L03 – C Shell Scripting - Part 1

1. What is a shell?

So now you've been using Linux operating system for a couple of weeks. Things are a little different here than in the world of Mac or PC that you are likely accustomed to. One major difference is that you are playing around in a terminal, and typing directly into a command line. Getting started in a Linux environment is like going through mouse detox. Instead of clicking our way around, everything happens at the command line of the terminal. But, how are the commands we type interpreted? This depends on the ***shell***, where the shell is just a ***command-line interpreter***. That is, a shell is really just a computer program that reads what you type into the terminal and then interprets what to do with it.

There are a lot of different shells. The most common shells today seem to be the Bourne Again Shell (bash) and the C Shell, but there are some older ones you might encounter such as the Korn shell.

To see which shells are actually available on your system type:

```
>> cat /etc/shells
```

I'm a big fan of the bash shell, and hence, in some nerdy circles am referred to as a ***basher***! Nonetheless, C shell is a very common shell to use in geophysics. This is somewhat historical, the bash shell wasn't written until 1987, long after most geophysicists started a tradition of shell scripting. The C shell was written in the late 1970's and hence has had a longer time to get indoctrinated into the geophysics community. It also turns out to be quite simple to use.

2. What is a shell script?

Normally, when you are sitting at your terminal, the shell is interactive. This means the shell takes the command you type in and then it executes this command. This can be rather tedious if you want to do a larger number of commands in a specific order and maybe do it over and over again on different sets of data. Luckily, we can just write our sequence of commands into a text file, and then tell the shell to run all of the commands in this text file. This text file containing all of our commands is a **shell script**.

Let's make a simple one as an example. Open up a new file named **example.csh** with your favorite text editor and type the following:

```
#!/bin/csh

# the simplest shell script possible

clear
echo "geophysics kicks ass"
```

After creating this file, type the following on the command line:

```
>> chmod +x example.csh
```

This will set the permissions for your new file **example.csh** such that you are allowed to execute it. You only need to do this once for a new file and not after every time you edit it.

Now you can execute the commands in this text file by typing:

```
>> ./example.csh
```

A couple notes on the above script.

Line 1: **#!/bin/csh** - this basically just says that I want to use the C Shell to interpret these commands. Every C Shell script must start out with this as the top-most line.

Line 2: # **the simplest…** - you can add comments, and should frequently, to your scripts if you start the line out with the **#** symbol

Filename: **example.csh** – unlike on a windows machine Linux machines do not require you to have a file extension in most cases. However, it usually makes sense for people to adopt some kind of nomenclature so that you quickly know what kind of file you are dealing with. Hence, I usually use **.csh** to let me know that I have a C Shell script.

OK, now that we have that out of the way, type up the following script and see what it does

```
#!/bin/csh
# Script to print user information who currently login ,
# current date & time

clear
echo "Hello $USER"
echo "Today is \c ";date

echo "Number of user login : \c" ; who | wc -l

echo "Calendar"
cal
```

Note that some versions of C-Shell require you to use **echo –e** so that the \c will not print to the screen.

3. C Shell Variables

There are two types of variables:

(1) System variables – that are created and maintained by the Linux system itself.

We saw one example of these in the example script above: **$USER**. Another example would be if you wanted to print out your home directory then you could type:

```
>> echo $HOME
```

(2) User defined variables – that are created and maintained by the User.

Setting variables in a C Shell script is done in two ways:

(a) String variables. String variables are just treated as a bunch of text characters. i.e., you cannot do math with them. String variables are created with the **set** command as shown below.

```
#!/bin/csh

set x = 1
set y = 10.5
set myvar = super

echo $x $y $myvar
echo $x + $y
```

(b) Numeric variables. The C Shell can only handle integer valued numeric variables. Setting variable names is done with the **@** symbol. Below is a simple example.

```
#!/bin/csh

@ x = 1
@ x = $x + 10
echo $x
```

What happens if you try:

```
set x = $x + 10
```

in the above script?

(c) Arrays of String Variables.

You can also use a single variable name to store an array of strings.

```
#!/bin/csh

set days = (mon tues wed thurs fri)

echo $days
echo $days[3]
echo $days[3-5]
```

As a special note: variables ***are*** case sensitive. For example, the three following combinations of the letters **n** and **o** are all considered to be a different variable by the C Shell. This is important to remember as it is not the case with other programming languages (e.g., in Fortran all three of these variable names would be considered to be the same variable).

```
set no = 10
set No = 11
set nO = 12
```

```
echo $no $No $nO
```

4. Displaying Shell Variables

In case you haven't figured it out by now, we typically use the echo command to display text or the value of a variable when we want to write it out to the screen (writing to the screen is usually called ***writing to standard out***).

Usually, one just types: **`echo $my_variable_name`**

But, in case you want to get fancy, do a man on echo and see what the following examples should produce:

```
#!/bin/csh

set minX = 80

echo "Xaxis Minimum is set to: " $minX

echo "Xaxis Minimum is set to: \a" $minX

echo "Xaxis Minimum is set to: "; echo $minX

echo "Xaxis Minimum is set to: \c"; echo $minX

echo "Xaxis Minimum is set to: \t" $minX

echo "Xaxis Minimum is set to: \\" $minX
```

It is also prudent at this point to consider the action of different types of quotes. There are three types of quotes

Quotes	Name	Meaning
`"`	Double Quotes	"Double Quotes" - Anything enclosed in double quotes removes the meaning of the characters (except \ and $). For example, if we **`set arg = blah`**, then **`echo "$arg"`** would result in **blah** being printed to the screen.
`'`	Single quotes	'Single quotes' – Text enclosed inside single quotes remains unchanged (including $variables). For example, **`echo '$arg'`** would result in **`$arg`** being printed to the screen. That is, no variable substitution would take place.
`` ` ``	Back quote	`Back quote` - To execute a command. For example, **`` `pwd` ``** would execute the print working directory command.

To see the effect of the single or double quote add the following to the above script:

```
echo "$minX"
```

```
echo '$minX'
```

The back quote is really useful. This allows us to set a shell variable to the output from a Unix command:

```
#!/bin/csh

set mydir = `pwd`  # set variable to current working directory

@ nr = `awk 'END {print NR}' input_file`    # what does this do?

@ nfiles = `ls *UU* | wc -l`
```

As a final note on displaying shell variables it is often useful to concatenate shell variables:

```
#!/bin/csh

set year    = 2010
set month   = 12
set day     = 30

set output1 = ${year}_${month}_${day}
set output2 = ${year}${month}${day}

echo $output1
echo $output2

mv inputfile ${output1}.txt
```

Note that we use the **{ }** brackets in this example. This is because if I just type **$year_** then the shell would look for a variable called **year_**.

5. Command Line Arguments

It is often useful to be able to grab input from the command line or to read user input. The next example shows a simple way to interactively get information and set the result to a variable.

```
#!/bin/csh

echo "How many records in this file do you want to skip? "
set nlines = $<

echo $nlines
```

To see how command line arguments are handled let's consider the following example where I want to read in a filename and then perhaps do some action on this file later.

```
#!/bin/csh

set ifile = $argv[1]

echo "Now lets perform some kind of action on file: $ifile"
```

If I named this C Shell script: **action.csh**

and we want to perform the action on the file **foo.txt**

then we need to type:

```
>> action.csh foo.txt
```

on the command line to make this work. This is really useful when we want to make generalized scripts that don't require editing the variable names every time we want them to run.

6. Redirection of standard output/input

The input and output of commands can be sent to or received from files using redirection. Some examples are shown below:

```
date > datefile
```

The output of the **date** command is saved into the contents of the file, **datefile**.

```
a.out < inputfile
```

The program, **a.out** receives its input from the input file, **inputfile**.

```
sort gradefile >> datafile
```

The **sort** command returns its output and appends it to the file, **datafile**.

A special form of redirection is used in shell scripts.

```
calculate << END_OF_FILE
...
...
END_OF_FILE
```

In this form, the input is taken from the current file (usually the shell script file) until the string following the << is found. An example of using the program SAC (Seismic Analysis Code) is shown below (it is becoming more and more of a rarity for people to write SAC macros!):

```
#!/bin/csh

sac << EOF
r infile.sac
qdp off
ppk
q
EOF
```

If the special variable, **noclobber** is set, any redirection operation that will overwrite an existing file will generate an error message and the redirection will fail. In order to force an overwrite of an existing file using redirection, append an exclamation point (!) after the redirection command. For example for the command:

```
date >! datefile
```

The file **datefile** will be overwritten regardless of its existence.

The output of one command can be sent to the input of another command. This is called piping. The commands which are to be piped together are separated by the pipe character. For example:

```
ls -l | sort -k 5n
```

This command takes the output of the **ls -l** command and puts the output of it into the **sort** command.

7. Homework

1) Write a C Shell script that will allow you to set the name of an input postscript file and desired output name of a jpg file, and then use ImageMagick's convert command to convert a postscript file into a jpeg image. E.g., At the very least I should enter, either by the command line or by interactive input the name of an input .ps file, and desired name of output .jpg file and the script will automatically create the .jpg file.

2) Write a C Shell script that will add the current date to the end of a filename. E.g., if today is Dec 25, 2010, then the shell script should change the ***filename*** to:

filename.20101225

The script should read the filename from the command line. Hence, if we named this script adddate then execution of this command should look like:

```
>> addate filename
```

3) Write a C Shell script that will remove dates added with the script written in Problem #2. Note: this script should also work when there is a dot in the filename. E.g., the code should work for any filename of the form…

```
foo.20101225
foo.foo.20101225
foo.foo.foo.20101225
foo.foo.foo.*.20101225
```

Output file names for the examples above should be:

```
foo
foo.foo
foo.foo.foo
etc.
```

4) Write a script that will replace spaces in file names with underscores. E.g., if the input file is named: My File.txt , then the output file should be named My_File.txt.