CHAPTER 14

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. In the following pages, we shall discuss about C shell in comparison to Bourn (Bourn again Shell) only.

14.1 ALIAS

In C shell also, we can define alias's. Unlike Bourn shell, we can define an alias without equal to symbol. For example, in the following we are defining two aliases.

```
alias dir 'ls –l'
alias NU 'who|wc –l'
```

As usual, by simply typing dir or NU we can run these aliases. To make them available for all the login sessions, we have to enter them in .cshrc file.

Here also, if we execute **alias** command without any arguments it displays all the aliases defined currently.

We can use **unalias** command to remove an alias.

14.2 DEFINING VARIABLES WITH SET, SETENV

We have already seen in the examples above how to define variables in Bourne-shell. In C shell, 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 = "Rama"

Here, Rama is assigned to variable local.

set myname = "`whoami`"

Here, whoami command is executed as it is enclosed between back quotes and the result is assigned to the variable myname.

These variables are then referred to by using the dollar '$' symbol. i.e. The value of the variable local is $local. For example, to print their values we can use echo command as shown below.

echo $local $myname

In C shell, global (or Environment) 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`"
```

Here also, "who am i" command is executed and the result is assigned to MYNAME.

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

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

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.

Some of the global variables of C shell

cdpath	autolist	cwd	autologout	dirstack
histfile	gid	home	path	user

14.3 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.

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

echo $#path

23

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.

14.4 PIPES AND REDIRECTION IN CSH

Like Bournshell, C shell also supports redirection operators such as :

<, >, >>, <<, |, and &.

We have already discussed about redirection operators with Bourn Shell. Most of the concepts are applicable in C shell also. For the sake of continuity, we shall repeat the concepts.

We know 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').

The '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
touch blah      # create an empty file blah
echo test > blah
blah: File exists.
```

If we are worried about overwriting files, then we can set 'noclobber' in our '.cshrc' file. 'noclobber' can be overridden using the '!' symbol. So

```
set noclobber
touch blah      # create an empty file blah
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'
'<<' Here the document.

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,

```
mail mark <<quit
Hello mark
Nothing much to say...
so bye
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. We have already explained about piping with Bourn Shell. The same works with C shell also. 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.

The 'tee' and 'script' commands

Occasionally we might want to have a copy of what we see on our terminal sent to a file. 'tee' and 'script' do this. For instance,

find / -type l -print | tee myfile

sends a copy of the output of 'find' to the file 'myfile'. 'tee' can split the output into as many files as you want :

***command* | tee *file1 file2* ....**

We can also choose to record the output an entire shell session using the 'script' command.

```
$script mysession
Script started, file is mysession
$echo Big brother is scripting you
Big brother is scripting you
$exit
exit
Script done, file is mysession
```

The file 'mysession' is a text file which contains a transcript of the session.

14.5 EXTRACTING PARTS OF A PATHNAME

We know a file or directory's path consists of a number of different parts :

- The path to the directory where a file is held.
- The name of the file itself.
- The file extension (after a dot).

By using one of the following modifiers, we can extract these different elements. We do have some commands such as dirname, basename, etc.,.

`':h'` The path to the file
`':t'` The filename itself
`':e'` The file extension
`':r'` The complete file-path minus the file extension

Here are some examples and the results :

```
set f = ~/progs/c/test.c
echo $f:h
/home/mark/progs/c
echo $f:t
test.c
echo $f:e
c
echo $f:r
/home/mark/progs/c/test
```

14.6 C SHELL 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 we want to perform an operation on numbers rather than strings, we 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
```

If one observes keenly, these operators are precisely those found in the C programming language.

14.7 C SHELL BUILTIN COMMANDS

When the C shell was created, some of the programs that were external to the Bourne shell were built into the new shell. In addition to the list of commands above, the following commands are usually built into most versions of the C shell.

alias [Name [WordList]]	**bg [%Job ...]**	**breaksw**	**case Label :**
chdir [Name]	default:	dirs	else
then ... else ...	end	endif	endsw
fg [%Job ...]	foreach Name (List) Command...	glob List	goto Word
hashstat	history [-r | -h] [n]	if (Expression) Command	jobs [-l]
kill -l |[[-Signal] % Job...|PID...]	limit-hResourceMax-Use [] [[]]	login	logout
nice [+n] [Command]	nohup [Command]	notify [%Job...]	onintr [- | Label]
popd [+n]	pushd [+n|Name]	rehash	repeat
setenv built-in	stop [%Job ...]	suspend	switchstring ()
time	umask [Value]	unalias *|Pattern	unhash
	unsetenv Pattern	while (Expression) Command...end	

History in C Shell

We have already enjoyed the tint of history in Bourn shell. Each command, or ''event'', input from the terminal is saved in the history list. The previous command is always saved, and the **history** shell variable can be set to a number to save that many commands. The **histdup** shell variable can be set to not save duplicate events or consecutive duplicate events.

Saved commands are numbered sequentially from 1 and stamped with the time. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an '!' in the **prompt** shell variable.

The shell actually saves history in expanded and literal (unexpanded) forms. If the **histlit** shell variable is set, commands that display and store history use the literal form.

The *history* builtin command can print, store in a file, restore and clear the history list at any time, and the **savehist** and **histfile** shell variables can be can be set to store the history list automatically on logout and restore it on login.

History substitutions introduce words from the history list into the input stream, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence.

History substitutions begin with the character '!'. They may begin anywhere in the input stream, but they do not nest. The '!' may be preceded by a '\' to prevent its special meaning; for convenience, a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. History substitutions also occur when an input line begins with '^'. This special abbreviation will be described later. The characters used to signal history substitution ('!' and '^') can be changed by setting the **histchars** shell variable. Any input line which contains a history substitution is printed before it is executed.

A history substitution may have an ''event specification'', which indicates the event from which words are to be taken, a ''word designator'', which selects particular words from the chosen event, and/or a ''modifier'', which manipulates the selected words.

An event specification can be

n A number, referring to a particular event

-*n* An offset, referring to the event *n* before the current event

#

The current event. This should be used carefully in **csh**(1), where there is no check for recursion. *tcsh* allows 10 levels of recursion. (+)

! The previous event (equivalent to '-1')

s The most recent event whose first word begins with the string *s*

?*s*?

The most recent event which contains the string *s*. The second '?' can be omitted if it is immediately followed by a newline.

For example, consider this bit of someone's history list :

9 8:30 nroff -man wumpus.man

10 8:31 cp wumpus.man wumpus.man.old

11 8:36 vi wumpus.man

12 8:37 diff wumpus.man.old wumpus.man

The commands are shown with their event numbers and time stamps. The current event, which we haven't typed in yet, is event 13. '!11' and '!-2' refer to event 11. '!!' refers to the previous event, 12. '!!' can be abbreviated '!' if it is followed by ':' (':' is described below). '!n' refers to event 9, which begins with 'n'. '!?old?' also refers to event 12, which contains 'old'. Without word designators or modifiers history references simply expand to the entire event, so we might type '!cp' to redo the copy command or '!!|more' if the 'diff' output scrolled off the top of the screen.

History references may be insulated from the surrounding text with braces if necessary. For example, '!vdoc' would look for a command beginning with 'vdoc', and, in this example, not find one, but '!{v}doc' would expand unambiguously to 'vi wumpus.mandoc'. Even in braces, history substitutions do not nest.

(+) While **csh**(1) expands, for example, '!3d' to event 3 with the letter 'd' appended to it, *tcsh* expands it to the last event beginning with '3d'; only completely numeric arguments are treated as event numbers. This makes it possible to recall events beginning with numbers. To expand '!3d' as in **csh**(1) say '!\3d'.

To select words from an event we can follow the event specification by a ':' and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are :

0 The first (command) word
n The *n*th argument
^ The first argument, equivalent to '1'
$ The last argument
% The word matched by an ?*s*? search
x-y A range of words
-y Equivalent to *'0-y'*
*

Equivalent to '^-$', but returns nothing if the event contains only 1 word
*x** Equivalent to *'x-$'*
x- Equivalent to *'x*'*, but omitting the last word ('$')

Selected words are inserted into the command line separated by single blanks. For example, the 'diff' command in the previous example might have been typed as 'diff !!:1.old !!:1' (using ':1' to select the first argument from the previous event) or 'diff !-2:2 !-2:1' to select and swap the arguments from the 'cp' command. If we didn't care about the order of the 'diff' we might have said 'diff !-2:1-2' or simply 'diff !-2:*'. The 'cp' command might have been written 'cp wumpus.man !#:1.old', using '#' to refer to the current event. '!n:- hurkle.man' would reuse the first two words from the 'nroff' command to say 'nroff -man hurkle.man'.

The ':' separating the event specification from the word designator can be omitted if the argument selector begins with a '^', '$', '*', '%' or '-'. For example, our 'diff' command might have been 'diff !!^.old !!^' or, equivalently, 'diff !!$.old !!$'. However, if '!!' is abbreviated '!', an argument selector beginning with '-' will be interpreted as an event specification.

A history reference may have a word designator but no event specification. It then references the previous command. Continuing our 'diff' example, we could have said simply 'diff !^.old !^' or, to get the arguments in the opposite order, just 'diff !*'.

The word or words in a history reference can be edited, or ''modified'', by following it with one or more modifiers, each preceded by a ':' :

h Remove a trailing pathname component, leaving the head.
t Remove all leading pathname components, leaving the tail.
r Remove a filename extension '.xxx', leaving the root name.
e Remove all but the extension.
u Uppercase the first lowercase letter.
l Lowercase the first uppercase letter.

s/*l*/*r*/

Substitute *l* for *r*. *l* is simply a string like *r*, not a regular expression as in the eponymous ed command. Any character may be used as the delimiter in place of '/'; a '\' can be used to quote the delimiter inside *l* and *r*. The character '&' in the *r* is replaced by *l*; '\' also quotes '&'. If *l* is empty (''''), the *l* from a previous substitution or the *s* from a previous '?*s*?' event specification is used. The trailing delimiter may be omitted if it is immediately followed by a newline.

& Repeat the previous substitution.
g Apply the following modifier once to each word.

a (+)

Apply the following modifier as many times as possible to a single word. 'a' and 'g' can be used together to apply a modifier globally. In the current implementation, using the 'a' and 's' modifiers together can lead to an infinite loop. For example, ':as/f/ff/' will never terminate. This behavior might change in the future.

p Print the new command line but do not execute it.

q Quote the substituted words, preventing further substitutions.

x Like q, but break into words at blanks, tabs and newlines.

Modifiers are applied to only the first modifiable word (unless 'g' is used). It is an error for no word to be modifiable.

For example, the 'diff' command might have been written as 'diff wumpus.man.old !#^:r', using ':r' to remove '.old' from the first argument on the same line ('!#^'). We could say 'echo hello out there', then 'echo !*:u' to capitalize 'hello', 'echo !*:au' to say it out loud, or 'echo !*:agu' to really shout. We might follow 'mail -s "I forgot my password" rot' with '!:s/rot/root' to correct the spelling of 'root' (but see **Spelling correction** for a different approach).

There is a special abbreviation for substitutions. '^', when it is the first character on an input line, is equivalent to '!:s^'. Thus we might have said '^rot^root' to make the spelling correction in the previous example. This is the only history substitution which does not explicitly begin with '!'.

In *csh* as such, only one modifier may be applied to each history or variable expansion. In *tcsh*, more than one may be used, for example

```
% mv wumpus.man /usr/man/man1/wumpus.1
% man !$:t:r
man wumpus
```

In *csh*, the result would be 'wumpus.1:r'. A substitution followed by a colon may need to be insulated from it with braces :

```
> mv a.out /usr/games/wumpus
> setenv PATH !$:h:$PATH
Bad ! modifier: $.
> setenv PATH !{-2$:h}:$PATH
setenv PATH /usr/games:/bin:/usr/bin:.
```

The first attempt would succeed in *csh* but fails in *tcsh*, because *tcsh* expects another modifier after the second colon rather than '$'.

Finally, history can be accessed through the editor as well as through the substitutions just described. The *up-* and *down-history*, *history-search-backward* and *-forward*, *i-search-back* and *-fwd*, *vi-search-back* and *-fwd*, *copy-prev-word* and *insert-last-word* editor commands search for events in the history list and copy them into the input buffer. The *toggle-literal-history* editor command switches between the expanded and literal forms of history lines in the input buffer. *expand-history* and *expand-line* expand history substitutions in the current word and in the entire input buffer respectively.

14.8 THE .CSHRC AND .LOGIN FILES

Most users run the C-shell, 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 us the command prompt.
- If and only if this is the *login shell* (not a sub-shell that we 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. We request users to recall about .profile or .bashrc configurations files which are used with Bourne shell. These, .cshrc and .login are their counterparts.

```
#
# .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
```

We do have variety of C shell variants which may differ in their startup of files. In some versions, csh.login, csh.cshrc, etc., are used. In terminal C shell, .tcshrc is used. Also, .logout is executed when we logout from the shell. Thus, startup files changes from C shell version to version.

14.9 C SHELL SCRIPTS

As explained earlier, in order to instruct C – shell to execute the statements in a shell script we have to add **#!/bin/csh –f** statement as shown below.

```
#!/bin/csh -f
#
# A simple script which greets the user
echo Hi $0, Have a Good Day
```

As usual, we do require to save a shell program in a file. For that file we have to give user executable permissions. In order to run a shell script, we have to type its name at the command prompt.

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.

Sub-shells ()

The C-shell does not allow us to define subroutines or functions, but we can create a local shell, with its own private variables by enclosing commands in parentheses.

```
#!/bin/csh
cd /etc
( cd /usr/bin; ls * ) > myfile
pwd
```

This program changes the working directory to /etc and then executes a sub-shell which *inside the brackets* changes directory to /usr/bin and lists the files there. The output of this private shell are sent to a file 'myfile'. At the end we print out the current working directory just to show that the 'cd' command in brackets had no effect on the main program.

Normally both parentheses must be on the same line. If a subshell command line gets too long, so that the brackets are not on the same line, you have to use backslash characters to continue the lines,

```
(   command \
    command \
    command \
)
```

14.10 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

Style 1:

```
if (expr) command
```

Style 2:

```
if (expr) then
   command
   command..
else
```

```
    command
    command..
endif
```

Style 3:

```
if (expr) the
——
else if(expr)then
—
else if(expr) then
——
else
——
endif
```

The third one is called as nested if. In fact, we can have any number of else if clauses.

The switch clause

```
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
filtest –F filename Displays device and i-node number of the given file.
The simplest way to learn about these statements is to use them, so we shall now look at some examples.

Example

```
#!/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.

Switch example: configure script

Here is another example which compiles a software package. The problem this script tries to address is the following. There are many different versions of UNIX and they are not exactly compatible with one another. The program this file compiles has to work on any kind of UNIX, so it tries first to determine what kind of UNIX system the script is being run on by calling 'uname'. Then it defines a variable 'MAKE' which contains the path to the 'make' program which will build *software*. The make program reads a file called 'Makefile' which contains instructions for compiling the program, but this file needs to know the type of UNIX, so the script first copies a file 'Makefile.src' using 'sed' replace a dummy string with the real name of the UNIX. Then it calls make and sets the correct permission on the file using 'chmod'.

```
#!/bin/csh -f
#################################################
#
#
# CONFIGURE Makefile AND BUILD software
#
#
#################################################
set NAME = ( `uname -r -s` )
switch ($NAME[1])
    case SunOS*:
                switch ($NAME[2])
                    case 4*:
                             setenv TYPE SUN4
                             setenv MAKE /bin/make
                             breaksw
                    case 5*:
                             setenv TYPE SOLARIS
                             setenv MAKE /usr/ccs/bin/make
                             breaksw
                endsw
```

```
            breaksw
    case ULTRIX*:
            setenv TYPE ULTRIX
            setenv MAKE /bin/make
            breaksw
    case HP-UX*:
            setenv TYPE HPUX
            setenv MAKE /bin/make
            breaksw
    case AIX*:
            setenv TYPE AIX
            setenv MAKE /bin/make
            breaksw
    case OSF*:
            setenv TYPE OSF
            setenv MAKE /bin/make
            breaksw
    case IRIX*:
            setenv TYPE IRIX
            setenv MAKE /bin/make
            breaksw
    default:
            echo Unknown architecture $NAME[1]
endsw
    # Generate Makefile from source file
sed s/HOSTTYPE/$TYPE/ Makefile.src > Makefile
echo "Making software. Type CTRL-C to abort and edit Makefile"
$MAKE software       # call make to build program
chmod 755 software   # set correct protection
```

14.11 GOTO STATEMENT

We can use goto statement in C shell. Its format is :

goto label;

We do have another means of program jumps. If we press break key, we wanted the program control to jump to some place. Then, the following will be useful.

onintr label;

14.12 LOOPS IN C SHELL

Like Bourn Shell, the C-shell also has three loop structures: 'repeat', 'while' and 'foreach'.

The structure of these loops is as follows

Syntax of repeat loop

```
repeat number-of-times command
```

Syntax of while loop

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

Syntax of for loop

```
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 super user 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.)

Example

The following program executes infinite times.

```
while(1)
echo "Hell0";
end
```

Example

The following program takes a number along the command line and prints all the natural numbers from 0 to till this number. Here, we use while loop.

```
set limit=argv[0];
set index=0;
while($index <= $limit)
echo $index;
@index++;
end
```

14.13 READING 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.

Example

The following program greets the users whose names are given inside the list of arguments.

```
foreach X ( Rao Raj Abhi )
echo Good Morning $X;
end
We will get the following output.
Good Morning Rao
Good Morning Raj
Good Morning Abhi
```

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
```

Example

Like C programs, C-shell scripts can accept command line arguments. Suppose we 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 we 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, we 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 we 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.

Example

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
```

Example

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.

```
#!/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
```

Differences between Bourn and C shell are given in the following table.

Bourne Shell	C Shell	Notes
$ command	% command	invoke, (run), the process, (command) in the foreground.
$ command &	% command &	invoke the process in the background.
$ command1 ; command2	% command1 ; command2	invoke command1, when it is finished,invoke command2This is often frowned upon in scripts.

$ command1 2>&1 ¦ command2	% command1 ¦ command2	invoke command1, send, (pipe), the output directly to command2. Both processes runin parallel.
$ command1 2>&1 ¦ command2	% command1¦& command2	invoke command1, send, (pipe), both standard out and standard error directly to command2. Both processes runin parallel.
$ sh file	% csh file	invoke a shell, then run the "file"Note: when testing scripts, it is often useful touse the sh -vx invocationof the command to see exactly which commands are executed.
$. file	% source file	run the file, and keep any changes madeto the environment. Handy for rerunningfiles like ".cshrc" or ".login".
$ (command1 ; command2)	% (command1; command2)	run both commands in a child shell called a *sub-shell*. Useful forcollecting all the output from multipleprocesses. Example (date ; ls) < what_when
$ command > reg_ output 2> error_output	% (command > reg_ output) >& error_output	Sometimes it is desirable to send the "normal" output to one file, andthe errors to another file. It is easy to doin the Bourne shell, as each file has its ownnumeric identifier. You must use subshellingin the C SHell.

14.14 CONCLUSIONS

In this chapter, we have introduced to user to elements of C shell programming. Conceptual differences between Bourn and C shells are highlighted. Examples which explains the C shell syntax are included.