CSH SCRIPTS

What is a Shell Script?

- A shell script is an executable file which contains shell commands. The script acts as a "program" by sequentially executing each command in the file.
- Each of the 5 common shells has its own scripting language. There are similarities between the languages, but also differences.
- A scripting language consists of control structures, shell commands, expressions and variables.
- If a shell script written in a given scripting language must run under the appropriate shell, the first line of the script should specify the shell it must run under. For example, a C shell script should have as the first line:

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

- Script files should be given "execute" file permission with the chmod command:

```
chmod u+x myscript
```

- A simple shell script which will run under any of the 5 shells appears below. This script simply displays a greeting and the date/time. Comments are preceeded with a pound sign (#):

```
#Simple Script
#
echo 'Welcome to the world of script files'
date
```

Simple Script Exercise

1. Open a new window or use an existing window. Change directory to the Scripts exercise directory:

```
cd ~/Scripts
```

2. Edit a new file - call it **simple**. Include the following lines in your **simple** file:

```
#!/bin/csh
#
#Simple Script
#
echo Welcome to the world of script files
echo -n "The current time and date are: "
date
```

   Save your file and exit the editor.

3. Make this file world executable. Check its permissions to make sure:

```
chmod a+x simple
ls -l simple
```

4. Execute the file by invoking its name: **simple**. Observe the output. How does the **-n** option modify the output from **echo**?

5. Return to the tutorial to learn about the next section before proceeding.

Expressions

- Expressions are statements, composed of constants, variables and operators, which are evaluated to determine a result. Expressions can be either mathematical or logical.

  Example 1: Mathematical expression, where "a" and "t" are variables, "3" and "4" are constants, and "=" and "+" are operators.

  ```
  a  = 3 + 4t
  ```

  Example 2: Logical expression, which evaluates to either "true" or "false". The string "exit" is a constant, "var" is a variable, and "(", ")" and "==" are operators.

  ```
  ($var == exit)
  ```

- Shell scripts commonly use expressions. Each shell has its own rules for writing expressions, however.

- The C Shell recognizes the following operators, in order of precedence.

  ```
  ()       - parenthesis - change order of evaluation
  -        - unary minus/negation
  ~       - one's complement
  !        - logical negation
  %        - remainder
  /        - divide
  *        - multiply
  -        - subtract
  +        - addition
  >>       - shift right
  <<       - shift left
  >        - greater than
  <        - less than
  >=       - greater than or equal
  <=       - less than or equal
  !=       - not equal to (strings)
  ==       - equal to (strings)
  &        - bitwise AND
  ^        - bitwise exclusive OR
  |        - bitwise inclusive OR
  &&       - logical AND
  ||       - logical OR
  ```

Expressions Exercise

1. Display the file **simple.2** which is already provided for you in your **~/Scripts**directory. Note the use of a couple simple expressions.

2. Check to make sure the file is executable. If it isn't, then use the **chmod** to make it executable at

least by you.

```
ls -l simple.2
chmod u+x
```

3. Execute the example file by invoking its name: **simple.2** and observe the output.
4. Now, edit the file and add the lines below. When finished, save the file and try executing it again.

```
@ count = 1
echo Count begins as $count
@ count = $count + 100
echo Count plus 100 = $count
```

5. Return to the tutorial to learn about the next section before proceeding.

Control Structures

Script languages make use of programming control structures, such as "if" statements and "loops". Those for C Shell are described below.

if

Used to test an expression and then conditionally execute a command. If the specified expression evaluates true, then the single command with arguments is executed. Command must be a simple command, not a pipeline, a command list, or a parenthesized command list.

Syntax:

```
if (expr) command [arguments]
```

Example:

```
#!/bin/csh
if ($#argv == 0) echo There are no arguments
```

In addition to the C Shell's logical expressions, you can use expressions that return a value based upon the status of a file. For example:

```
if (-e myfile) echo myfile already exists
```

The possible file status expressions are:

```
d     - file is a directory
e     - file exists
f     - file is an ordinary file
o     - user owns the file
r     - user has read access to the file
w     - user has write access to the file
x     - user has execute access to the file
z     - file is zero bytes long
```

---

if Exercise

1. Create a new file called **simple.3** which include these lines:

```
#!/bin/csh
if ($#argv == 0) echo There are no arguments
if ($#argv != 0) echo There are $#argv arguments
```

2. After saving the file, make it executable and then run it as shown:

```
chmod +x simple.3
simple.3
```

3. Now try running it several more times, but with a different number of arguments each time. For example:

```
simple.3  one two
simple.3  a b c 3 4
simple.3  9*3 65 100.34     - will cause an error - why?
simple.3  '9*3' 65 100.34
simple.3  '10 disc write'   - why is this considered 1?
```

4. Edit the file again, adding the following lines:

```
if (-e $1) echo The file: $1 exists.
if (! -e $1) echo The file: $1 does not exist.
```

5. Now execute the file with a filename as the only argument and observe the output:

```
simple.3  nosuchfile
simple.3  simple.3
simple.3  simple
```

6. Return to the tutorial to learn about the next section before proceeding.

---

if / then / else

Used to test multiple conditions and to execute more than a single command per condition. If the specified expr is true then the commands to the first else are executed; otherwise if expr2 is true then the commands to the second else are executed, etc. Any number of else-if pairs are possible; only one endif is needed. The else part is likewise optional.

The words else and endif must appear at the beginning of command lines; the if must appear alone on its command line or immediately after an else.

Syntax:

```
if (expr) then
   commands
else if (expr2) then
   commands
else
   commands
endif
```

Example:

```
#!/bin/csh
if ($#argv == 0) then
   echo "No number to classify"
else if ($#argv > 0) then
   set number = $argv[1]
   if ($number < 0) then
      @ class = 0
   else if (0 <= $number && $number < 100) then
      @ class = 1
   else if (100 <= $number && $number < 200) then
      @ class = 2
   else
      @ class = 3
   endif
   echo The number $number is in class $class
endif
```

if then else Exercise

1. Display the file **simple.4** which is already provided for you in your **~/Scripts** directory. It is the same as the one from the tutorial. Attempt to understand what the expressions are evaluating and the logic of the if-then-else nesting.

2. Make sure the file is executable (you know how to do this by now) and then try running it with different arguments. Compare the output with the script's logic as you understand it. Do they agree?

```
simple.4 34
simple.4 199
simple.4 1000
simple.4 -33
simple.4
```

3. Return to the tutorial to learn about the next section before proceeding.

foreach / end

The foreach statement is a type of loop statement. The variable name is successively set to each member of wordlist and the sequence of commands until the matching end statement are executed. Both foreach and end must appear alone on separate lines.

Syntax:

```
foreach name (wordlist)
    commands
end
```

Example:

```
#!/bin/csh
foreach color (red orange yellow green blue)
   echo $color
end
```

while / end

The while statement is another type of loop statement. Statements within the while/end loop are conditionally executed based upon the evaluation of the expression. Both while and end must appear alone on separate lines.

Syntax:

```
while (expression)
    commands
end
```

Example:

```
#!/bin/csh
set word = "anything"
while ($word != "")
  echo -n "Enter a word to check (Return to exit): "
  set word = $<
  if ($word != "") grep $word /usr/share/dict/words
end
```

---

foreach Exercise

1. Display the file **simple.5** which is already provided for you in your **~/Scripts**directory. Understand what it does and its use of the **foreach** construct.

2. Make sure the file is executable (you know how to do this by now) and execute it. Does it correctly determine your login machine?

while Exercise

1. Now create your own executable script by copying the **while** loop example from the tutorial. It is reproduced below for convenience:

```
#!/bin/csh
set word = "anything"
while ($word != "")
   echo -n "Enter a word to check (Return to exit): "
   set word = $<
   if ($word != "") grep $word /usr/share/dict/words
end
```

2. Give it execute permission and then run it. What does it do?

3. Return to the tutorial to learn about the next section before proceeding.

---

break

Used to interrupt the execution of a foreach or while loop. Transfers control to the statement after the end statement, thus terminating the loop. If there are other commands on the same line as a break statement, they will be executed before the break occurs. Multi-level breaks are thus possible by writing them all on one line.

Syntax:

```
break
```

Example:

```
#!/bin/csh
foreach number (one two three exit four)
  if ($number == exit) then
    echo reached an exit
    break
  endif
  echo $number
end
```

---

continue

Used to interrupt the execution of a foreach or while loop. Transfers control to the end statement, thus continuing the loop. If there are other commands on the same line as a continue statement, they will be executed before the continue occurs.

Syntax:

```
continue
```

Example:

```
#!/bin/csh
foreach number (one two three exit four)
  if ($number == exit) then
    echo reached an exit
    continue
  endif
  echo $number
end
```

---

goto

The goto statement transfers control to the statement beginning with label:

Syntax:

```
goto label
```

Example:

```
#!/bin/csh
if ($#argv != 1) goto error1
if ($argv[1] < 6) goto error2
goto OK

error1:
  echo "Invalid - wrong number or no arguments"
  echo "Quitting"
  exit 1
```

```
error2:
  echo "Invalid argument - must be greater than 5"
  echo "Quitting"
  exit 1

OK:
  echo "Argument = $argv[1]"
  exit 1
```

---

switch / case / breaksw / endsw

The switch structure permits you to set up a series of tests and conditionally executed commands based upon the value of a string. If none of the labels match before a ‘default’ label is found, then the execution begins after the default label.

Each case label and the default label must appear at the beginning of a line. The command breaksw causes execution to continue after the endsw. Otherwise control may fall through case labels and default labels. If no label matches and there is no default, execution continues after the endsw.

Syntax:

```
switch (string)
case str1:
  commands
  breaksw
case str2:
  commands
  breaksw
...
default:
  commands
  breaksw
endsw
```

Example:

```
#!/bin/csh
if ($#argv == 0 ) then
   echo "No arguments supplied...exiting"
   exit 1
else
   switch ($argv[1])
   case [yY][eE][sS]:
     echo Argument one is yes.
     breaksw
   case [nN][oO]:
     echo Argument one is no.
     breaksw
   default:
     echo Argument one is neither yes nor no.
     breaksw
   endsw
endif
```

---

switch / case / breaksw / endsw Exercise

1. Display the file **simple.6** which is already provided for you in your **~/Scripts**directory. This file is a little lengthier than previous files. Notice the use of the switch construct.

2. Make sure the file is executable (you know how to do this by now) and execute it.

3. Now edit the **simple.6** file and add the necessary commands to handle the month of Dec. Re-execute the script. Does it still work?

4. Return to the tutorial to learn about the next section before proceeding.

Interrupt handling

The onintr statement transfers control when you interrupt (CTRL-C) the shell script. Control is transferred to the statement beginning with label:

Can be useful for gracefully cleaning up temporary files and exiting a program should it be interrupted.

Syntax:

```
onintr  label
```

Example:

```
#!/bin/csh
onintr close
while (1 == 1)
  echo Program is running
  sleep 2
end

close:
echo End of program
```

Miscellaneous

Miscellaneous tasks for C Shell programming are described below.

Using quotes

The shell uses both single (' ') quotes and double (" ") quotes. They have different effects.

Single quotes:
- allow inclusion of spaces
- prevent variable substitution
- permit filename generation

Double quotes:
- allow inclusion of spaces
- permit variable substitution
- permit filename generation

Example 1: Variable substitution

```
#!/bin/csh
set opt=-l
set x1='ls $opt'
echo $x1
set x2="ls $opt"
echo $x2
```

```
Will produce the output:
     ls $opt
     ls -l
```

Example 2: Filename generation

```
#!/bin/csh
set ls1='some files: [a-z]*'
echo $ls1

set ls2="some files: [a-z]*"
echo $ls2
```

```
Sample output (identical):
     some files: csh.html images man misc other.materials
     some files: csh.html images man misc other.materials
```

Storing the output of a command

The shell uses backquotes to obtain the output of the command enclosed within the backquotes. This output can be stored within an array variable. Each element can then be indexed and processed as required.

Syntax:

```
set variable = `command`
```

Example:

```
#!/bin/csh
set date_fields=`date`
echo $date_fields
echo $date_fields[1]
echo $date_fields[2]
foreach field(`date`)
  echo $field
end
```

```
Sample output:
     Thu Mar 9 22:25:45 HST 1995
     Thu
     Mar
     Thu
     Mar
     9
     22:25:45
     HST
     1995
```

Reading user input

Depending on your system, you can use either "$<" or the output of the "head -1" command to read stdin into a variable. Note that if you use the "head -1" command, it must be enclosed in backquotes.

Note: Be careful on making sure that you use "$<" and not "<$". The latter case will usually cause your script to fail.

Syntax:

```
set variable = $<
     - or -
set variable = `head -1`
```

Example:

```
#!/bin/csh
echo -n Input your value:
set input = $<
echo You entered: $input

        - or -

#!/bin/csh
echo -n Input your value:
set input = `head -1`
echo You entered: $input
```