

Shell Programming

- Shell script (shell program)
 - a series of Unix commands placed in an ASCII file
 - Each shell provides a mechanisms for control (e.g. if, for, do)
 - Unix commands + shell variables + control mechanisms = high-level programming language
 - Alternative scripting languages include perl, python, and awk.

Executing Shell Scripts

- Explicit shell execution
 - sh MyScript
 - csh MyScript
 - ksh MyScript
- Implicit shell execution
 - MyScript
 - first line in script is:
 - #!
 - #!/bin/ksh
 - #
 - use csh
 - # indicates a script comment
 - Use chmod to make “executable”
 - chmod [augo][+ -][rwx] <filename>
 - chmod o+x MyScript

Example #1

- Creating a simple shell script

```
cat > gohome
```

```
#!/bin/sh
```

```
cd ~
```

```
ls -F
```

```
^D
```

- Changing permissions

```
chmod u+x gohome
```

- Executing the script

```
gohome
```

Shell Variables

- A shell variable can be any string with the following form:
`[A-Za-z][A-Za-z0-9]*`
- sh and ksh
 - `variable=value`
 - `export variable`
 - `export` makes value visible to child processes
- csh
 - `set variable=value`
 - `setenv variable value`
 - `setenv` makes value visible to child processes

Using Variables

- `person=alex`
 - notice that no spaces are adjacent to the equal sign
- To retrieve the value, precede the variable name with dollar sign (\$)

```
echo person
```

```
person
```

```
echo $person
```

```
alex
```

- Unsetting Variables

```
person=
```

- ksh and sh only

```
unset person
```

Some Built-in Variables

- ksh and sh use upper case letters
- csh uses lower case
 - CDPATH or cdpath
 - search path for cd command
 - HOME or home
 - path to user's home directory
 - PATH or path
 - search path for commands
 - PS1 or prompt
 - command-line prompt
 - HISTSIZE or history
 - number of commands recorded for recall

Shell Variables and Quotes

- Quotes are used to specify values which contain spaces or special characters
- Double quotes prevent shell interpretation of special characters except the \$.

```
people="jack and jill *"
```

```
echo "$people"
```

```
jack and jill *
```

```
echo $people
```

```
jack and jill file1 file2 file3
```

Single Quotes and Backslash

- Single quotes prevent shell interpretation of all special characters.

```
echo '$people'
```

```
$people
```

- The backslash (\) prevents shell interpretation of the next character

```
echo \$people
```

```
$people
```

Command Line Variables

- Q: How do we handle command line arguments?

- Example:

- `postgrades -s hw1`

- A: The shell assigns numbered variables to each word in the command.

- `$0=postgrades`

- `$1=-s`

- `$2=hw1`

- Only \$0 to \$9 available in sh
- Use brackets for larger values in csh and ksh (e.g. `${10}`)

Changing the Values of Command line Variables

- `set` - manually sets the values of parameters
 - example:
 - `set a b c d`
 - `echo $1 $2 $3 $4`
 - `a b c d`
- `shift` - eliminate parameter 1 and renumber the parameters
 - Example
 - `set a b c d`
 - `shift`
 - `echo $1 $2 $3`
 - `b c d`

Example

```
cat ex1
```

```
#!/bin/sh
```

```
# Echo and variable example
```

```
person=John
```

```
sport=soccer
```

```
place=$1
```

```
echo "Did $person play $sport at \  
"$1"?"
```

```
ex1 LC
```

```
Did John play soccer at LC?
```

More Access to Command-line arguments (sh & ksh)

- `$#` = number of command line arguments
- `$*` = all command line arguments with single quotes around whole group
- `@$` = all command line arguments with single quotes around each word.

More Special Variables (ksh &sh)

- \$\$ - process number of current shell (also available in csh)
- \$? - exit status of last command
- \$! - process number of last background command

Example

```
cat ex2
```

```
#!/bin/sh
```

```
# Command line arguments
```

```
echo "Argument 1 = $1"
```

```
echo "Argument 2 = $2"
```

```
echo "All args = $*"
```

```
ex2 hello world !!
```

```
Argument 1=hello
```

```
Argument 2=world
```

```
All args = hello world !!
```

Backquotes

- Backquotes around a command are used to capture its output.

```
cat ex4
```

```
today=`date`
```

```
echo "Today is "$today
```

```
echo "user is `whoami`"
```

```
cat ex5
```

```
set `date`
```

```
echo $*
```

```
echo $2 $3, $6
```

```
ex5
```

```
Sun Oct 06 12:04:09 EST 2002
```

```
Oct 06, 2002
```

The Read Command

- Use the read command to obtain input at run-time

```
cat read1
```

```
echo "Input something: \c"
```

```
read userinput
```

```
echo "You entered: $userinput"
```

```
read1
```

```
Input something: Unix is fun
```

```
You entered: Unix is fun
```

Read Command (cont.)

- read assigns one word to each variable with all leftover words going to the last variable

```
cat readex
```

```
echo "Input:\c"
```

```
read word1 word2 word3 rest
```

```
echo $word1 $rest
```

```
readex
```

```
Input: Unix is sure fun, isn't it?
```

```
Unix fun, isn't it?
```

If - Then - Else

```
if <test-command>
```

```
then
```

```
    <commands>
```

```
[else
```

```
    <commands>]
```

```
fi
```

```
if test "$#" != 2
```

```
then
```

```
    echo "Please supply 2 args"
```

```
    exit 0
```

```
fi
```

For loops

- For loops can be used to iterate through a list of objects

```
#!/bin/sh
```

```
for name in sue joe bill cindy
```

```
do
```

```
    print "hello" | mail ${name}
```

```
done
```

```
#!/bin/csh
```

```
foreach name (sue joe bill cindy)
```

```
end
```

for examples

```
#!/bin/ksh
```

```
# change names to name.old
```

```
for x in $@
```

```
do
```

```
    mv $x $x.old
```

```
done
```

```
# do ls on every dir in the path
```

```
IFS=":"
```

```
for x in $PATH
```

```
do
```

```
    ls $x
```

```
done
```

Conditions

- test command is used to evaluate conditions:
 - test “\$2” -eq 10 # test equality
 - test “\$2” -lt 0 # test less than
 - test -r “\$1” # see if \$1 is a readable file

Other test switches:

- -s file exists and is not empty
- -f file is an ordinary file
- -d file is a directory
- -w file is writeable
- Other numerical relationships:
 - -ne, -gt, -ge, -le
- And, or, and not
 - -a, -o, !

A Better Syntax for test

- The test command is equivalent to placing a condition in brackets.
- These are equivalent:
 - test -r filename
 - [-r filename]
- There must be a space between each bracket and the condition.

Evaluating Expressions

- Bourne shell and Korn shell
 - use `expr` command
 - arithmetic operators (+, -, *, /, %)
 - Relational operators:
 - (<, <=, =, !=, >=, >)
 - examples
 - `x = `expr $x + 1``
 - `x = `expr $1 * $2 / $3 - 10``
 - `expr 5 \< $1`
- C shell
 - use `@` command
 - `@ x += 1`
 - `@ x = $1 * $2 / $3 - 10`
 - `@ 5 \< $1`

While loops

```
#!/bin/sh
```

```
while [ ${month} -le 12 ]
```

```
do
```

```
    process ${month}
```

```
    month = `expr ${month} + 1`
```

```
done
```

```
#!/bin/csh
```

```
set month = 1
```

```
while (${month} <= 12)
```

```
    process ${month}
```

```
    @ month = ${month} + 1
```

```
end
```

case statement

```
#!/bin/sh
```

```
case $variable in
```

```
    pattern1) command list;;
```

```
    pattern2) command list;;
```

```
esac
```

```
#!/bin/csh
```

```
switch ( $variable )
```

```
case value1:
```

```
    command list
```

```
    breaksw
```

```
case value2:
```

```
endsw
```

Case Example

case "\$month" in

1|[jJ][aA]*)

 \$month = "January"

;;

2|[fF]*)

 \$month = "February"

;;

*) # default case

 \$month = "Unknown"

;;

esac

Reading from a File

- Use a descriptor number to read a file

```
exec 3< ${filename}  
while [ read -u3 line ]  
do  
    print "$line"  
done
```

Shell processes

- Generally, each command that is executed by a shell is executed as a separate process
- Shell main loop:
 - read command
 - create child process for command
 - if command did not end with `&`
wait for child process to complete
- Exceptions:
 - built-in shell commands (e.g. `cd`, `pwd`)

Handling Signals

- The execution of the shell script can be interrupted by a number of signals

1 Hangup

2 Interrupt

3 Quit

9 Kill

15 Terminated

```
trap 'rm *.tmp; exit 1' 1 2 3 9 15
```

Undefined Variables

- Referencing undefined variables will cause an error in a shell script
- To provide a default value for an evaluation
 - `${myvar:-default}`
 - `${x:-0}`
 - This does not change the value of x
- To provide a default value for a variable
 - `${myvar:+default}`
 - `${x:+100}`
 - This sets x to 100 if it is undefined