

Intermediate Unix practical guide



Intermediate Unix practical guide

Introduction 1

The operating system 1

This document 1

Before you start 1

Standard input and output 2

Introduction 2

Redirection to/from files 2

Filename substitution 3

Shell variables 4

Quotation marks 5

Command substitution 6

Shell scripts 6

Expressions 8

Numeric expressions 8

Pipes 9

Startup files 10

The Vi editor 11

Introduction 11

Advanced editing 12

Searching 12

Search and replace 12

Cut/copy/paste 13

Leaving Vi 13

Job control 13

Awk 14

References 16

Introduction

The operating system

The software that controls a computer is called an **operating system**. This provides all of the basic facilities that turn a machine into a useful tool. The computer user rarely comes into direct contact with the operating system. Instead they issue instructions to a **command line interpreter** (CLI) which checks what they have typed and passes the instructions on to the operating system.

The Unix operating system provides this sort of **user interface**. Users issue commands to a CLI called the **shell**, which in turn passes them on to the Unix **kernel**. Since the shell is largely independent of the kernel, the 'look and feel' of the shell can be changed without the need to modify the kernel. As a result, there are now a variety of shells available for Unix systems. The original shell, the Bourne Shell, is still used, but has been superseded by the widely used C Shell. Other shells, such as the Korn Shell, provide advanced features, but are not available on all Unix systems.

The C Shell is so named because its command structure is based on that of the C programming language. Indeed, as well as having a large array of features, it can also be used to write programs. Its functionality, as well as its availability, make it an ideal shell to learn and use.

This document

This document is an intermediate-level practical guide to Unix. It is aimed at users who have a basic knowledge of Unix (see Reference A), and would like to use Unix as tool for data manipulation.

It is designed as a tutorial; topics are illustrated with practical examples, rather than lengthy descriptions. The result should be that you get a flavour of what Unix has to offer, and hence be better able to interpret Unix reference documents, which are many and often obscure.

The document mainly covers the facilities provided by the C Shell. If you find that you will be using another shell, you will still find this course useful, if only to learn basic principles. Also, some shell-independent topics are covered.

Before you start

To make best use of this document you need access to the C Shell on a Unix system. Most systems here in the University provide the C Shell as the default. You should have this shell if a '%' sign is displayed when you log in. If you find that you do not have a C Shell you can, either:

- a) Arrange to have your default shell changed to the C Shell. You can often do this yourself, but it would be best to consult the person who looks after the computer.
- b) Run the C Shell within your usual shell by typing:

csh

This should display a % prompt on your screen. If this does not seem to work, consult the person who looks after the computer.

When you have finished with a C Shell run in this way, type *exit* to get back to your normal shell. You can also type *Ctrl/d* (that is, hold down the *Ctrl* key and press the *d* key) to exit.

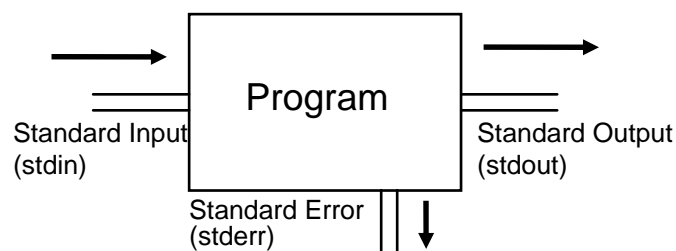
Standard input and output

Introduction

Possibly one of the most useful features of Unix is the ability to direct the flow of data easily to and from programs and commands. Each program running on a Unix system has at least three data channels associated with it. These are:

- a) standard input
- b) standard output
- c) standard error

Consider the following diagram:



A running program/command (called a process) can take input from the standard input channel. By default this channel is connected to your terminal, so when the process reads from standard input it expects data to be typed on the keyboard. Similarly, the two output channels, standard output and standard error, default to the terminal, so when the process writes data on these channels, it appears on the screen. The difference between standard output and standard error is that the latter is used to output error messages.

Redirecting these channels is easy. You can redirect them so that the process sends data to or receives data from either files or other processes. Redirecting data to other processes is covered later on. This section looks at redirecting data to and from files.

Redirection to/from files

The *date* command displays the current date and time on the screen. Try it out by typing:

```
% date
```

To redirect the output from 'date' so that it goes to a file called *date.log* rather than the screen, type:

```
% date > date.log
```

This creates (or overwrites!) the file *date.log*. To check that the file is there and it contains the right output, type:

```
% ls
% cat date.log
```

To add to the end of *date.log*, type:

```
% date >> date.log
```

Redirecting standard input employs similar notation. Try:

```
% wc < date.log
```

This reads the contents of the file *date.log* into the *wc* command. The *wc* command counts the number of lines, words and characters in an input stream, and displays these three figures on the terminal screen.

Redirecting both standard output and standard error is achieved with:

```
% date 9304281605 >& date.log
```

for example. This command attempts to change the date on the system. Obviously only the person who looks after the computer can do this so this command produces an error. This is sent on its standard error channel which here has been redirected (using the ‘&’ character) to *date.log*, along with any other output produced. Unfortunately there is no way to redirect standard output and standard error to separate files.

As mentioned above, a file is overwritten if output is directed to it using ‘>’ (not ‘>>’). To prevent this, type:

```
% set noclobber
```

Having set ‘noclobber’, if you now type:

```
% date > date.log
```

the C Shell refuses to overwrite *date.log* and displays the following error message:

```
date.log: File exists.
```

To override the ‘noclobber’ setting, type:

```
% date >! date.log
```

Filename substitution

Files on a Unix system are usually referred to explicitly by name. For example, create a set of five files by typing:

```
% touch graph.c graph.o p1 project.c project.o
```

Often, though, it is useful to be able to refer to a group of files, without having to type all of their names. This can be done using **filename substitution**, or **wildcarding**, as it is sometimes called. To get a long listing of all files with names starting with the letter ‘p’, type:

```
% ls -l p*
```

The ‘*’ matches any number of characters, including none. In this case it would also have listed a file that was just called *p*. To list all files that have an ‘r’ in their name type:

```
% ls *r*
```

To perform a more specific filename substitution you can use the single character matching symbol ‘?’ . Type:

```
% file p?
```

You should get the output:

```
p1: empty
```

(The *file* command describes the contents of a file. In this case *p1* is empty.) Notice that the files *project.c* and *project.o* do not match this filename substitution.

Other filename substitution facilities allow you to list choices. For example, the command:

```
% file project.[co]
```

matches the files *project.c* and *project.o*. Also:

```
% ls -l {project,graph}.c
```

selects *project.c* and *graph.c*.

Combinations of these facilities can be used. So:

```
% ls -l {project,graph}.[co]
```

```
% ls -l {project,graph}.*
```

are both valid commands.

Shell variables

Within a computer program, data can be kept in named data-stores called **variables**. These provide a convenient way of handling data that is used frequently within a program.

Since the C Shell can be used as a programming language, it has facilities for using variables. Some of these variables can be used to customise your shell, and others can be used for your own purposes.

The shell variable called 'term' is used to define your **terminal type**. This tells the C Shell what sort of terminal you have logged in on. Often this is a vt100. To set your terminal type variable to vt100, type:

```
% set term=vt100
```

Notice that the variable's **name** is 'term', and its **value** is 'vt100'.

Setting your own variables is just as easy as this. To create a variable called 'name' and give it the value 'Bill Smith', type:

```
% set name="Bill Smith"
```

If Bill Smith works in office 1.2L, you could type:

```
% set office=1.2L
```

To display a list of all of your shell variables, both those that have special meaning and those that are for your own use, type:

```
% set
```

To look at the value of a particular variable, you can use the *echo* command. So, to look at the value of 'name', type:

```
% echo $name
```

Notice that to access the value of a variable its name is preceded by a dollar sign (\$).

As well as containing single items of data, variables can contain a list of items (known as an array). The following command sets up a list of words in a variable called 'fruit':

```
% set fruit=(apple banana pear orange)
```

To look at the value of a particular item in the list you can use that item's index. For example, 'banana' is the second item in the list, so you can type:

```
% echo ${fruit[2]}
```

to get the word 'banana'.

You can also extract a range of items from the list. For example:

```
% echo ${fruit[1-3]}
```

prints items one, two and three.

The value of a variable can be used in any C Shell command. For example:

```
% set id="$name from office $office"
```

would set a variable called 'id' to have the value:

```
Bill Smith from office 1.2L
```

Quotation marks

The C Shell treats any set of characters delimited by spaces as a **word**. So, the sequence:

```
the cat sat on the mat
```

counts as six distinct words.

To group words together to form one big 'word' they must be surrounded by quotation marks. Hence:

```
"the cat sat on the mat"
```

would be recognised by the C Shell as just one word.

Consider the example given in the previous section where you defined a shell variable called 'name':

```
% set name="Bill Smith"
```

Here the quotes are necessary, otherwise the word Smith would have been lost.

As well as double quotes (") you can use single quotes ('). Using the example from the previous section:

```
% set name="Bill Smith"
```

```
% set office=1.2L
```

```
% set id="$name from office $office"
```

```
% echo $id
```

displays:

```
Bill Smith from office 1.2L
```

If you type:

```
% set id='$name from office $office'  
% echo $id
```

you would see:

```
$name from office $office
```

This is because the C Shell allows variable conversion within double quotes, but not within single quotes. In other words, single quotes are more 'literal' than double quotes.

Command substitution

One of the most useful features of the C Shell is the facility that allows you to make the output of one command form part of another command. This facility, called **command substitution**, is illustrated as follows:

```
% set datestamp=`date`
```

The command *date* normally displays the current date and time on the screen. Used as it has been above, the date/time is stored in the shell variable 'datestamp'. Notice that the command is enclosed in backwards quotes, not to be confused with normal single quotes.

Command substitution can be regarded as a shorthand facility. Rather than typing a series of commands to achieve a particular task, they can be combined often into just a single command. The following illustrates this:

```
% which date  
/bin/date  
% ls -l /bin/date
```

Here, you first have to find out where the *date* command resides, and then get a long listing of it. These two commands can be combined into one, thus:

```
% ls -l `which date`
```

Try these out.

Shell scripts

C Shell scripts are files that contain a sequence of commands. The file is itself run like a command and the C Shell executes (runs) each command listed in it. Using an editor (Pico, Vi), create a file called *prog* that has 3 lines as follows:

```
#!/bin/csh  
ls  
date
```

This script runs the *ls* command followed by the *date* command. The first line (the line that reads: *#!/bin/csh*) is necessary so that the system knows that this is a C Shell script.

To make the file executable type:

```
% chmod u+x prog
```

You need to do this only once for each script.

You now have a C Shell script ready to run. To run it, simply type its name:

```
% prog
```

As well as writing simple scripts like the one above, you can use the programming language features of the C Shell to write C Shell script programs. Consider the following example:

```
#!/bin/csh                                A  
#                                           B  
set names=(Jim Bob Mary Sue Jack Simon Lucy) C  
#                                           D  
echo "List of names: "                     E  
echo ""                                     F  
foreach i ($names)                         G  
    echo $i                                 H  
end                                         I  
echo ""                                     J
```

Type this script into a file called *friends* (do not type in the letters shown in the right column), make it executable and run it using the commands:

```
% pico friends  
% chmod u+x friends  
% friends
```

What the script should produce is a list of names on the screen. The list is defined in the variable called 'names'. Each item in the list is echoed to the screen from within the 'foreach loop' on lines G-I. If these three lines were written in English, they would look like:

```
For each item in the list called 'names':    G  
Display this particular item on the screen,  H  
Go back for the next one, until the end of the list. I
```

Lines that start with a '#' character are **comment** lines. These enable you to add your own notes to a shell script. For example, we could modify line B to read:

```
# This script displays a list of names
```

Expressions

When manipulating data, it is often necessary to test items of data by comparing them with other entities. In the C Shell this is achieved using expressions. Consider the following shell script:

```
#!/bin/csh                                     A
#                                               B
set names=(Jim Bob Mary Sue Jack Simon Lucy)  C
#                                               D
foreach i ($names)                             E
    if ($i == "Jack") then                       F
        echo "Found Jack."                       G
    endif                                        H
end                                              I
echo "Reached end of list."                     J
```

Type this in to a file and run it.

Here, if lines E-I were written in English, they would look like:

```
For each item in the list called 'names':      E
Is the current item "Jack"?                   F
YES: Display "Found Jack." message.          G
                                              H
Go back for the next list item, until the end. I
```

On lines F and G in the script, change the word 'Jack' to your first name. Then run the script again.

Other tests that you could use are:

```
if ($i != "Jack") then
    ...
```

to do something if Jack is *not* the current item.

```
if (($i == "Jack") || ($i == "Jill")) then
    ...
```

to do something if the current item is either Jack or Jill.

```
if (($i != "Jack") && ($i != "Jill")) then
    ...
```

to do something if the current list item is *not* Jack and is *not* Jill.

Numeric expressions

As well as using data in the form of strings (of characters) you can manipulate numbers. Like strings, numbers are stored in named variables, although the notation used is slightly different. To set a numeric variable called 'num' to have the value 1, type:

```
@ num = 1
```

However, to test the value of a numeric variable in an expression use the same notation as used with string variables. For example:

```
if ($num == 2) then
    ...
```

tests to see if the number stored in 'num' is equal to 2. The following expression is true if 'num' is greater than or equal to 1.

```
$num >= 1
```

Try the following example:

```
#!/bin/csh                                A
#                                           B
set names=(Jim Bob Mary Sue Jack Simon Lucy) C
#                                           D
@ num = 1                                  E
#                                           F
foreach i ($names)                         G
    echo "$num $i"                            H
    @ num ++                                  I
end                                          J
```

Line E initialises the numeric variable 'num' with the value 1. On line H, 'num' is used to display the 'index number' of the current item in the list. Line I increments 'num' by one. This line could have been written:

```
@ num = $num + 1
```

instead, but *@ num ++* is more efficient when incrementing by one.

When run, the script should produce the output:

```
1 Jim
2 Bob
3 Mary
4 Sue
5 Jack
6 Simon
7 Lucy
```

Pipes

Earlier input/output redirection was used to channel data between a program and a file. Another input/output facility provided by the shell allows data to be channelled between programs using what are called **pipes**. The most common use of the pipe facility is illustrated in the following command:

```
% ls | more
```

This is normally used when the output from the *ls* command is more than one screen full. The output from *ls* is piped to the *more* command which prints one screen full at a time. When this command line is typed, the shell connects the standard output of the *ls* command with the standard input of the *more* command. Any output produced by this *ls* is channelled to the *more* command, rather than going straight to the screen. The *more* command then paginates the data and displays it on the screen.

Consider the following example:

```
% cat /etc/passwd | grep Smith | sort
```

This example has a pipeline with three components. The first component *cat /etc/passwd* normally displays the entire contents of the system password file on the screen (*/etc/passwd* contains a list of people who are allowed to use the system). This list from the *cat* command is piped to

the *grep* command which picks out all entries that contain the word 'Smith'. The list is then passed on to the *sort* command which sorts it alpha- numerically and finally displays it on the screen.

Try out the following example:

```
#!/bin/csh                                     A
#                                               B
set names=(Jim Bob Mary Sue Jack Simon Lucy)   C
#                                               D
foreach i ($names)                             E
    echo $i | tr '[A-Z]' '[a-z]'              F
end                                             G
```

Line F is a pipeline comprising two commands: *echo* and *tr*. Each time through the 'foreach' loop the current name in the list is piped to the *tr* command. Here, the *tr* command translates all upper case letters to their lower case equivalent. So, you should expect to see all of the names in the list displayed in lower case, thus:

```
jim
bob
mary
sue
jack
simon
lucy
```

Startup files

When you log in, two files are used to customise your C Shell. They are your *.login* file and your *.cshrc* file. They can contain any command that you would normally type on the keyboard, but they usually contain settings like your terminal type, command path, and aliases.

The commands in your *.login* file are only executed on login. The *.cshrc* file is used every time you start a new C Shell, that is every time you log in, run a C Shell script, or type the *csh* command to run a 'sub-shell'.

Hence, all settings should go into your *.login*, except those that you want to take effect in sub-shells (including C Shell scripts). These should be added to your *.cshrc*.

The following is an example *.login*:

```
set prompt="`hostname`_! > "
set term=vt100
set history=20
set path=($path ~/bin)
alias h history
alias j jobs
```

An example *.cshrc* is:

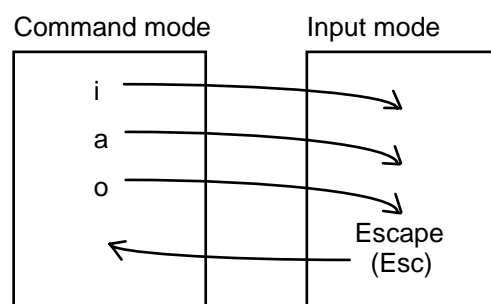
```
set name="Bill Smith"
alias graph ~/project/bin/graph
```

The Vi editor

Introduction

Vi is the standard Unix screen text editor. It is not the easiest of editors to use, but has a very comprehensive set of editing commands, and is available on all Unix systems. This section gives a brief overview of Vi. See References B and C for more information.

Vi operates in two 'modes'. The first, **command mode**, is where you type all editing commands, and the second is **insert mode**, in which you type in text/data. Throughout an editing session, you are constantly switching between these two modes. This is possibly the most cumbersome aspect of Vi and people often forget which mode they are in. The following diagram illustrates the procedure for moving back and forth between modes:



To move from command mode to insert mode, type the *i* command (press the *i* key), and to move back to command mode from insert mode, press Esc (the 'escape' key). A good tip is to press Esc when you are unsure which mode you are in. Pressing Esc always brings you into command mode. If you press Esc in command mode, nothing happens (apart from a 'beep' from the terminal).

There are a number of other ways to enter insert mode. The following table summarises these and other Vi commands:

command action

<i>i</i>	Insert characters before the cursor
<i>a</i>	Insert characters after the cursor
<i>o</i>	Open a new line after the current line
<i>O</i>	Open a new line before the current line
<i>x</i>	Delete the character under the cursor
<i>dd</i>	Delete the current line
<i>dw</i>	Delete the rest of the current word
<i>h</i>	Move one space backwards (left arrow key)
<i>l</i>	Move one space forwards (right arrow key)
<i>k</i>	Move one line upwards (up arrow key)
<i>j</i>	Move one line downwards (down arrow key)
<i>^u</i>	Scroll up (back) through the file (NB: '^u' means hold down the Ctrl key and press <i>u</i>)
<i>^d</i>	Scroll down (forward) through the file
<i>^b</i>	Jump back one page
<i>^f</i>	Jump forward one page

Advanced editing

Possibly the most common editing operations that people perform on text/data are:

- ◆ Search (for a string)
- ◆ Search and replace
- ◆ Cut, copy and paste

To try these out on a file, create an example file by typing:

```
man ls | col -b > ls.man
```

This creates a copy of the system manual entry for the *ls* command and stores it in a file called *ls.man*. Type:

```
vi ls.man
```

Move about the file using the commands mentioned above.

Remember: press Esc if you are not sure which mode you are in.

Searching

To search for the string 'list' in the file, type:

```
/list
```

You must press the Enter key after this. To search for the next occurrence of this string, type:

```
/
```

You can also type:

command	action
<code>^list</code>	Search for 'list' at the start of a line
<code>/list\$</code>	Search for 'list' at the end of a line
<code>^list\$</code>	Search for lines that only contain the word 'list'

Search and replace

To search for a string and replace it with another, type:

```
:1,$s/list/LIST/g
```

This searches all lines (*I*,*\$*) and replaces all occurrences of the string 'list' with the string 'LIST'. Notice that even words that contain the sequence 'list' are affected.

To be more selective in the search, type:

```
:g/the/s/LIST/List/g
```

Here, only lines that contain the strings 'the' are considered in the search/replace. On lines that contain 'the', the string 'LIST' is replaced with 'List'.

The 'g' flag used at the end of these commands means that all occurrences of the search string are replaced. Without it, only the first occurrence would be replaced.

Cut/copy/paste

Vi has a set of cut/copy/paste facilities which allow blocks of text to be moved around a file. For example, to move the next five lines in a file to another part of the file, use the following procedure:

- ↗ Move the cursor to the start of any block of text.
- ↗ Type *5dd*.
- ↗ Move the cursor to the new part of the file.
- ↗ Type *p*.

The command *5dd* cuts the next 5 lines from the file. The *p* command pastes these 5 lines into the new position.

To copy and paste 5 lines:

- ↗ Move the cursor to the start of any block of text.
- ↗ Type *5Y*.
- ↗ Move the cursor to the new part of the file.
- ↗ Type *p*.

Leaving Vi

There are a number of ways to get out of Vi. These include:

command	action
<code>:wq</code>	Save and quit
<code>ZZ</code>	(same as above)
<code>:q!</code>	Quit without saving changes

Job control

One of the features of Unix systems is that they are **multi-tasking**. This means that they can perform (or appear to perform) more than one task at once. The job control facility available in the C Shell allows you to make use of this feature.

In the previous section you edited a file called *ls.man*. Use Vi to edit this again, by typing:

```
vi ls.man
```

Often it is desirable to leave Vi temporarily, perform a particular task, and then resume the Vi session. The lengthy way of doing this is to quit Vi in the usual way, and then restart it after performing the task. The quickest way, however, is to use job control. To stop Vi in this way, press Ctrl/Z (hold down the Ctrl key and type z).

Immediately the word `Stopped` is printed at the bottom of the screen, followed by the shell prompt. You can now type other commands, for example:

```
ls
```

to get a directory listing.

Have a look at the status of the suspended Vi job by typing:

jobs

The message displayed should look like:

```
[1] + Stopped      vi ls.man
```

This indicates that this particular Vi session is stopped in the **background**.

When you have finished typing commands, you can resume the stopped Vi session by typing:

fg

This brings the Vi session back to the **foreground**. In this example, the Vi session was effectively frozen in its background state, and was not doing anything. Another use for job control involves making jobs run in the background. This feature is generally best suited to programs that run for a long time.

A simple example of a program that runs for a period of time is the following shell script:

```
#!/bin/csh
sleep 30
```

Type this into a file called *thirty*, and make it executable.

There are two ways to make this script run in the background. Try each of them:

- a) Type *thirty* to run it in the foreground.
Press Ctrl/Z to stop it.
Type *bg* to resume it in the background.
- b) Type *thirty &* to run it immediately in the background.

Awk

C Shell scripts are an excellent way of writing programs quickly. They give you full access to all Unix commands as well as a wide variety of programming constructs (loops, etc), and can be modified easily. Because of these advantages, there is one inherent disadvantage: speed. Each command in a shell script has to be started individually by the shell, and there is an unavoidable time overhead involved in this.

If you want to write a shell script to process data (especially a large amount) then it may be better to use the Awk utility. Awk is a pattern scanning and processing language that is designed to be able to process tables of data whose structure has some form of regular pattern to it. For this purpose, Awk has the advantage of speed over shell scripts, because it is just one command. The shell executes the Awk command, and the rest of the calculations are performed within Awk itself.

The data that you might wish to process using Awk might look something like:

	Field 1	Field 2	Field 3	...
Record 1	datum	datum	datum	...
Record 2	datum	datum	datum	...
...

Each record is on a separate line, and fields are normally separated by spaces (although another character can be used as a separator).

Once the nature of the input data has been established you can write the Awk program.

Like the C Shell, Awk has a full range of programming constructs. In Awk these constructs have been tailored for processing record-structured data. Consider the following example:

```
Mary    32    75
Alan    31    69
Richard 29    77
Fiona   26    81
Clare   34    78
```

This table of data consists of a list of five people, their age, and their score in a particular test; five records with three fields each. Type this data into a file called *test.dat*.

If you want to examine this table and simply list the names of the five participants, you could write an Awk program (script) that would look like:

```
{ print $1 }
```

Type this line into a file called *test.awk*.

To execute the Awk script on the table of data, type:

```
awk -f test.awk test.dat
```

This should display the list of names as follows:

```
Mary
Alan
Richard
Fiona
Clare
```

Notice that the *print* instruction is executed for each record in the table. Each time it prints the first field (\$1).

If you want to select a certain record from this table, you could write an Awk script like:

```
{ if ($1 == "Richard")
  print "Richard's score: " $3
}
```

Type this into a file called *test2.awk*, and execute it with:

```
awk -f test2.awk test.dat
```

Like the C Shell, you can perform numerical computations within Awk. The following Awk script totals the scores attained by each person in the test:

```
BEGIN    { print "NAME  SCORE" }           A
          { print $1 " " " $3         B
            t = t + $3                C
          }                             D
END      { print ""                   E
          print "TOTAL: " t           F
          }                             G
```

(NB: the letters at the end of each line don't form part of the Awk script).

Type this into a file called *test3.awk*, and execute it with:

```
awk -f test3.awk test.dat
```

The output should look like:

```
NAME      SCORE
Mary      75
Alan      69
Richard   77
Fiona     81
Clare     78
```

```
TOTAL:    380
```

In this Awk script there are 3 sections. The first is labelled BEGIN (line A). This section is only executed once, right at the start of the script, before any records have been processed. It is useful for displaying titles, or column headings. The second section (lines B to D) is the main section. Each instruction here is executed on each record in the data file. Here, this is the part that displays each persons score and updates the running total. The third part, labelled END (lines E to G), like the BEGIN section, is only run once. All the instructions in the END section are run after the last record has been processed. It is useful for printing out final results. Here, the total of all the scores is displayed.

For more information about Awk, see Reference D.

References

- A. Practical Introduction to Unix
Document unix-t1
- B. Using the Unix screen-based editor Vi
Document vi-r1
- C. Summary of Vi commands
Document vi-r2
- D. Awk - A Pattern Scanning and Processing Language
Unix Supplementary Documents