University of Minnesota Supercomputer Institute

Introduction to C-Shell Programming

This tutorial written by Jennifer Dudley was originally located at:
http://www.msi.umn.edu/~dudley/cshell/csh.html
but has since been removed.

Some minor modifications for the University of Melbourne have been made by:

Kevin Keay   Apr 21 2008**.**

Written By Jennifer Dudley 7-16-1996

Makeup of a shell script

- Shell designator `#!..`
- Shell flags
- C-shell builtin commands (`echo,cd,set...`)
- UNIX executables
- Comments (`#`)

Example:

```
#!/bin/csh -f
# example: This is an example of a basic shell script. It contains all of
the basic elements
# The first line must have the shell designator; in this case csh.
# Lines preceded with a # (hash sign) are comments and ignored.
# Some builtin C-shell commands used in this case are: exit, set, and
while.
#
# The -f flag after the designator tells the the script not to read the
users .cshrc.
# This makes the program more portable to others as it will not depend
# on any of the creators aliases.
#
if ( $#argv !=2 ) then # Script needs exactly 2 arguments


     echo "Incorrect number of arguments"

     exit 1


set start=$1
set end=$2

@ i = start
while ( $i <= $end )


    a.out < input.$i > output.$i

    @ i= $i + 1


end
```

Creating a shell script

Shell scripts are ordinary text files created by your favorite text editor like `nedit`, `vi` or `emacs`.

```
% nedit my_script
```

The second step in creating shell scripts is to set the permissions on the file using **`chmod`**. In order to execute a shell script, the executable bit must be set. For example:

```
% ls -al my_script
```

```
-rw-------    myself    452        my_script
```

```
% chmod 755 my_script
```

755 allows execution access to everybody, 700 just the user. Alternatively use the plus-minus notation e.g. **`chmod u+rwx g+rx o+rx my_script`**, where u,g,o refer to user (you), group (normally colleagues in Meteorology) and others.

```
% ls -al my_script
```

```
-rwx------    myself    452        my_script
```

The x in the fourth field signifies that the script is now executable.

Executing shell scripts

A shell script is executed by the same means that you would execute any other UNIX command or binary. There are two ways to do this:

1) Type the full path to the script
`% /home/user/userdir/my_script` e.g. /home/kevin/bin/my_script

2) If the script resides in your path then only the file name need be typed.
`% my_script`

By default a shell script will direct the output to the "standard output". You may however redirect the output to a file or through a pipeline of commands.

Example "hello world":

```
% set path = ($path        # Includes your current
.)                         directory to your path variable

% nedit hello              # Create the script file

#!/bin/csh -f
echo "Hello, World"        # Type the script

% chmod 755 hello          # Make the script executable

% hello                    # Execute the script

                           # Standard output (after
Hello, World               execution)

% hello > output.txt       # Send the output to a file

% cat output.txt           # edit the script

Hello, World

% hello | rev              # Pipe the output through rev

dlroW ,olleH
```

Written By Jennifer Dudley 7-16-1996 *University of Minnesota Supercomputer Institute*

Interacting with the shell-script

The examples we have seen so far have been **non-interactive** . This means that they take no input from the user. A script is executed and, barring any bugs, will run to completion.

An interactive script can take input from several sources.
For example:

- Keystrokes from a terminal
- Redirected from a file
- Read from a pipeline of commands

% nedit read_name

```
#!/bin/csh -f

echo "Please enter your name: "

set name = $<

echo "Hello, $name"
```

```
% chmod 755 read_name
```

Input using interactive typing:

```
% read_name
```
Please enter your name:
Jen
Hello, Jen

Input redirected from a file:

```
% echo "Jen" > input
% read_name < input
```
Please enter your name:
Hello, Jen

Input through a pipeline:

```
% cat input | read_name
Please enter your name
Hello, Jen
```

Written By Jennifer Dudley 7-16-1996 *University of Minnesota Supercomputer Institute*

Shell Programming

- Shell variables
- Argument passing
- Looping in shell-scripts
- Branching using `if/then/else`
- Debugging shell scripts

Written By Jennifer Dudley 7-16-1996 *University of Minnesota Supercomputer Institute*

Shell variables

There are two types of variables available to the C-shell programmer: the string (text) variable and the numerical variable.

The syntax for setting a C-shell string variable is "`set name=value`" e.g.:
`% set inputfile = my_input`

The syntax for setting a *numerical* variable is "`@ name=value`" e.g.:
`@ i = 5`

However, **all** variables are then referenced with a dollar sign "`$`".

Example var-test:

```
% nedit var-test

#!/bin/csh -f
set my_string1="Cats have"
set my_string2="nine lives"
@ x = 9
echo "$my_string1 $x $my_string2"

% chmod 755 var_test
% var_test
```
Cats have nine lives

Written By Jennifer Dudley 7-16-1996 *University of Minnesota Supercomputer Institute*

Argument passing

Arguments can be passed to your shell script. These arguments are known as "special variables" `$0, $1, $2, ... $n`. The special variable `$*` is used to list all variables.

Example arg_pass:

```
%nedit arg_pass


     #!/bin/csh -f

     echo "This script is called $0"

     echo "The first argument is $1"

     echo "The second argument is $2"

     echo "All of the arguments are $*"


% chmod 755 arg_pass
% arg_pass right now
```

This script is called arg_pass
The first argument is right
The second argument is now
All of the arguments are right now

Looping in shell scripts

Using while

If you have a large number of files similarly named (input.1, input.2, input.3, ... input.n) and all of these files are to be processed the same and the output written to corresponding output files (outfile.1, outfile.2, outfile.3, ... outfile.n). The task could get very tedious and the chances for error are tremendous.

This is where a shell script with a **while** loop would be extremely useful.

Example while_loop:

```
% nedit while_loop

#!/bin/csh -f
# set x = 1
@ x = 1
while ($x <= $1)      #  $1 being the number of files specified at the command line
  a.out outfile.$x    #  Process the file through a.out (compiled from a Fortran or C program)
  @ x = $x + 1
end
```

Written By Jennifer Dudley 7-16-1996 *University of Minnesota Supercomputer Institute*

Branching using if/then/else

Since there are many different UNIX systems in use today, it can be difficult to remember the particular commands and options for each kind. A lot of people these days have multiple accounts so it could be a great help to create a shell script to take the guess work out of what commands to use.

Example myps:

```
% nedit myps
```

Type in the following text:

```
#!/bin/csh -f
# This program will test the operating system for Linux or
# SunOS and will give the appropriate options and will grep
# for for the string given in the first argument.

set OS = ‘uname`

if ($OS == "Linux") then

      set ps_command="ps -edf"

else if ($OS == "SunOS") then # Note: else if NOT elseif

      set ps_command="ps -aux"

else

      echo "Unknown Operating System”

endif

echo $ps_command | grep $1
```

Important note: Use `else if` *not* `elseif`. The latter results in the rest of the `if` block being ignored.

```
% chmod 755 myps
% myps dudley

dudley   19507   19481   3      17:03:17 pts/2   0:00   grep dudley
dudley   19432   19480   63     17:02:59 pts/2   0:02   -tsch
dudley   19506   19481   5      17:03:17 pts/2   0:00   ps -edf
```

As you can see this is much more powerful than "alias"

Debugging shell scripts

So far we have looked at the `-f` flag for the descriptor line. Another useful option is the `-x` flag which can be used as a debugging tool. It echoes the shell script commands as they are executing.

Example endless

```
% nedit endless

#!/bin/csh -f -x
@ i = 1
while ($i < $1)

  a.out outfile.$i

end

% chmod 755 endless
% endless 10
while (1 < 10)

 a.out

while (1 < 10)

 a.out

while (1 < 10)

 a.out

while (1 < 10)

 a.out

.
.
.
```

In this case we forgot to increment the variable x