`, `${11}`, and so on

Positional Parameters (cont'd)

- `$*` and `$@` both denote all of the arguments and they mean different things when double quoted:
  - `"$*"` is one word with spaces between the arguments
  - `"$@"` produces a list where each argument is a separate word

Positional Parameters (cont'd)



Environment Variable	Purpose of the Variable
`$0`	Name of program
`$1 - $9`	Values of command line arguments 1 through 9
`$*`	Values of all command line arguments
`$@`	Values of all command line arguments; each argument individually quoted if `$@` is enclosed in quotes, as in "`$@`"
`$#`	Total number of command line arguments
`$$`	Process ID (PID) of current process
`$?`	Exit status of most recent command
`$!`	PID of most recent background process

Sample script

```
#!/bin/sh -u
PATH=/bin:/usr/bin ; export PATH
umask 022

# Body of script
myvar="howdy doody"
echo "The value of \$myvar is: $myvar"  #notice backslash
echo "The number of arguments is: $#"
echo "The command name is $0"
echo "The arguments are: $*"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "The third argument is: $3"
```

Sample script

- How to write a command to swap two files?

```
$ cat swap
     #!/bin/bash
     mv $1 /tmp/$1
     mv $2 $1
     mv /tmp/$1 $2
     $ cat it1
     contents of file1
     $ cat it2
     contents of file2
     $ swap it1 it2
     $ cat it1
     contents of file2
     $ cat it2
     contents of file1
     $
```

Shift

- The shift command promotes each command line argument by one (e.g., the value in $2 moves to $1, $3 moves to $2, etc.)

```
$ cat shiftargs
#!/bin/bash
echo "The args are 0 = $0, 1 = $1, 2 = $2"
shift
echo "The args are 0 = $0, 1 = $1, 2 = $2"
shift
echo "The args are 0 = $0, 1 = $1, 2 = $2"
shift
$ shiftargs arg1 arg2 arg3
The args are 0 = shiftarg, 1 = arg1, 2 = arg2
The args are 0 = shiftarg, 1 = arg2, 2 = arg3
The args are 0 = shiftarg, 1 = arg3, 2 =
```

- The previous $1 becomes inaccessible

shift Example

How to write a general version of the swap command for two or more files?

```
swap f1 f2 f3 ... fn_1 fn
```

```
f1    <--- f2
f2    <--- f3
f3    <--- f4
...
fn_1 <--- fn
fn    <--- f1
```

Interacting with the user

- to get input from the user, we can use the `read` builtin
- `read` returns an exit status of 0 if it successfully reads input, or non-zero if it reaches EOF
- `read` with one variable argument reads a line from stdin into the variable
- Example:

```
#!/bin/sh -u
read aline #script will stop, wait for user
echo "you entered: $aline"
```

Interacting with the user (cont'd)

- Use the –p option to read to supply the user with a prompt
- Example

```
#!/bin/sh –u
read –p "enter your string:" aline
echo "You entered: $aline"
```

Interacting with the user (cont'd)

- `read var1` puts the line the user types into the variable `var1`
- `read var1 var2 var3` puts the first word of what the user types in to `var1`, the second word into `var2`, and the remaining words into `var3`

```
#!/bin/sh –u
read var1 var2 var3
echo "First word: $var1"
echo "Second word: $var2"
echo "Remaining words: $var3"
```

Backquotes: Command Substitution

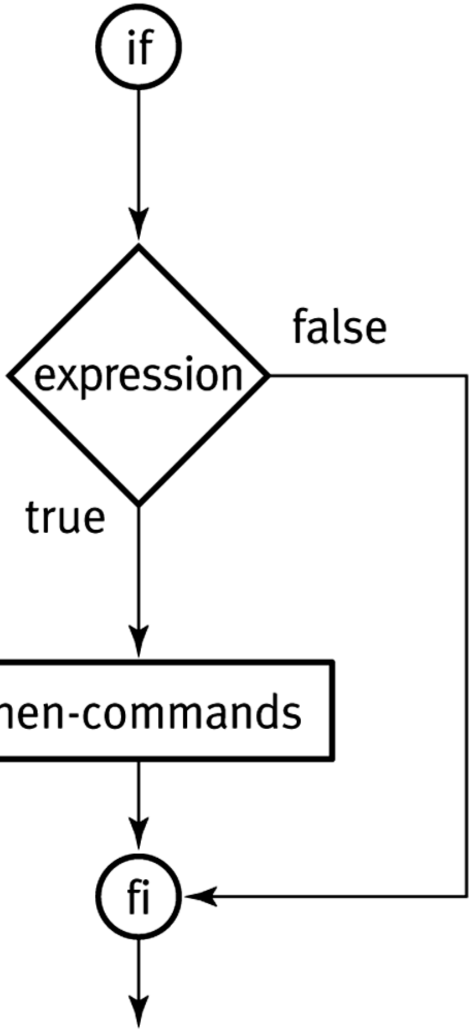
- A command or pipeline surrounded by backquotes causes the shell to:
  - Run the command/pipeline
  - Substitute the output of the command/pipeline for everything inside the quotes
- You can use backquotes anywhere:

```
$ whoami
wen99999
$ cat test7
#!/bin/bash
user=`whoami`
numusers=`who | wc -l`
echo "Hi $user! There are $numusers users logged on."
$ ./test7
Hi wen99999! There are       6 users logged on.
```

Control Flow

- The shell allows several control flow statements:
  - `if`
  - `while`
  - `for`

Semantics of the if statement



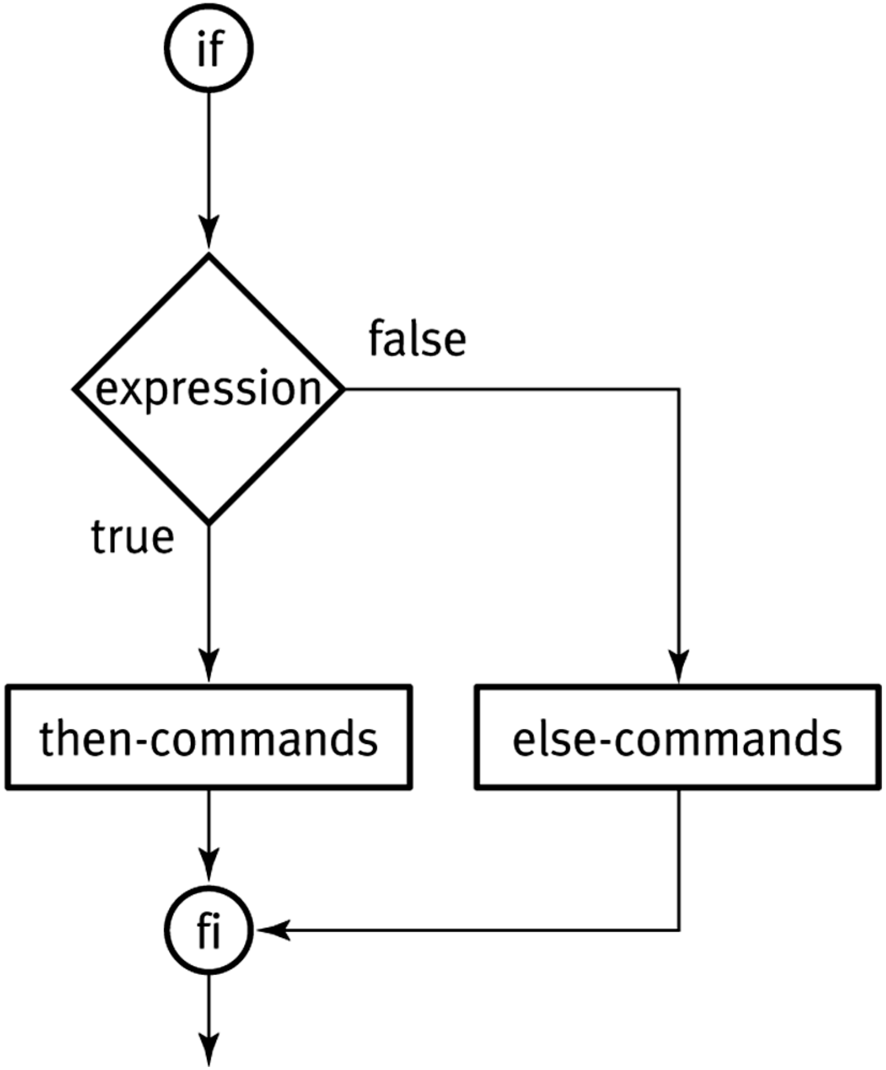
If statement

```
if list1; then
    list2
fi
```

```
if list1
    then
        list2
fi
```

- `list1` is executed, and if its exit status is 0, then `list2` is executed
- A `list` is a sequence of one or more pipelines, but for now, let's say it's a command

Semantics of the if and else statement



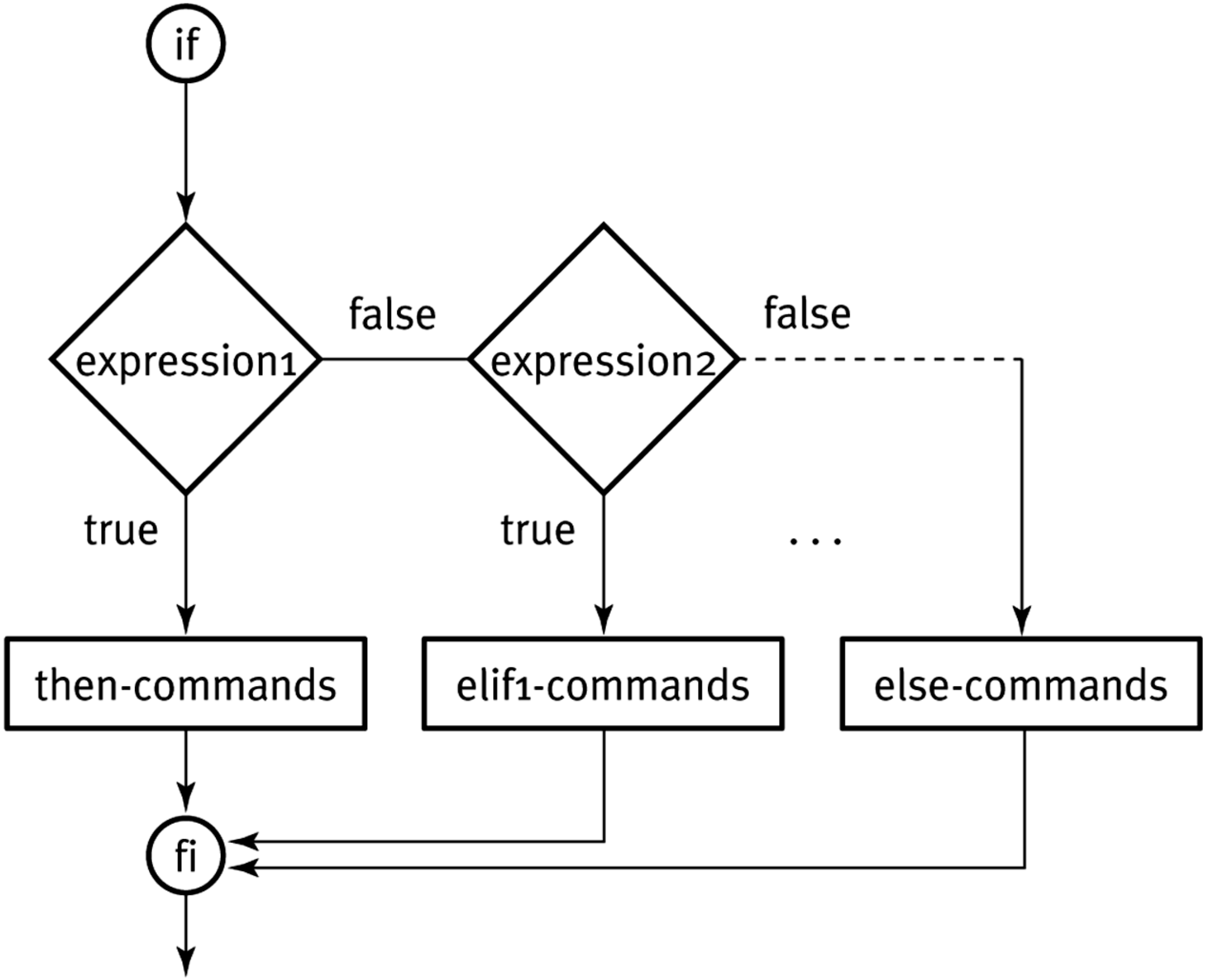
if …then … elif

- We can include an else clause, with commands to run if `list1` is false (has exit status of non-zero)

```
if list1; then
    list2
else
    list3
fi
```

```
if list1
    then
        list2
    else
        list3
fi
```

Semantics of the if…then…elif statement



if … then … elif

- The elif statement combines else and if to construct a nested set of if…then…else structure.

```
if list1
    then
         list2
    elif list3
         then
              list4
    …
    else
         list5
fi
```

Boolean Expressions

- Relational operators:

-eq, -ne, -gt, -ge, -lt, -le

- File operators:

-f *file* True if *file* exists and is not a directory

-d *file* True if *file* exists and *is* a directory

-s *file* True if *file* exists and has a size > 0

- String operators:

-z *string* True if the length of *string* is zero

-n *string* True if the length of *string* is nonzero

s1 = *s2* True if *s1* and *s2* are the same

s1 != *s2* True if *s1* and *s2* are different

s1 True if *s1* is not the null string

Integer tests (man test)

- INTEGER1 –**eq** INTEGER2
  INTEGER1 is equal to INTEGER2
- INTEGER1 –**ge** INTEGER2
  INTEGER1 is greater than or equal to INTEGER2
- INTEGER1 –**gt** INTEGER2
  INTEGER1 is greater than INTEGER2
- INTEGER1 –**le** INTEGER2
  INTEGER1 is less than or equal to INTEGER2
- INTEGER1 –**lt** INTEGER2
  INTEGER1 is less than INTEGER2
- INTEGER1 –**ne** INTEGER2
  INTEGER1 is not equal to INTEGER2

String tests (man test)

- **-n** STRING

  the length of STRING is nonzero
- STRING equivalent to **-n** STRING
- **-z** STRING

  the length of STRING is zero
- STRING1 = STRING2

  the strings are equal
- STRING1 != STRING2

  the strings are not equal

file tests (man test)

- These are just a few of them See `man test` for more:
- **-d** FILE

  FILE exists and is a directory
- **-e** FILE

  FILE exists
- **-f** FILE

  FILE exists and is a regular file
- **-r** FILE

  FILE exists and read permission is granted
- **-w** FILE

  FILE exists and write permission is granted
- **-x** FILE

  FILE exists and execute (or search) permission is granted

Test program

- A common command to use in the test list of an `if` statement is the `test` command
- `man test`
- Examples:
- `test -e /etc/passwd`
- `test "this" = "this"`
- `test 0 -eq 0`
- `test 0 -ne 1`
- `test 0 -le 1`

If statement with test

```
if test "$1" = "hello"; then
   echo "First arg is hello"
fi

if test "$2" = "hello"; then
   echo "Second arg is hello"
else
   echo "Second arg is not hello"
fi
```

The program named [

```
[wen001:centOS65 ~]$ ls -li /usr/bin/test /usr/bin/[
786463 -r-xr-xr-x  1 root  root  34716 22 Nov  2013 /usr/bin/[
786517 -r-xr-xr-x  1 root  root  31124 22 Nov  2013 /usr/bin/test
```

- notice that on OSX, `[` is another name for the `test` program:

```
if [ -e /etc/passwd ]; then
    echo "/etc/passwd exists"
fi
```

is the same as

```
if test -e /etc/passwd; then
    echo "/etc/passwd exists"
fi
```

Practicing with [

```
$ [ 0 –eq 0 ]
$ echo $?
0
$ [ "this" = "that" ]
$ echo $?
1
$ [ "this" = "this" ]
echo $?
0
$ ["this" = "this"]                    # forgot the space after [
–bash: [this: command not found
$ [ "this" = "this"]                   # forgot the space before ]
–bash: [: missing ']'
```

Combining tests

- (EXPRESSION)

  EXPRESSION is true
- ! EXPRESSION

  EXPRESSION is false
- EXPRESSION1 –**a** EXPRESSION2

  both EXPRESSION1 and EXPRESSION2 are true
- EXPRESSION1 –**o** EXPRESSION2

  either EXPRESSION1 or EXPRESSION2 is true

And, Or, Not

- You can combine and negate expressions with:

-a	And
-o	Or
!	Not

```
$ cat test10
#!/bin/bash
if [ `who | grep gates | wc -l` -ge 1 -a `whoami` != “gates" ]
then
    echo "Bill is loading down the machine!"
else
    echo "All is well!"
fi
$ test10
Bill is loading down the machine!
```

test examples

- `test` is a program we run just to find out its exit status
- The arguments to the `test` command specify what we're testing
- The spaces around the arguments are important because `test` will not separate arguments for you:
  - `"a" ="a"` is the same as `a =a` which is two args and `test` wants three with the second one `=`
- When trying out `test` examples, we can run test and find out the results by looking at `$?` immediately after the `test` command finishes

test examples (cont'd)

- Alternatively, we can try any example by putting it in an `if`-statement:

```
if [ 0 -eq 1 ]; then
   echo that test is true
else
   echo that test is false
fi
```

test examples (strings)

- Is the value of `myvar` an empty (zero-length) string?

```
[ -z "$myvar" ]
```

- Is the value of `myvar` a non-empty string?

```
[ -n "$myvar" ]
```

or

```
[ "$myvar" ]
```

test examples (strings cont'd)

- Is the value of `myvar` equal to the string `"yes"`?

```
[ "$myvar" = "yes" ]
```

or

```
[ "$myvar" = yes ]
```

or

```
[ "yes" = "$myvar" ]
```

or

```
[ yes = "$myvar" ]
```

test examples (strings cont'd)

- Is the value of `myvar` NOT equal to the string `"yes"`?

```
[ "$myvar" != "yes" ]
```

or

```
[ ! "$myvar" = yes ]
```

or

```
[ "yes" != "$myvar" ]
```

or

```
[ ! yes = "$myvar" ]
```

test examples (integers)

- Is the value of `myvar` a number equal to `4`?

```
[ "$myvar" -eq "4" ]
```

or

```
[ "$myvar" -eq 4 ]
```

- Notice that double quotes around a number just means the shell will not honor special meaning, if any, of the characters inside
- Digits like `4` have no special meaning in the first place, so double quotes do nothing

test examples (integers)

- Is the value of `myvar` a number NOT equal to `4`?

```
[ "$myvar" -ne 4 ]
```

or

```
[ ! 4 -eq "$myvar" ]
```

or

```
[ ! "$myvar" -eq 4 ]
```

or

```
[ "$myvar" -ne 4 ]
```

test examples (integers)

- Is `00` a number equal to `0`? yes

```
[ 00 -eq 0 ]
```

- Is `004` a number equal to `4`? yes

```
[ 004 -eq 4 ]
```

- Notice double quotes don't change anything
- Is `00` equal to `0` as strings? no

```
[ 00 = 0 ]
```

- Is `0004` equal to `4` as strings? no

```
[ 0004 = 4 ]
```

test examples

- Is `abc` a number equal to `0`? error

  `[ abc -eq 0 ]` ERROR `abc` is not a number
- The following is the same as `[ 1 ]` with stdin redirected from file named `2`

```
[ 1 < 2 ]
```

- Remember we can put redirection anywhere in the command we want:

```
ls > myfile
```

is the same as

```
> myfile ls
```

test examples (files)

- Does `/etc/passwd` exist?

```
[ -e /etc/passwd ]
```

- Does `/etc` exist?

```
[ -e /etc ]
```

- Does the value of `myvar` exist as a file or directory?

```
[ -e "$myvar" ]
```

test examples (files)

- Is `/etc/passwd` readable?

```
[ -r /etc/passwd ]
```

- Is `/etc` readable?

```
[ -r /etc ]
```

- Is the value of `myvar` readable as a file or directory?

```
[ -r "$myvar" ]
```

- Not readable?

```
[ ! -r "$myvar" ]
```

test (combining tests)

- If we need to check whether two files both exist, we check for each individually, and combine the tests with `-a`, meaning AND

```
[ -e /etc/foo -a -e /etc/bar ]
```

- Given a number in `myvar` we can check whether it's greater than or equal to `4` AND less than or equal to `10`

```
[ "$myvar" -ge 4 -a "$myvar" -le 10 ]
```

test (combining tests)

- If we need to check whether at least one of two files exists, we check for each individually, and combine the tests with `-o`, meaning OR

```
    [ -e /etc/foo -o -e /etc/bar ]
```

- Given a number in `myvar` we can check whether it's greater than or equal to `4` OR less than or equal to `10`

```
[ "$myvar" -ge 4 -o "$myvar" -le 10 ]
```

test (not)

- We can use `!` to test if something is NOT true
- Test whether /etc/passwd is NOT executable

```
[ ! -e /etc/passwd ]
```

test (parenthesis)

- Just like arithmetic, we use parenthesis to control the order of operations
- Remember that (and) are special to the shell so they need to be escaped or quoted from the shell
- Check whether `file1` or `file2` exists, and also check whether `1` is less than `2`:

```
[ \( -e file1 -o -e file2 \) -a 1 -lt 2 ]
```

- Without parentheses we'd be testing whether `file1` exists, or whether `file2` exists and `1` is less than `2`

test (order of operations)

- Like regular expressions, to get comfortable with the order of operations, we can borrow our comfort with arithmetic expressions

test operation	arithmetic alalog	comment
( )	( )	\(and \) or '(' and ')' to protect from shell
!	–	That's the arithmetic unary "oposite of" operator, as in –4 or –(2+2)
–a	multiplication	
–o	addition	

Example 1: capitalize.sh

```
#!/bin/sh -u
PATH=/bin:/usr/bin ; export PATH
umask 022
echo "You passed $# arguments, and those are:$*:"
if [ $# -eq 0 ]; then
    echo "You didn't give me much to work with"
else
    echo -n "Here are the arguments capitalized:"
    echo "$*" | tr '[[:lower:]]' '[[:upper:]]'
fi
```

stderr versus stdout

- Often the purpose of a script is to produce useful output, like filenames, or maybe a list of student numbers
  - this output should go to stdout
  - it may be redirected to a file for storage
  - we don't want prompts and error messages in there
- There may also be other output, like warning messages, error messages, or prompts for the user, for example
  - this output should go to stderr
  - we don't want this type of output to be inseparable from the real goods the script produces

Error Messages

- Here is an example of a good error message

```
echo 1>&2 "$0: Expecting 1 argument; found $# ($*)"
```

- Why is it good?
  - It redirects the message to stderr: `1>&2`
  - It gives the user all the information they may need to see what is wrong
    - `$0` is the name used to invoke the script (remember, files can have more than one name so it shouldn't be hard-coded into the script)
    - `$#` is the number of arguments the user passed
    - `$*` shows the actual arguments, put in parenthesis so the user can see spaces, etc.

Example 2: match.sh

```
#!/bin/sh –u
PATH=/bin:/usr/bin ; export PATH
umask 022
if [ $# -ne 1 ]; then
    echo 1>&2 "$0: Expecting 1 argument; found $# ($*)"
else
    read -p "Enter your string:" userString
    if [ "$userString" = "$1" ]; then
        echo "The string you entered is the same as the argument"
    else
        echo "The string you entered is not the same as the argument"
    fi
fi
```

For loop

```
for name [ in word... ] ; do list ; done
```

- `name` is a variable name we make up
- `name` is set to each `word...` in turn, and list is exectuted
- if `[ in word... ]` is omitted, the positional parameters are used instead

For loop example

```
for f in hello how are you today; do
   echo "Operating on $f"
done
```

While loop

```
while command; do
   # this code runs over and over
   # until command has
   # non-zero exit status
done
```

While loop example

```
while read -p "enter a word: " word; do
    echo "You entered: $word"
done
```