The C Shell tutorial

Taken from **http://www.eng.hawaii.edu/Tutor/csh.html on August 7**, 2006

What is a shell?

A shell is a program which provides a user interface. With a shell, users can type in commands and run programs on a Unix system. Basically, the main function a shell performs is to read in from the terminal what one types, run the commands, and show the output of the commands.

What's so good about C Shell?

The C shell was written by Bill Joy at the University of California at Berkeley. His main intent for writing the C shell was to create a shell with C language-like syntax.

What can one do with C Shell?

The main use of the C shell is as an interactive shell, but one can write programs using the C shell. These programs are called shell scripts.

Features of C Shell

Some of the features of the C shell are listed here:

- Customizable environment.
- Abbreviate commands. (Aliases.)
- History. (Remembers commands typed before.)
- Job control. (Run programs in the background or foreground.)
- Shell scripting. (One can write programs using the shell.)
- Keyboard shortcuts.

Features of the shell environment

The C shell provides programming features listed below:

- Control constructs. (For example, loop and conditional statements.)
- File permissions/existence checking.
- Variable assignment.
- Built-in Variables.

Files for the C Shell environment customization

The C shell has three separate files which are used for customizing its environment. These three files are *.cshrc*, *.login*, and *.logout*. Because these files begin with a period (.) they do not usually appear when one types the **ls** command. In order to see all files beginning with periods, the **-a** option is used with the **ls** command.

The *.cshrc* file contains commands, variable definitions and aliases used any time the C shell is run. When one logs in, the C shell starts by reading the *.cshrc* file, and sets up any variables and aliases.

The C shell reads the *.login* file after it has read the *.cshrc* file. This file is read once only for login shells. This file should be used to set up terminal settings, for example, backspace, suspend, and interrupt characters.

The *.logout* file contains commands that are run when the user logs out of the system.

Sample *.cshrc* file

```
#!/bin/csh
# Sample .cshrc file
setenv EXINIT 'set smd sw=4 wm=2'
set history=50
set savehist=50
set ignoreeof noclobber
if ($?prompt) then
  set prompt='[\!]% '
  alias f finger -R
  alias lo logout
endif
```

Sample *.login* file

```
#!/bin/csh
# Sample .login file
stty erase ^H intr ^C susp ^Z
echo "Welcome to Wiliki\!"
frm -s n
```

Sample *.logout* file

```
#!/bin/csh
# Sample .logout file
echo -n "Logged out of Wiliki "
date
```

Special characters in C Shell

Some characters are special to the shell, and in order to enter them, one has to precede it with a backslash (\). Some are listed here with their meaning to the shell.

!	History substitution.
< >	Output redirection.
|	Pipes.
*	Matches any string of zero or more characters.
?	Matches any single character.
[ ]	Matches any set of characters contained in brackets.
{ }	Matches any comma-separated list of words.
;	Used to separate commands.
&	Also used to separate commands, but puts them in the background.
\	Quote the following character.
$	Obtains the value of the variable.
'	Take text enclosed within quotes literally.
`	Take text enclosed within quotes as a command, and replace with output.
"	Take text enclosed within quotes literally, after substituting any variables.

Variables

Variables in C shell are defined using the internal **set** command. C shell supports both regular and array variables. Some examples are given below:

```
set var1=a3 #sets var1's value to a3.
set var2=(a b c)
# sets the array variable var2 to a b, and c.
```

Using variables

Variables can be used in C shell by typing a dollar sign ($) before the variable name. If the variable is an array, the subscript can be specified using brackets, and the number of elements can be obtained using the form `$#var2`.

The existence of variables can be checked using the form `$?variable`. If the variable exists, the expression evaluates to a one (true), otherwise, it evaluates to a zero (false). Simple integer calculations can be performed by C shell, using C language-type operators. To assign a calculated value, the **@** command is used as follows:

```
@ var = $a + $x * $z
```

Built-in shell variables

Certain variables control the behavior of the C shell, and some of these don't require a value. (I.e., can be set simply by using **set** command by itself without any value.) The **unset** command can be used to unset any undesirable variables.

`argv`	Special variable used in shell scripts to hold the value of arguments.
`autologout`	Contains the number of minutes the shell can be idle before it automatically logs out.
`history`	Sets how many lines of history (previous commands) to remember.
`ignoreeof`	Prevents logging out with a *control-D*.
`noclobber`	Prevents overwriting of files when using redirection.
`path`	Contains a list of directories to be searched when running programs or shell scripts.
`prompt`	Sets the prompt string.
`term`	Contains the current terminal type.

History

If the `history` variable is set to a numerical value, that many commands typed previous would be remembered in a history list. Commands from the history are numbered from the first command being 1. To see the history, the **history** command is used.

Commands from the history can be recalled using the exclamation point. For example, `!!` repeats the previous command, `!25` re-types command number 25 from the history, and `!-2` re-types the second line previous to the current line.

Individual words from these command lines can also be retrieved using this history. For example, `!25:$` returns the last argument (word) from command 25, `!!:*` returns all the arguments (all words but the first one) from the last command, and `!-2:0` returns the command (the first word) of the second line previous.

Aliasing

A shorthand can be assigned to a command or sequence of commands which are frequently used. By assigning an alias with the **alias** command, one can essentially create their own commands, and even "overwrite" existing commands. For example:

```
alias cc cc -Aa -D_HPUX_SOURCE
```

This alias definition will substitute the **cc** with the ANSI compiler option on an HP System (such as *Wiliki*) whenever `cc` is typed. To undefine an alias, the **unalias** command is used.

If the filenames used behind an alias must come before text being substituted, history substitution can be used, as follows:

```
alias manl 'man \!* | less -p'
```

This form of the command places the arguments placed after the `manl` alias between the **man** command and the | (pipe).

Input/Output Redirection

The input and output of commands can be sent to or gotten 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.

If the special variable, `noclobber` is set, if any redirection operation will overwrite an existing file, an error message is given 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.

Adding an ampersand (&) to the end of an output redirection command will combine both the standard error and the standard output and place the output into the specified file.

Pipes

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.

By appending an ampersand (&) after the pipe character, one can combine the standard error and standard output and send it to the standard input of the program receiving the piped output.

Job control

The C shell handles job control, which allows the user to run certain programs in the background, and recall them to the foreground when necessary. In order to place a running process into the background, the suspend character must be set by the stty command shown earlier. Processes may be started in the background by following the command with an ampersand (&).

When a job is placed in the background, information for the job is shown similar to the example given below:

```
[1] 15934
```

This specifies that the process has been placed in the background, and is job 1. In order to recall jobs placed in the background, the **fg** command is used, while the **bg** command places a recently stopped process into the background. The **jobs** command gives a list of all processes under control of the current shell. Also, typing a percent sign (%) with the job number brings that particular job to the foreground.

Control structures

The C shell has control structures similar to the C programming language. These are **foreach**, **if**, **switch** and **while**. These are usually used in shell scripts.

There are two forms of the **if** statement. The first one has a simple command after the expression. This simple command cannot be an alias, nor can it use statements that use the backquote (`). The second form of the **if** command *must* have the word, *then* following the expression. Several **if** statements can be chained together, through the use of the **else** statement. This statement must have a corresponding **endif** statement.

```
if (expression) simple command

if (expression) then
  ...
else
  ...
endif
```

The **switch** statement can replace several **if** ... **then** statements. For the string given in the **switch** statement's argument, commands following the **case** statement with the matching pattern are executed until the **endsw** statement. These patterns may contain *?* and * to match groups of characters or specific characters.

```
switch (string)
  case pattern1:
    commands...
    breaksw
  case pattern2:
    commands...
    breaksw
  default:
    commands...
    breaksw
endsw
```

The **while** statement will enter the loop only if the expression evaluates to *true* (or non-zero). Once within the loop, the commands within it will continue to execute until the expression evaluates to *false* (zero).

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

The **foreach** statement takes an array variable and places the contents of each array element into the loop variable for each iteration.

```
foreach variable (array variable or list)
...
end
```

The **break** statement breaks out of the current loop.

```
break
```

The **continue** command returns to the top of the current loop after testing the condition for the loop.

```
continue
```

The **shift** command without arguments will shift the variable, `argv` down by one element. In other words, `argv[2]` becomes `argv[1]` and so forth, with `argv[1]` being discarded. With an array variable argument, the **shift** command performs the same operation on the variable specified.

```
shift
```

```
shift variable
```

Conditional expressions

The expressions used in the **while** and **if** commands are similar to C language expressions, with these exceptions:

=~	If the right hand side matches a pattern, (i.e., similar to filename matching, with asterisks and question marks.) the condition is true.
!~	If the right hand side doesn't match a pattern, the condition is true.
-d $var	True if the file is a directory.
-e $var	True if the file exists.
-f $var	True if the file is a file. (I.e., not a directory)
-o $var	True if the file is owned by the user.
-r $var	True if the user has read access.
-w $var	True if the user has write access.
-x $var	True if the user has execute access.
-z $var	True if the file is zero-length.

Command line shortcuts

Here are a few keys which may be pressed to perform certain functions.

<escape>

The escape key preceded by a partial command or filename will attempt to complete the filename. If there are more than one filename matching, the common letters are completed, and the C shell beeps.

Control-D

When typed after a partial filename, C shell gives a list of all matching filenames or commands.

Control-W

Erases over the previous word.

Shell scripting

Shell scripts are programs written in C shell. They are plain text files which can be edited and created by any text editor. There are a few guidelines to follow, however.

1. Create a file using any text editor. The first line *must* begin with the string *#!/bin/csh*.

2. Give yourself execute permission with the **chmod u+x** *filename* command.
3. You can run the shell script by simply typing `filename` as if it were a regular command.

The shell script file can contain any commands which can be typed in, as well as the control structures described above.

Shell script arguments

When you write a shell script, a special array variable `argv` is created with the arguments entered. For example, if the shell script *tester* is created, and is run by typing it with the arguments as shown, `tester one two jump`, the array variable `argv` will contain "one", "two", and "jump" in its three elements.

- Go to the introduction page.
- Go to the UH CoE WWW Server home page.

Author: Ben Yoshino (ben@wiliki.eng.hawaii.edu)

Comments, Questions? | E-mail: *webmaster@wiliki.eng.hawaii.edu*
