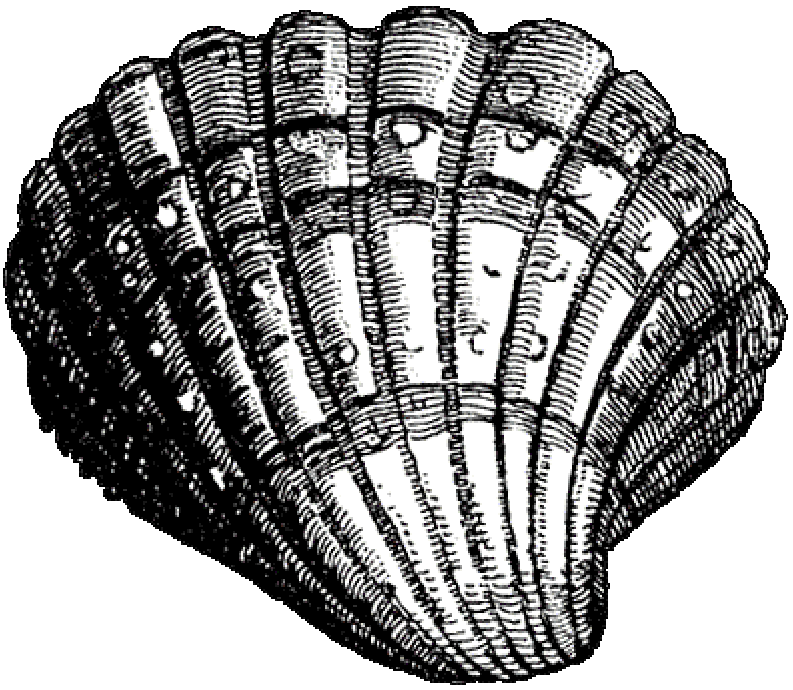
ITS Information Technology Service

Guide 3

Version 3.0

Writing C-shell scripts

The C-shell is the program which interprets the commands that you type at the keyboard when you use the Unix operating system. It is possible to put C-shell commands into a file, called a *script*. This course teaches you how to write these C-shell scripts. It assumes that you are an experienced user of Unix, and are familiar with the contents of the Information Technology Service document *Guide 2: Further UNIX*.



Durham University

Document code: **Guide 3**
Title: **Writing C-shell scripts**
Version: **3.0**
Date: **May 2007**
Produced by: **University of Durham Information Technology Service**

Conventions:

In this document, the following conventions are used:

- A typewriter font is used for what you see on the screen.
- A **bold typewriter font** is used to represent the actual characters you type at the keyboard.
- A ***slanted typewriter font*** is used for items such as filenames which you should replace with particular instances.
- A **bold font** is used to indicate named keys on the keyboard, for example, **Esc** and **Enter**, represent the keys marked Esc and Enter, respectively.
- A **bold font** is also used where a technical term or command name is used in the text.
- Where two keys are separated by a forward slash (as in **Ctrl/B**, for example), press and hold down the first key (**Ctrl**), tap the second (**B**), and then release the first key.

Contents

1. Introduction

1.1 The aim of this course

The C-shell is a program that can be executed from the UNIX operating system. It is the program that understands the commands that a user types at the keyboard. However, it is also possible to put C-shell commands into a file, called a *script.* The aim of this course is to introduce details of how to write C-shell scripts.

1.2 Before you begin

This course assumes that you already have some understanding of some of the basic ideas of UNIX. This may have been achieved by attendance at the ITS's course **An introduction to the UNIX operating system**. It is also desirable for you to be familiar with the contents of the ITS document *Guide 2: Further UNIX*.

1.3 Teaching yourself

This document has been written so that it can be used as a teach-yourself guide. If you prefer to attend a **Writing C-shell Scripts** course, please contact the IT Service Desk.

1.4 Further information about UNIX

For full details about the C-shell, you need to look at the **man** page for the **csh** command. The following book is a lot easier to read, and it provides a thorough coverage of the C-shell:

The UNIX C Shell Field Guide, by G. Anderson and P. Anderson, published by Prentice Hall (1986). Unfortunately, it costs £34.75.

2. What is the shell?

When a user types a command line at the keyboard, the part of the operating system that analyses this line is called the command line processor (CLP). In UNIX, the CLP is completely separate from the rest of the operating system. So, the CLP is written as a separate program, and each user communicates with a copy of this program. The program itself is called the **shell**.

Surprisingly enough, there are usually at least two shells available on a UNIX system: they are the **Bourne shell** (**sh**) and the **C-shell** (**csh**). There may be other shells, such as the **Korn shell** (**ksh**) and the **Bourne-again shell** (**bash**). The Bourne shell was devised by Steve Bourne of Bell Labs, and the C-shell was developed by Bill Joy of the University of California at Berkeley (UCB). They are both command line processors. However, the language that is used to communicate with them is different. In this course, we will be concerned with the C-shell.

3. Simple shell scripts

3.1 Getting started

It will be useful to create the files for this course in a new subdirectory:

1 type

```
cd
```

2 and then type

```
mkdir cshell
```

3 followed by

```
cd cshell
```

To save you from doing a lot of typing, some files for this course have already been prepared. It will be useful to copy these files to this new directory, so:

1 type

```
cp ~courses/cshell/simple/* .
```

Note: that this command line ends in a space followed by a dot.

1 Type

```
ls -l
```

in order to see which files have been copied. We will look at the contents of each of these files as we go through the course.

3.2 What is a shell script?

Two commands that help you find out what is happening on your computer are **w** and **who**.

1 Type

```
w
```

2 followed by

```
who
```

In your use of UNIX, the C-shell has been used to process the commands that you type at the keyboard. It is also possible to get the shell to obey commands given in a file. Such a file of commands is called a **shell script** (or a *shell procedure*).

Note: The name of this file should not be the same as that of a UNIX command. In particular, do not use the name **test** as there is a UNIX command called **test**.

We will look at a simple example. You should have a file called **spy**:

1 type

```
cat spy
```

This command should output:

```
#!/bin/csh
# spy dxy3abc 920520
# spy outputs some details about what's happening on the computer.
# It takes no parameters.
w | more
echo ""
echo -n "Number of users: "
who | wc -l
```

The file **spy** contains a script. The first line of any file that is a C-shell script should contain:

```
#!/bin/csh
```

It is important that the **#!** are in the first two columns.

A **hash character** (i.e., a **#**) also marks the start of a **comment**: this is a piece of text that is only present for documentation purposes. A comment can appear on a line of its own, or it can be given after the command at the end of the command line.

The commands of the shell script that is in the file **spy** will be obeyed if you:

1 type

csh spy

Notes: the command **echo " "** produces a blank line. If an **n** option is used with the **echo** command, the parameters are sent to the standard output without an end-of-line character.

3.3 Using parameters to pass information to a shell script

You will often want to pass information to a shell script. This is done through **parameters**. In the shell script, the first, second, ..., ninth parameters can be accessed using the notation **$1, $2**, **...**, **$9**. They can also be accessed using the notation **$argv[1], $argv[2], ..., $argv[9]**.

Note: argv is called a **wordlist**. Later in these notes (in Section 8.5), we will look at other ways of creating a shell variable that contains a wordlist.

Suppose you want a shell script that tells you some information about a file:

1 type

cat fileinfo

The file **fileinfo** contains:

```
#!/bin/csh
# fileinfo dxy3abc 920307
# fileinfo displays some details about a file.
# It takes one parameter which is the name of a file.
ls -l $1
wc -l $1
file $1
```

The current directory has a file called **fred**, so in order to execute the script on the file **fred**:

1 type

  csh fileinfo fred

This command has the same effect as:

```
ls -l fred
wc -l fred
file fred
```

3.4 Making a shell script executable

It is a nuisance to have to type:

```
csh spy
csh fileinfo fred
```

It would be nicer if the scripts could be executed by:

```
spy
fileinfo fred
```

In order to be able to do this, the files containing the shell scripts must be 'executable'.

The **file mode** of a file can be changed using the **chmod** command. Normally, a file that you create using an editor can only be read from or written to:

1 type

  ls -l

Notice that columns 2 to 4 of this output contain the characters **rw-.** In order to make the files **spy** and **fileinfo** executable by you:

1 type

  chmod u+x spy fileinfo

2 and then type

  ls -l

Notice that columns 2 to 4 now contain the characters **rwx**.

Having done this, these shell scripts can be used just like any other UNIX command:

1 type

  spy

2 followed by

  fileinfo fred

Exercise A

Produce a shell script called **wld** which contains the commands: **who, ls** and **date**. [Remember to include **#!/bin/csh** as the first line of the script.] Check that the script works by typing the command line:

```
csh wld
```

Now make the file executable, and then type:

```
wld
```

Note: most of these exercises involve the writing of a shell script. A possible solution to an exercise is given in an appropriately named file in the directory **~courses/cshell/solutions**. For example, a solution to this exercise is in **~courses/cshell/solutions/wld**. The contents of this file will be displayed if you type:

```
peep wld
```

This is because you have a shell script in your current directory called **peep**.

Exercise B

Produce a shell script called **lpqs** which displays the contents of the printer queues *printername1* and *printername2*. In your shell script, use the **echo** command to identify the lines of the output. Make the file executable so that you are able to execute it just by typing. (Note: the solution given to this exercise uses the names lasercc1 and dcc1 which do not correspond to any of the networked printers.)

```
lpqs
```

3.5 Storing shell scripts in a subdirectory

We have seen that a shell script like spy can be executed just by typing:

```
spy
```

provided that you are in the subdirectory containing the file **spy**. If you have written a shell script that you may want to use from any of your subdirectories, it is useful to put the shell script into a special subdirectory (just containing executable commands) rather than having copies of the shell script in each of the subdirectories where you might want to use it.

It is conventional for a user to put their private collection of shell scripts in the directory **~/bin**:

1 type

```
mkdir ~/bin
```

This directory needs to be included in the list of directories that are searched in order to find commands. This list is contained in the *shell variable* called **path**. The contents of this list can be altered by adding a line to the file **~/.login**.

At this point use an editor to alter the contents of the file **~/.login**. It needs to have the following command added to the end of the file:

```
set path = ( $path ~/bin )
```

If you use the Pico editor, for example, you can alter this file by typing the following commands:

```
cd
cp .login .login.old
pico .login
```

Make sure that you do not alter any of the existing lines of the file.

This alteration to the **.login** file will have no immediate effect. In order for it to have some effect:

1 type

```
source ~/.login
```

Now make sure that you are in the directory being used for this course:

1 type

```
cd ~/cshell
```

Having done that, we ought to move shell scripts like **spy** and **fileinfo** to the **~/bin** directory:

1 type

```
mv spy fileinfo peep wld lpqs ~/bin
```

2 followed by

```
ls -l
```

You should find that the files **spy, fileinfo, peep, wld** and **lpqs** are no longer in this directory. They have been moved to the **~/bin** directory.

Shell scripts that have just been added to a directory that is mentioned in the path cannot be executed immediately.

1 Type

```
spy
```

You should get the error message

```
spy: Command not found
```

This occurs because the shell has a built-in shortcut method of getting to such commands. And it works out the short-cuts whenever it reads the **set path** command in the **.login** file. You can get the shell to re-initialise its short-cuts, if you:

1 type

```
rehash
```

If you now:

1 type

```
spy
```

you should find that it executes the shell script that is in the file **~/bin/spy**.

Exercise C

Produce a shell script called **showbin** that displays on the screen the contents of the shell script passed as a parameter. For example, the command:

```
showbin spy
```

should execute the **more** command on the file **~/bin/spy**.

1. Create the file **showbin** in the directory **~/cshell**. Make the file executable, and test it by typing:
   ```
   showbin spy
   ```
2. If it works, type:
   ```
   mv showbin ~/bin
   ```
3. to move the file to the ~/bin directory. Type:
   ```
   rehash
   ```
4. and then test it again by typing:
   ```
   showbin spy
   ```

4. Parameters, shell variables and ‘here documents’

4.1 Getting some more pre-prepared files

We will now obtain some more files that have already been prepared:

1. type
   ```
   cp ~courses/cshell/others/* ~/bin
   ```

We are not copying these files into the current directory (**~/cshell**) but into **~/bin:**

2. type
   ```
   ls -l ~/bin
   ```

Notice that the files that have just been copied already have the file modes set so that we can execute them. However, since new executable files have been added to a directory mentioned in the path, we will need to type

```
rehash
```

if we wish to execute any of them from another directory.

4.2 How to refer to all of the parameters of a shell script

We have seen that **$1, $2, ..., $9** can be used to refer to a particular parameter of the shell script. The notation **$*** or **$argv[*]** is a way of referring to all of the parameters.

The file **~/bin/fileinfo2** contains an example of **$***. The shell script **showbin** produced in the last exercise will be used to output the contents of this file:

1 type

showbin fileinfo2

This file contains:

```
#!/bin/csh
# fileinfo2 dxy3abc 920307
# fileinfo2 displays some details about the files passed as parameters.
ls -l $*
wc -l $*
file $*
```

Now execute it:

1 type

fileinfo2 fred bert jane

This command is equivalent to the commands:

```
ls -l fred bert jane
wc -l fred bert jane
file fred bert jane
```

4.3 Using shell variables

You can use *variables* whilst communicating with the shell. A **shell variable** is given a value in the following way:

```
set VariableName = SomeValue
```

In particular, a string of characters can be stored in a variable. For example, suppose there is a rather long directory name which you know you will have to type many times. You can save some of this typing by storing the directory name in a variable:

```
set dir = ~courses/firstunix
```

The value of a shell variable can be obtained by using the notation: **$VariableName .** So:

```
cd $dir
```

is equivalent to the command:

```
cd ~courses/firstunix
```

And:

```
cat $dir/portia.txt
```

is equivalent to:

```
cat ~courses/firstunix/portia.txt
```

There are some predefined shell variables. It is best not to use these names for your own variables. A list of the shell variables that currently have values will be displayed if you:

1 type

```
set
```

Besides the shell variables that are only active for the current shell, there are also *environment variables*. These will have effect all the way from login to logout. A list of the environment variables can be displayed:

1 type

```
env
```

Note: suppose you have files called **amap, bmap**, and so on, and a shell script uses a shell variable **char** which contains a letter. An error will occur if the shell script contains something like:

```
cat $charmap
```

This will be understood as an attempt to access a shell variable called **charmap**. However, it is possible to use the notation **${VariableName**} instead of **$VariableName**. So, for the above example, the script can use:

```
cat ${char}map
```

Note: Although these notes introduce the **set** command as a way for a shell script to give a value to a shell variable, you may also find it useful to type commands like:

```
set dir = ~courses/firstunix
cat $dir/portia.txt
```

at the UNIX prompt.

Exercise D

Produce a shell script called **fileinfoagain** that is the same as the script in **~/bin/fileinfo** (given in Section 3.3), except that it has the following changes:

- add the line:

```
set filename = $1
```

as the first command to be executed by the script.

- Replace all other occurrences of **$1** in the script by **$filename**.

Note: introducing a shell variable that has the same value as a parameter is often done in order to make the rest of the shell script easier to understand.

4.4 Using ‘here documents’

It is often the case that a shell script contains a command which requires data. For example, suppose that as part of a shell script you want to edit a file automatically. It is not easy to do this using a screen editor; so, in the shell scripts for this course, the editor **ed** (which is always available on UNIX systems) will be used.

Although it is possible to tell the shell that we want to get the edit commands from a file:

```
...
ed bert <EditCommandsFilename
...
```

it is often preferable to include the edit commands in the shell script. This can be done by using what is called a **here document**.

Here is how it is done:

```
...
ed bert <<%
first line of edit commands
second line of edit commands
...
last line of edit commands
%
...
```

The lines between the two **%** characters form the **here document** — it is used as the standard input for the command that is given on the same line as the **<<**. The line following the last line of the input must contain a **%** on its own with the **%** appearing in the first column of the line. **Note**: the two **%** characters may be replaced by some other suitable character or by a word. **Note also**: any command or program can use a here document.

The here document may refer to parameters and variables. Here are three worked examples:

Example 1

Suppose a shell script is required that outputs the first line of the file passed as a parameter to the procedure. For example:

```
first fred
```

is to output the first line of the file **fred**.

Note: the task performed by **first** is better done by the **head** command: e.g.,

```
head -1 fred
```

Solution 1

The file **~/bin/first** contains a solution to this problem:

1 type

showbin first

The file **first** contains the following script:

```
#!/bin/csh
# first dxy3abc 920308
# first outputs the first line of a file.
# It takes one parameter which is the name of a file.
ed -s $1 <<%
1p
q
%
```

Now execute it:

1 type

first fred

The first line of the file **fred** will be displayed.

When an **ed** command is executed, the first task that is normally done by **ed** is to display the number of characters of the file that it is editing - this output can be suppressed by using **ed -s** rather than **ed**.

The **p** command of the editor **ed** displays lines on the screen. For example:

```
2,7p
```

means *print* (i.e., display on the screen) lines 2 to 7 of the file being edited. Note that: **1,1p** can be abbreviated to **1p**. The **q** command means *quit the editor.*

Example 2

Suppose a shell script is required that alters a file replacing all occurrences of one string by some other string. The script is to be called by typing a command like:

```
rao seperate separate first.tex
```

This is to change all occurrences of **seperate** to **separate** in the file **first.tex**.

Solution 2

1 Type

showbin rao

The following script is a solution to this problem:

```
#!/bin/csh
# rao dxy3abc 920308
# rao replaces all occurrences in a file of a string by another string
# It takes three parameters: old-string new-string filename
ed -s $3 <<LastLine
g/$1/s//$2/gp
w
q
LastLine
```

The **g** command of the editor **ed**:

```
g/str/cmd
```

means that the command **cmd** is to be performed on all lines that contain the string **str**. The **s** command:

```
s/old/new/
```

means substitute the string **new** for the string **old**. If the string **old** is a null string as in:

```
s//$2/gp
```

then the **old** string is the last string that was typed - in this case, it is **$1**. A **g** at the end of an **s** command means *change all occurrences on the line*, and a **p** means *print each line* on the screen.

Example 3

A shell script is required that indents each line of a file by 6 spaces. So:

```
add6 fred
```

is to alter the file **fred** so that each line of **fred** is indented by 6 spaces.

Solution 3

The following solution does not work:

```
#!/bin/csh
# add6 dxy3abc 920308
# add6 adds 6 spaces to the start of each line of a file.
# It takes one parameter which is the name of a file.
ed -s $1 <<%
1,$s/^/      /
w
q
%
```

It will fail because the shell would interpret the **$s** to mean *use the value of the variable* ***s****.* To prevent this, use a **** to quote the **$** character:

```
...
ed -s $1 <<%
1,\$s/^/      /
w
q
%
```

Alternatively, the shell is prevented from doing variable substitutions and parameter substitutions if the here document is quoted. This can be done in two ways:

```
...                                        ...
ed -s $1 <<\%                              ed -s $1 <<'LastLine'
1,$s/^/      /                      1,$s/^/      /
w                                                 w
q                                                 q
\%                                         'LastLine'
```

Exercise E

The UNIX command **tail** can be used to output the last 10 lines of a file, e.g.:

```
tail fred
```

Produce a shell script called **last10** which does this task. [Do not cheat by using the **tail** command in your script!] Your script should use **ed** and a *here document*. [**Hint:** the **ed** command **$-9,$p** can be used to output the last 10 lines of the file.] Test your script by:

```
last10 bert
```

Exercise F

Suppose you want a shell script to output some explanatory information to the screen, say, the following 4 lines:

```
You are using the NIH product called 'SuperEd' on the file $1.
We hope you find this product convenient and user-friendly.
You can support us in our endeavours by sending $27 to
the following address: NIH Software Ltd., NIH Street, NIHTown.
```

There are a number of methods that can be used to do this:

- The shell script could **cat** a file that contains the 4 lines.
- The shell script could use 4 **echo** commands.
- The shell script could use a **cat** command that gets its input from a *here document.*

Produce a shell script called **supered** that takes one parameter, the name of a file. The only task it performs is to output the above 4 lines using the third method mentioned above.

Note that the text contains a **$1**. Here your script should output the filename that is passed to **supered** as a parameter. The text also contains **$27**. Here you should output the characters **$27**.

5. Looping in a shell script

5.1 Constructs for controlling the flow

Like most programming languages, a shell language has constructs for controlling the flow through a shell script. The C-shell includes the following constructs: **if, switch, foreach, while** and **goto**.

In this section of the notes, we will be looking at the **foreach** command.

5.2 The foreach command

It is often necessary to repeat a sequence of commands a number of times. In the C-shell, this can be done using the shell's **foreach** command:

```
...
foreach VariableName (SomeList)
  command1
  command2
  ...
  commandn
end
...
```

The sequence of commands is executed a number of times: each time the variable following the **foreach** is given a new value from the list that is inside the parentheses.

For example, if a shell script contains:

```
foreach name (fred bert jane)
  echo $name
end
```

then the following output would be produced:

```
fred
bert
jane
```

We will now look at two common uses of **foreach** loops.

5.2.1 Looping for each of the parameters

Earlier (in Section 3.3), there was a shell script called **fileinfo** that output some information about the file passed as a parameter to the script. In Section 4.2, we looked at **fileinfo2**, a script that outputs this information for each of the files passed as a parameter.

We now look at a better version of **fileinfo2**.

1 Type

showbin fileinfo3

You should obtain:

```
#!/bin/csh
# fileinfo3 dxy3abc 920308
# fileinfo3 displays some details about the files passed as parameters
foreach filename ($*)
   echo --------------------
   ls -l $filename
   wc -l $filename
   file $filename
end
echo --------------------
```

Try the script out:

1 type

fileinfo3 fred bert

When the script is executed, the shell replaces the **$*** with a list of the parameters that have been passed to the script. **Advice:** if you are thinking of writing a shell script to do some task on a file, turn it into one which does the task on any number of files passed as parameters.

Note: do not use a **foreach** loop to execute a command which will already loop over filenames. For example:

```
...
foreach filename ($*)
   ls -l $filename
end
...
```

is very inefficient compared to:

```
...
ls -l $*
...
```

Exercise G

The calendar for the year 1992 can be displayed on the screen by the UNIX command:

```
cal 1992
```

Produce a shell script called cals which outputs a calendar for each year passed as a parameter to the script. For example:

```
cals 1992 2000 1752
```

5.2.2 Looping for all files matching a pattern

Suppose we want a version of **fileinfo** that produces output for all files that match a particular pattern; for example, for all files with names ending in **.tex**. We could use:

```
fileinfo3 *.tex
```

However, if this task is frequently performed, we could instead use a special script for it:

1 type
showbin texfileinfo

You should obtain:

```
#!/bin/csh
# texfileinfo dxy3abc 920308
# texfileinfo displays some details about the TeX files that are in
# this directory. It takes no parameters.
foreach filename (*.tex)
   echo --------------------
   ls -l $filename
   wc -l $filename
   file $filename
end
echo --------------------
```

When this script is executed, the shell will replace ***.tex** with a list of files that match this pattern. Execute the script:

1 type
texfileinfo

Exercise H

Produce a shell script called **zzs** which makes a copy of each file in the current directory. Each of the new filenames is to be the same as the old filename prefixed by the characters **zz**. So if the directory currently contains the files **bert, fred, jane**, after executing the command:

```
zzs
```

the directory will contain the files **bert, fred, jane, zzbert, zzfred**, and **zzjane**.

Exercise I

Find out what happens if a shell script containing:

```
foreach name ($*)
```

is executed when the shell script has no parameters. Are the commands in the **foreach** loop executed once or zero times?

Exercise J

What is the difference between the following two **foreach** constructs:

```
foreach name ($*)
foreach name (*)
```

5.3 Other looping commands

The C-shell also has **while** and **goto** commands that can be used to achieve looping. These commands will not be covered in this course.

6. Variable modifiers and the $0 notation

6.1 Variable modifiers

A pathname, such as **/home/hudson/pg/dxy3abc/papers/first.tex**, is sometimes stored in a shell variable. It can often be useful to extract the various components of the pathname. For example, we may want the directory part, i.e., **/home/hudson/pg/dxy3abc/papers**, or all of the pathname except the *extension*, i.e**.,** **/home/hudson/pg/dxy3abc/papers/first**. The C-shell has a number of variable *modifiers* that can be used to extract components. The role of each variable modifier is illustrated by the examples in the following table:

expression	*value*
$filename	/home/hudson/pg/dxy3abc/papers/first.tex
$filename:r	/home/hudson/pg/dxy3abc/papers/first
$filename:h	/home/hudson/pg/dxy3abc/papers
$filename:t	first.tex
$filename:e	tex

Here are some commands that make a backup copy of each **.tex** file that exists in the current directory:

```
foreach filename (*.tex)
   echo processing $filename
   set root = $filename:r
   cp -p $filename $root.old
end
```

6.2 The $0 notation

We have seen the use of **$1, $2, ..., $**9 to obtain the values of the first 9 parameters. The notation **$0** refers to the name by which the shell script was called. It is occasionally useful. [**Note:** there is no **$argv[0]** notation.]

For example, suppose the file **echoall** in the current directory contains a script that includes:

```
echo $0 $1 $2
```

then the command:

```
echoall hi there fred
```

will produce:

```
echoall hi there
```

If the file **echoall** is in another directory, say, in the directory **~dxy3abc/bin**, the command:

```
echoall hi there fred
```

will produce something like:

```
/home/hudson/pg/dxy3abc/bin/echoall hi there
```

A variable modifier may not be used with **$0**. If a shell script contains:

```
echo $0: about to process $filename
```

the **$0:** will be misunderstood as an attempt to use a variable modifier. Instead you can use:

```
echo ${0}: about to process $filename
```

However, the following might be more appropriate:

```
set myname = $0
set mynametail = $myname:t
echo ${mynametail}: about to process $filename
```

7. Decision making: using the if command

7.1 Introduction

There are two conditional commands available in the C-shell: the **if** command and the **switch** command. In this section, we will be considering the **if** command. We will look at *switches* in Section 9.

7.2 Each command returns an exit status

When a UNIX command has finished executing, it returns an integer value to the shell - this value is called the *exit status*. By convention, the value **0** means *the command ran successfully*, whereas a non-zero value is returned if the command was unsuccessful.

For example, the exit status returned by the **grep** command is as follows:

status	*meaning*
0	if at least one match has been found
1	if no matches have been found
2	if there are syntax errors or a file is inaccessible

An exit status is also returned to the shell whenever a shell script finishes. Normally, this is the exit status of the last command that was executed by the shell script. However, the shell script can arrange for a particular value to be returned by using the shell's exit command. For example:

```
exit 2
```

Note: any program written in a programming language can return an exit status by calling the UNIX function **exit**.

7.3 The shell's status variable

The variable status can be used to determine the exit status of the last command that was executed.

1 Type

grep date ~/bin/wld

2 followed by

echo $status

You should find that the **grep** outputs the line containing the **date** command, and the status variable has the value **0**.

1 Type

grep freddie ~/bin/wld

2 followed by

echo $status

Since the file **~/bin/wld** does not contain the line **freddie**, the **grep** command produces no output, and the status variable has the value **1**.

1 Type

grep date benny

2 followed by

echo $status

Since the file **benny** does not exist, you should find that the **grep** command outputs an error message, and the status variable has the value **2.**

7.4 The if command

The **if** command has the following syntax:

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

The **else if** section may occur zero or more times, and the **else** section is optional. Each of the expressions is evaluated in turn, and if an expression has the value **true** the corresponding sequence of commands is executed and then the command following the **endif** is executed.

We will look at some examples of the **if** command in Section 7.6.

7.5 The various kinds of conditions that can be tested

As in most programming languages, the expressions that are used in the condition parts of an **if** command can take many forms: you can compare two strings, you can compare two integers, or you can inspect an attribute of a file (e.g., whether it exists). You can also form more complex expressions by using **&&, ||** and **!** operators, and by using parentheses.

The following table attempts to explain some of the possibilities. There must be at least one space between each operand and its operator. The **<, >, <=** and **>=** operators can only be used if the operands are shell variables containing strings that are integer values.

condition	***meaning***
!b	is *b* false?
b && c	are *b* and *c* both true?
b || c	is at least one of *b* and *c* true?
i < j	is integer *i*< integer *j*?
i > j	is integer *i* > integer *j*?
i <= j	is integer *i* <= integer *j*?
i >= j	is integer *i* >= integer *j*?
i == j	do the integers *i* and *j* have the same value?
i! = j	are the integers *i* and *j* different?
s == t	do the strings *s* and *t* have the same value?
s != t	are the strings *s* and *t* different?
s =~p	does the string *s* match the pattern *p*?
s !~ p	does the string *s* not match the pattern *p*?
-r filename	is the *filename* readable?
-w filename	is the *filename* writeable?
-x filename	is the *filename* executable?
-e filename	does the file *filename* exist?
-o filename	does the current user own the file *filename*?
-z filename	is the file *filename* of zero length?
-f filename	is the file *filename* a plain file (rather than a directory)?
-d filename	is the file *filename* a directory (rather than a plain file)?

7.6 Examples of the if command

A few examples will be given to illustrate some of the possibilities of the **if** command.

The UNIX command **mv** can be used to change the name of a file:

```
mv fred bert
```

However, it often comes as a bit of a shock to some people that this command will still work if a file **bert** already exists — the original contents of **bert** are overwritten.

Here are some shell scripts that could be used in place of **mv**. They gradually increase in terms of user-friendliness/verboseness:

Example 1

```
if ( ! -e $2 ) then
   mv $1 $2
endif
```

Example 2

```
if ( -e $2 ) then
   echo mv has not been done because $2 already exists
else
   mv $1 $2
endif
```

Example 3

```
if ( (! -f $1) || -e $2 ) then
   echo mv not done because $1 is not a file or $2 already exists
else
   mv $1 $2
endif
```

Example 4

```
if ( ! -f $1 ) then
   echo mv has not been done because $1 is not a file
else if ( -e $2 ) then
   echo mv has not been done because $2 already exists
else
   mv $1 $2
endif
```

Example 5

```
if ( -f $1 ) then
   set fromfile = isafile
else
   set fromfile = isnotafile
   echo mv has not been done because $1 is not a file
endif
if ( -e $2 ) then
   echo mv has not been done because $2 already exists
else if ( $fromfile == isafile ) then
   mv $1 $2
endif
```

Exercise K

If you type:

```
cal 92
```

you will get the calendar for the year 92 rather than 1992. Produce a shell script called **nicecal** that will default to the 21st century if the parameter is less than 50 and to the 20th century if the parameter is between 50 and 99.

Exercise L

Produce a shell script called **filetest** that tests whether a file (that is passed as a parameter) exists, is a plain file, and is readable. If the file satisfies all these criteria, the script should execute an exit **0**. Otherwise, it should execute an exit **1**.

Produce a shell script called **nicecat** that takes a filename as a parameter. It should execute a **filetest** command, and then it should test the value of the status variable. If the variable has the value **0, nicecat** should use **cat** to display the file's contents. Otherwise, it should display an error message.

8. More about parameters and variables

8.1 The $#argv notation

The notation **$#argv** is a way of referring to the number of parameters that were given in the command line. It is often used to check that a shell script has been called with the right number of parameters.

```
if ( $#argv != 2 ) then
  echo Usage: nicemv currentname newname
  exit 1
endif
if ( ! -f $1 ) then
  echo nicemv: the mv command has not been done because $1 is not a file
  exit 2
endif
if ( -e $2 ) then
  echo nicemv: the mv command has not been done because $2 already exists
  exit 3
endif
mv $1 $2
```

Exercise M

The syntax of the UNIX command **chmod** is not particularly easy to remember. Produce a shell script called **plusx** which adds *execute* permission to each of the files passed as a parameter. If **plusx** is called with no parameters, it should instead add *execute* permission to each of the files in the directory **~/bin**.

8.2 The $$ notation

$$ is a way of referring to the *process number* of the current shell. The characters **$$** are often used as part of a filename in order to generate a unique name for a temporary file.

Suppose a shell script (called **whichttys**) is required that tells you the terminal numbers of a user that is logged in. For example:

```
whichttys dxy3abc dxy3def
```

is to output only the lines produced by the **who** command that contain the strings **dxy3abc** or **dxy3def**.

Here is one possibility for the file **whichttys:**

```
...
who >/tmp/whichttys$$
foreach username ($*)
   grep $username /tmp/whichttys$$
end
rm /tmp/whichttys$$
```

8.3 Shifting the parameters along by one

The shell's **shift** command removes the first parameter from the shell script's parameter list. After it has been executed,

- **$1** will contain what was in **$2**,
- **$2** will contain what was in **$3**,

and so on.

Note: the shift command does not affect the value of **$0**.

In Example 2 in Section 4.4, a shell script, **rao**, was given which replaces all occurrences in a file of one string by another string. Suppose a shell script is required that will do the task for any number of files, e.g.:

```
raos seperate separate ~/papers/*.doc
```

1. Type

   showbin raos

You should obtain:

```
#!/bin/csh
# raos dxy3abc 920312
# raos replaces all occurrences in files of a string by another string.
# The first two parameters are the old string and the new string.
# Other parameters are the names of the files to be altered.
set myname = $0
set mynametail = $myname:t
if ( $#argv <= 2 ) then
   echo Usage: $mynametail oldstring newstring filename ...
   exit 1
endif
set oldstring = $1
set newstring = $2
shift
shift
foreach filename ($*)
   echo ${mynametail}: about to process $filename
   ed -s $filename <<LastLine
   g/$oldstring/s//$newstring/gp
   w
   q
LastLine
end
```

Alternatively, the loop (of **raos**) could use **rao**, as follows:

```
foreach filename ($*)
   echo ${mynametail}: about to process $filename
   rao $oldstring $newstring $filename
end
```

8.4 Reading a line from standard input

A string can be read from standard input by means of the expression: **$<**

Here is an example of its use. If the destination file of a UNIX **cp** command already exists, it will be overwritten by the **cp** command. Suppose a shell script is required that asks whether the file should be overwritten. It is called just like the **cp** command:

```
nicecp fred bert
```

1 Type
showbin nicecp

You should obtain:

```
#!/bin/csh
# nicecp dxy3abc 920312
# nicecp copies the file named as the first parameter to the file named
# as the second parameter. If necessary, it asks the user to confirm
# whether he/she wants the file named as second parameter overwritten.
if ( $#argv != 2 ) then
  echo Usage: nicecp existingname nameofcopy
        exit 1
endif
if ( -f $2 ) then
  echo -n "Is it OK to overwrite $2? (type y or n): "
  set reply = $<
  if ( $reply != y ) then
    echo nicecp has not done the copying
    exit 2
  endif
endif
cp $1 $2
```

Note: the same sort of thing can be done by:

```
cp -i fred bert
```

8.5 Using the output produced by a command

The shell permits an expression that is the output produced by the execution of a command. This is known as *command substitution*. It is denoted by: **`cmd`.**

Note: the character used on both sides of **cmd** is **`**, i.e., a backquote or left-quote. This character is different from **'**, the single quote character.

For example, the command:

```
set today = `date`
```

sets the variable **today** to a string containing something like:

```
Fri Mar 13 12:53:11 GMT 1992
```

This **set** command is equivalent to a command like:

```
set today = (Fri Mar 13 12:53:11 GMT 1992)
```

This has assigned a **wordlist** to the shell variable **today**.

It is possible to access a component of a wordlist. For example, the commands:

```
echo It is `date`
set today = `date`
echo As you can see, today is $today[1] and the year is $today[6]
```

would output something like:

```
It is Fri Mar 13 14:22:59 GMT 1992
As you can see, today is Fri and the year is 1992
```

Exercise N

Produce a shell script called **nicerm** which works through the files of the current directory. It outputs the name of each file, reads a reply from the standard input, and if the reply is **y** or **Y**, it removes the file. **Note:** whilst testing this script, use something which would not cause a disaster, such as:

```
echo would remove $filename
```

rather than:

```
rm $filename
```

Note: this shell script performs a task similar to: **rm -i ***

Exercise O

Produce a shell script called **lslong** which does an **ls -l** for the files passed as parameters to the script. However, if there are no parameters, **lslong** reads a line from the user containing the names of the files that are to be passed to **ls -l**.

Try out your script by typing:

```
lslong bert jane
```

and:

```
lslong
```

Exercise P

An example of command substitution is:

```
more `ls -rt`
```

What does this command do?

Exercise Q

Produce a shell script called **sizes** that for each filename passed as a parameter displays only the filename and the size of the file in bytes. **Hint:** use the output from **ls -l**.

Try out your script by typing:

```
sizes bert jane
```

and:

```
sizes *
```

9. Decision making: using the switch command

When a shell script has to execute one of a number of alternative sequences of commands, it is often better to use a **switch** command rather than a long complex **if** command.

The script **catday** demonstrates a few things including the use of **switch** commands:

1 type

```
showbin catday
```

You should obtain the following output:

```
#!/bin/csh
# catday dxy3abc 920312
# catday outputs the day (e.g., Thu) given three parameters
# (such as 92 03 12) that give the year, month and date in month.
if ( $#argv != 3 ) then
   echo "Usage: catday year month date e.g., catday 92 05 21"
   exit 1
endif
set year = $1
set month = $2
set dateinmonth = $3

switch ($year)
   case 92:
      set lyf = 1
      set soyf = 1
      breaksw
   case 93:
      set lyf = 0
      set soyf = 3
      breaksw
   case 94:
      set lyf = 0
      set soyf = 4
      breaksw
   case 95:
      set lyf = 0
      set soyf = 5
      breaksw
   case 96:
      set lyf = 1
      set soyf = 6
      breaksw
endsw
switch ($month)
```

```
case 1:
case 01:
  set mm = 01
  set som = 1
  breaksw
case 2:
case 02:
  set mm = 02
  set som = 32
  breaksw
case 3:
case 03:
  set mm = 03
  set som = `expr $lyf + 60`
  breaksw
case 4:
case 04:
  set mm = 04
  set som = `expr $lyf + 91`
  breaksw
case 5:
case 05:
  set mm = 05
  set som = `expr $lyf + 121`
  breaksw
case 6:
case 06:
  set mm = 06
  set som = `expr $lyf + 152`
  breaksw
case 7:
case 07:
  set mm = 07
  set som = `expr $lyf + 182`
  breaksw
case 8:
case 08:
  set mm = 08
  set som = `expr $lyf + 213`
  breaksw
case 9:
case 09:
  set mm = 09
  set som = `expr $lyf + 244`
  breaksw
case 10:
  set mm = 10
  set som = `expr $lyf + 274`
  breaksw
case 11:
  set mm = 11
```

```
      set som = `expr $lyf + 305`
      breaksw
   case 12:
      set mm = 12
      set som = `expr $lyf + 335`
      breaksw
endsw
switch ($dateinmonth)
   case 1:
   case 2:
   case 3:
   case 4:
   case 5:
   case 6:
   case 7:
   case 8:
   case 9:
      set dateinmonth = 0$dateinmonth
      breaksw
   default:
      breaksw
endsw
@ dateinmonthnum = $som + $soyf
@ dateinmonthnum = $dateinmonthnum + $dateinmonth
@ dateinmonthnum = $dateinmonthnum % 7 + 1
set dayname = (Sun Mon Tues Wed Thu Fri Sat)
echo $dayname[$dateinmonthnum]
```

A **switch** is followed by an expression inside parentheses. This expression is sometimes called the *selector*. Following the **switch** line, there are a number of **arms**. Each arm consists of one or more **case** patterns, followed by zero or more commands which usually end in a **breaksw** command. When a **switch** command is executed, the *selector* is evaluated, and the commands of the first *arm* where the value of the selector matches the arm's **case** pattern is then executed.

If an arm is executed, and the arm does not end in a **breaksw** command, the commands of the following arm are then executed. If the value of the selector does not match any of the patterns, then the commands following **default**: (if present) are executed.

The characters ***, ?** and **[...]** have special meanings within a **case** pattern.

Notes: the above example illustrates two ways of doing arithmetic on a shell variable: either use variables that are assigned values using @ rather than **set**, or use the **expr** command. Towards the end of the script, the operator **%** is used. It gives the remainder after integer division. Then **dayname** is assigned a wordlist, and one of the components of **dayname** is output.

10. Hints on debugging shell scripts

There are several ways of debugging a shell script.

If you type:

```
csh -n ScriptFile
```

then the shell reads the file **ScriptFile** checking it for syntax errors. There is no need to supply parameters, because the commands of the shell script are not executed. If there is an error, only an error message is output - it does not tell you which line is in error.

Note: if **ScriptFile** is not in the current directory, you will have to supply the full pathname of **ScriptFile**, e.g.:

```
csh -n ~/bin/ScriptFile
```

If you type:

```
csh -nv ScriptFile
```

then each line of the shell script is output as it is syntax-checked. [The script is not executed.]

If you type:

```
csh -v ScriptFile parameter ...
```

then the script is executed. Each line of the script is output before it is executed.

The **x** option tells the shell to output each line after variable and command substitutions have taken place but before the line is executed. It can be combined with the **v** option:

```
csh -vx ScriptFile parameter ...
```

However, such a command can produce a large amount of output, especially if the script contains loops.

There is another approach to debugging scripts - it does not use these options. Instead, **echo** commands are inserted into the script at appropriate points.

For example, you might identify where in the script the execution has reached by adding lines like:

```
echo checking whether the input file exists
```

or:

```
foreach filename ($*)
  echo processing the file $filename
  ...
```

It is also useful to output the values of shell variables, especially when tricky code has been used:

```
set uid = `expr $id[1] : '.*=\(.*\)('`
echo uid has the value $uid
```

The **echo** command can also be useful when testing a shell script that could be disastrous if it goes wrong. Put **echo** at the start of a command line to prevent it doing its dastardly deed, and only remove it when you are sure it will do what you want it to do. An example is illustrated by:

```
foreach filename (*.tex)
   set root = $filename:r
   echo mv $filename $root.old
end
```

Finally, if you think that the earlier part of a script is failing to work properly, then put an **exit** command at a suitable point to stop the shell from executing the rest of the script. First, get the code prior to this **exit** command working properly, before moving the **exit** command to a later point in the script.