C shell

Programmers who are used to C or C++ often find it easier to program in C-shell because there are strong similarities between the two.

.cshrc and .login files

Most users run the C-shell `` `/bin/csh' `` as their login environment, or these days, preferably the `` `tcsh' `` which is an improved version of csh. When a user logs in to a UNIX system the C-shell starts by reading some files which configure the environment by defining variables like `path`.

- The file `` `.cshrc' `` is searched for in your home directory. i.e. `` `~/.cshrc' ``. If it is found, its contents are interpreted by the C-shell as C-shell instructions, before giving you the command prompt.
- If and only if this is the *login shell* (not a sub-shell that you have started after login) then the file `` `~/.login' `` is searched for and executed.

With the advent of the X11 windowing system, this has changed slightly. Since the window system takes over the entire login procedure, users never get to run `login shells', since the login shell is used up by the X11 system. On an X-terminal or host running X the `` `.login' `` file normally has no effect.

With some thought, the `` `.login' `` file can be eliminated entirely, and we can put everything into the `.cshrc` file. Here is a very simple example `` `.cshrc' `` file.

```
#
# .cshrc - read in by every csh that starts.
#

# Set the default file creation mask
umask 077

# Set the path
set path=( /usr/local/bin /usr/bin/X11 /usr/ucb /bin /usr/bin . )

# Exit here if the shell is not interactive
if ( $?prompt == 0 ) exit

# Set some variables

set noclobber notify filec nobeep
set history=100
set prompt="`hostname`%"
set prompt2 = "%m %h>"    # tcsh, prompt for foreach and while

setenv PRINTER myprinter
setenv LD_LIBRARY_PATH /usr/lib:/usr/local/lib:/usr/openwin/lib

# Aliases are shortcuts to UNIX commands

alias passwd  yppasswd
alias dir     'ls -lg \!* | more'
alias sys     'ps aux | more'
alias h       history
```

It is possible to make a much more complicated .cshrc file than this. The advent of distributed computing and NFS (Network file system) means that you might log into many different machines running different versions of UNIX. The command path would have to be set differently for each type of machine.

Defining variables with set, setenv

We have already seen in the examples above how to define variables in C-shell. Let's formalize this. To define a local variable -- that is, one which will not get passed on to programs and sub-shells running under the current shell, we write

```
set local = "some string"
set myname = "`whoami`"
```

These variables are then referred to by using the dollar `` `$' `` symbol. i.e. The value of the variable `` `local' `` is `` `$local' ``.

```
echo $local $myname
```

Global variables, that is variables which all sub-shells inherit from the current shell are defined using `` `setenv' ``

```
setenv GLOBAL "Some other string"
setenv MYNAME "`who am i`"
```

Their values are also referred to using the `` `$' `` symbol. Notice that `set` uses an `` `=' `` sign while `` `setenv' `` does not.

Variables can be also created without a value. The shell uses this method to switch on and off certain features, using variables like `` `noclobber' `` and `` `noglob' ``. For instance

```
nexus% set flag
nexus% if ($?flag) echo 'Flag is set!'
Flag is set!
nexus% unset flag
nexus% if ( $?flag ) echo 'Flag is set!'
nexus%
```

The operator `` `$?variable' `` is `` `true' `` if *variable* exists and `` `false' `` if it does not. It does not matter whether the variable holds any information.

The commands `` `unset' `` and `` `unsetenv' `` can be used to undefine or delete variables when you don't want them anymore.

Arrays

A useful facility in the C-shell is the ability to make arrays out of strings and other variables. The round parentheses `` `(..)' `` do this. For example, look at the following commands.

```
nexus% set array = ( a b c d )
nexus% echo $array[1]
a
nexus% echo $array[2]
b
nexus% echo $array[$#array]
d

nexus% set noarray = ( "a b c d" )
nexus% echo $noarray[1]
a b c d
nexus% echo $noarray[$#noarray]
a b c d
```

The first command defines an array containing the elements `` `a b c d' ``. The elements of the array are referred to using square brackets `` `[..]' `` and the first element is `` `$array[1]' ``. The last element is `` `$array[4]' ``. *NOTE: this is not the same as in C or C++ where the first element of the array is the zeroth element!*

The special operator `$#' returns the number of elements in an array. This gives us a simple way of finding the end of the array. For example

```
nexus% echo $#path
23

nexus% echo "The last element in path is $path[$#path]"
The last element in path is .
```

To find the next last element we need to be able to do arithmetic. We'll come back to this later.

Pipes and redirection in csh

The symbols

```
               <  >  >>  <<  |  &
```

have a special meaning in the shell. By default, most commands take their input from the file `stdin' (the keyboard) and write their output to the file `stdout' and their error messages to the file `stderr' (normally, both of these output files are defined to be the current terminal device `/dev/tty', or `/dev/console').

`stdin', `stdout' and `stderr', known collectively as `stdio', can be redefined or *redirected* so that information is taken from or sent to a different file. The output direction can be changed with the symbol `>'. For example,

```
echo testing > myfile
```

produces a file called `myfile' which contains the string `testing'. The single `>' (greater than) sign always creates a new file, whereas the double `>>' appends to the end of a file, if it already exists. So the first of the commands

```
echo blah blah >> myfile
echo Newfile > myfile
```

adds a second line to `myfile' after `testing', whereas the second command writes over `myfile' and ends up with just one line `Newfile'.

Now suppose we mistype a command

```
ehco test > myfile
```

The command `ehco' does not exist and so the error message `ehco: Command not found' appears on the terminal. This error message was sent to *stderr* -- so even though we redirected output to a file, the error message appeared on the screen to tell us that an error occurred. Even this can be changed. `stderr' can also be redirected by adding an ampersand `&' character to the `>' symbol. The command

```
ehco test >& myfile
```

results in the file `myfile' being created, containing the error message `ehco: Command not found'.

The input direction can be changed using the `<' symbol for example

```
/bin/mail mark < message
```

would send the file `message' to the user `mark' by electronic mail. The mail program takes its input from the file instead of waiting for keyboard input.

There are some refinements to the redirection symbols. First of all, let us introduce the C-shell variable `noclobber'. If this variable is set with a command like

```
set noclobber
```

then files will not be overwritten by the `` `>' `` command. If one tries to redirect output to an existing file, the following happens.

```
UNIX% set noclobber
UNIX% touch blah        # create an empty file blah
UNIX% echo test > blah
blah: File exists.
```

If you are nervous about overwriting files, then you can set `` `noclobber' `` in your `` `.cshrc' `` file. `` `noclobber' `` can be overridden using the pling `` `!' `` symbol. So

```
UNIX% set noclobber
UNIX% touch blah        # create an empty file blah
UNIX% echo test >! blah
```

writes over the file `` `blah' `` even though `` `noclobber' `` is set.

Here are some other combinations of redirection symbols

`` `>>' ``
: Append, including `` `stderr' ``

`` `>>!' ``
: Append, ignoring `` `noclobber' ``

`` `>>&!' ``
: Append `` `stdout' ``, `` `stderr' ``, ignore `` `noclobber' ``

`` `<<' ``
: See below.

The last of these commands reads from the standard input until it finds a line which contains a word. It then feeds all of this input into the program concerned. For example,

```
nexus% mail mark <<quit
nexus 1> Hello mark
nexus 2> Nothing much to say...
nexus 2> so bye
nexus 2>
nexus 2> quit
Sending mail...
Mail sent!
```

The mail message contains all the lines up to, but not including `marker'. This method can also be used to print text verbatim from a file without using multiple echo commands. Inside a script one may write:

```
cat << "marker";

           MENU

    1) choice 1
    2) choice 2
    ...

marker
```

The `cat` command writes directly to `stdout` and the input is redirected and taken directly from the script file.

A very useful construction is the `pipe' facility. Using the `` `|' `` symbol one can feed the `` `stdout' `` of one program straight into the `` `stdin' `` of another program. Similarly with `` `|&' `` both `` `stdout' `` and `` `stderr' `` can be piped into the input of another program. This is very convenient. For instance, look up the following commands in the manual and try them.

```
ps aux | more
echo 'Keep on sharpening them there knives!' | mail henry
```

```
vmstat 1 | head
ls -l /etc | tail
```

Note that when piping both standard input and standard error to another program, the two files *do not mix synchronously*. Often `` `stderr' `` appears first.

Scripts with arguments

One of the useful features of the shell is that you can use the normal UNIX commands to make programs called *scripts*. To make a script, you just create a file containing shell commands you want to execute and make sure that the first line of the file looks like the following example.

```
#!/bin/csh -f
#
# A simple script: check for user's mail
#
#

set path = ( /bin /usr/ucb )              # Set the local path
cd /var/spool/mail                        # Change dir
foreach uid ( * )
   echo "$uid has mail in the intray! "    # space prevents an error!
end
```

The sequence `` `#!/bin/csh' `` means that the following commands are to be fed into `` `/bin/csh' ``. The two symbols `` `#!' `` must be the very first two characters in the file. The `` `-f' `` option means that your `` `.cshrc' `` file is not read by the shell when it starts up. The file containing this script must be executable (see `` `chmod' ``) and must be in the current path, like all other programs.

Like C programs, C-shell scripts can accept command line arguments. Suppose you want to make a program to say hello to some other users who are logged onto the system.

```
say-hello mark sarah mel
```

To do this you need to know the names that were typed on the command line. These names are copied into an array in the C-shell called the *argument vector*, or `` `argv' ``. To read these arguments, you just treat `` `argv' `` as an array.

```
#!/bin/csh -f
#
# Say hello
#

foreach name ( $argv )
   echo Saying hello to $name
   echo "Hello from $user! " | write $name
end
```

The elements of the array can be referred to as `` `argv[1]' ``..`` `argv[$#argv]' `` as usual. They can also be referred to as `` `$1' ``..`` `$3' `` up to the last acceptable number. This makes C-shell compatible with the Bourne shell as far as arguments are concerned. One extra flourish in this method is that you can also refer to the name of the program itself as `` `$0' ``. For example,

```
#!/bin/csh -f

echo This is program $0 running for $user
```

`` `$argv' `` represents all the arguments. You can also use `` `$*' `` from the Bourne shell.

Tests and conditions

No programming language would be complete without tests and loops. C-shell has two kinds of decision structure: the `` `if..then..else' `` and the `` `switch' `` structure. These are closely related to their C counterparts. The syntax of these is

```
if (condition) command

if (condition) then
   command
   command..
else
   command
   command..
endif


switch (string)

  case one:
             commands
             breaksw

  case two:
             commands
             breaksw
  ...

endsw
```

In the latter case, no commands should appear on the same line as a `case' statement, or they will be ignored. Also, if the `` `breaksw' `` commands are omitted, then control flows through all the commands for case 2, case 3 etc, exactly as it does in the C programming language.

We shall consider some examples of these statements in a moment, but first it is worth listing some important tests which can be used in `` `if' `` questions to find out information about files.

`` `-r *file*' ``
: True if the file exists and is readable

`` `-w *file*' ``
: True if the file exists and is writable

`` `-x *file*' ``
: True if the file exists and is executable

`` `-e *file*' ``
: True if the file simply exists

`` `-z *file*' ``
: True if the file exists and is empty

`` `-f *file*' ``
: True if the file is a plain file

`` `-d *file*' ``
: True if the file is a directory

We shall also have need of the following comparison operators.

`` `==' ``
: is equal to (string comparison)

`` `!=' ``
: is not equal to

`` `>' ``
: is greater than

`<'
: is less than

`>='
: is greater than or equal to

`<='
: is less than or equal to

`=~'
: matches a wildcard

`!~'
: does not match a wildcard

The simplest way to learn about these statements is to use them, so we shall now look at some examples.

```
#!/bin/csh -f
#
#  Safe copy from <arg[1]> to <arg[2]>
#
#

if ($#argv != 2) then

  echo "Syntax: copy <from-file> <to-file>"
  exit 0

endif

if ( -f $argv[2] ) then
   echo "File exists. Copy anyway?"
   switch ( $< )                     # Get a line from user
      case y:
              breaksw
      default:
               echo "Doing nothing!"
               exit 0
   endsw
endif

echo -n "Copying $argv[1] to $argv[2]..."
cp $argv[1] $argv[2]
echo done

endif
```

This script tries to copy a file from one location to another. If the user does not type exactly two arguments, the script quits with a message about the correct syntax. Otherwise it tests to see whether a plain file has the same name as the file the user wanted to copy to. If such a file exists, it asks the user if he/she wants to continue before proceeding to copy.

Loops in csh

The C-shell has three loop structures: `` `repeat' ``, `` `while' `` and `` `foreach' ``. We have already seen some examples of the `` `foreach' `` loop.

The structure of these loops is as follows

```
repeat number-of-times command

while ( test expression )
   commands
end

foreach  control-variable  ( list-or-array )
   commands
```

```
end
```

The commands `` `break' `` and `` `continue' `` can be used to break out of the loops at any time. Here are some examples.

```
repeat 2 echo "Yo!" | write mark
```

This sends the message "Yo!" to mark's terminal twice.

```
repeat 5 echo `echo "Shutdown time! Log out now" | wall ; sleep 30` ; halt
```

This example repeats the command `echo Shutdown time...' five times at 30 second intervals, before shutting down the system. Only the superuser can run this command! Note the strange construction with `echo echo'. This is to force the repeat command to take two shell commands as an argument. (Try to explain why this works for yourself.)

Input from the user

```
# Test a user response
echo "Answer y/n (yes or no)"
set valid = false
while ( $valid == false )
   switch ( $< )
      case y:
              echo "You answered yes"
              set valid = true
              breaksw
      case n:
              echo "You answered no"
              set valid = true
              breaksw
      default:
              echo "Invalid response, try again"
              breaksw
   endsw
end
```

Notice that it would have been simpler to replace the two lines

```
  set valid = true
  breaksw
```

by a single line `` `break' ``. `` `breaksw' `` jumps out of the switch construction, after which the `` `while' `` test fails. `` `break' `` jumps out of the entire while loop.

Arithmetic

Before using these features in a real script, we need one more possibility: numerical addition, subtraction and multiplication etc.

To tell the C-shell that you want to perform an operation on numbers rather than strings, you use the `` `@' `` symbol followed by a space. Then the following operations are possible.

```
@ var = 45                      # Assign a numerical value to var
echo $var                       # Print the value

@ var = $var + 34               # Add 34 to var
@ var += 34                     # Add 34 to var
@ var -= 1                      # subtract 1 from var
@ var *= 5                      # Multiply var by 5

@ var /= 3                      # Divide var by 3 (integer division)
@ var %= 3                      # Remainder after dividing var by 3

@ var++                         # Increment var by 1
@ var--                         # Decrement var by 1

@ array[1] = 5                  # Numerical array

@ logic = ( $x > 6 && $x < 10)  # AND
@ logic = ( $x > 6 || $x < 10)  # OR
@ false = ! $var                # Logical NOT

@ bits = ( $x | $y )            # Bitwise OR
@ bits = ( $x ^ $y )            # Bitwise XOR
@ bits = ( $x & $y )            # Bitwise AND

@ shifted = ( $var >> 2 )       # Bitwise shift right
@ back    = ( $var << 2 )       # Bitwise shift left
```

These operators are precisely those found in the C programming language.

Examples

The following script uses the operators in the last two sections to take a list of files with a given file extension (say `` `.doc' ``) and change it for another (say `` `.tex' ``). This is a partial solution to the limitation of not being able to do multiple renames in shell.

```
#!/bin/csh -f
#############################################################
#
# Change file extension for multiple files
#
#############################################################

if ($#argv < 2) then
  echo Syntax: chext oldpattern newextension
  echo "e.g: chext *.doc tex "
  exit 0
endif

mkdir /tmp/chext.$user                 # Make a scratch area

set newext="$argv[$#argv]"             # Last arg is new ext
set oldext="$argv[1]:e"

echo "Old extension was ($oldext)""
echo "New extension ($newext) -- okay? (y/n)"

switch( $< )

   case y:
           breaksw
   default:
           echo "Nothing done."
           exit 0
endsw

##############################################################
# Remove the last file extension from files
##############################################################
```

```
 i = 0

foreach file ($argv)

   i++
  if ( $i == $#argv ) break
  cp $file /tmp/chext.$user/$file:r        # temporary store

end

###############################################################
# Add .newext file extension to files
###############################################################

set array = (`ls /tmp/chext.$user`)

foreach file ($array)

  if ( -f $file.$newext ) then
    echo  destination file $file.$newext exists. No action taken.
    continue
  endif

  cp /tmp/chext.$user/$file $file.$newext
  rm $file.$oldext

end

rm -r /tmp/chext.$user
```

Here is another example to try to decipher. Use the manual pages to find out about `awk'. This script can be written much more easily in Perl or C, as we shall see in the next chapters. It is also trivially implemented as a script in the system administration language cfengine.

```
#!/bin/csh -f
###########################################################
#
# KILL all processes owned by $argv[1] with PID > $argv[2]
#
###########################################################

if ("`whoami`" != "root") then
  echo Permission denied
  exit 0
endif

if ( $#argv < 1 || $#argv > 2 ) then
  echo Usage: KILL username lowest-pid
  exit 0
endif

if ( $argv[1] == "root") then
  echo No! Too dangerous -- system will crash
  exit 0
endif

############################################################
# Kill everything
############################################################

if ( $#argv == 1 ) then

  set killarray = ( `ps aux |  awk '{ if ($1 == user) \
{printf "%s ",$2}}' user=$argv[1]` )

  foreach process ($killarray)

     kill -1 $process
     kill -15 $process > /dev/null
     kill -9 $process > /dev/null

     if ("`kill -9 $process | egrep -e 'No such process'`" == "") then
        echo "Warning - $process would not die - try again"
     endif
  end

#############################################################
# Start from a certain PID
#############################################################

else if ( $#argv == 2 ) then

  set killarray = ( `ps aux |  awk '{ if ($1 == user && $2 > uid) \
{printf "%s ",$2}}' user=$argv[1] uid=$argv[2]` )

  foreach process ($killarray)

     kill -1 $process > /dev/null
     kill -15 $process
     sleep 2
     kill -9 $process > /dev/null

     if ("`kill -9 $process | egrep -e 'No such process'`" == "") then
        echo "Warning - $process would not die - try again"
     endif
  end

endif
```

Summary: Limitations of shell programming

To summarize the last two long and oppressive chapters we shall take a step back from the details and look at what we have achieved.

The idea behind the shell is to provide a user interface, with access to the system's facilities at a simple level. In the 70's user interfaces were not designed to be user-friendly. The UNIX shell is not particularly use friendly, but it is very powerful. Perhaps it would have been enough to provide only commands to allow users to write C programs. Since all of the system functions are available from C, that would certainly allow everyone to do what anything that UNIX can do. But shell programming is much more *immediate* than C. It is an environment of *frequently used tools*. Also for quick programming solutions: C is a compiled language, whereas the shell is an interpreter. A quick shell program can solve many problems in no time at all, without having to compile anything.

Shell programming is only useful for `quick and easy' programs. To use it for anything serious is an abuse. Programming difficult things in shell is clumsy, and it is difficult to get returned-information (like error messages) back in a useful form. Besides, shell scripts are slow compared to real programs since they involve starting a new program for each new command.

These difficulties are solved partly by Perl, which we shall consider next -- but in the final analysis, real programs of substance need to be written in C. Contrary to popular belief, this is not more difficult than programming in the shell -- in fact, many things are much simpler, because all of the shell commands originated as C functions. The shell is an extra layer of the UNIX onion which we have to battle our way through to get where we're going.

Sometimes it is helpful to be shielded from *low level* details -- sometimes it is a *hindrance*. In the remaining chapters we shall consider more involved programming needs.