Chapter 6

The Shell

Introduction

You will hear many people complain that the UNIX operating system is hard to use. They are wrong. What they actually mean to say is that the UNIX command line interface is difficult to use. This is the interface that many people think is UNIX. In fact, this command line interface, provided by a program called a shell, is not the UNIX operating system and it is only one of the many different interfaces that you can use to perform tasks under UNIX. By this stage many of you will have used some of the graphical user interfaces provided by the X−Windows system.

The shell interface is a powerful tool for a Systems Administrator and one that is often used. This chapter introduces you to the shell, it's facilities and advantages. It is important to realise that the shell is just another UNIX command and that there are many different sorts of shell. The responsibilities of the shell include

- providing the command line interface
- performing I/O redirection
- performing filename substitution
- performing variable substitution
- and providing an interpreted programming language

The aim of this chapter is to introduce you to the shell and the first four of the responsibilities listed above. The interpreted programming language provided by a shell is the topic of chapter 8.

Executing Commands

As mentioned previously the commands you use such as `ls` and `cd` are stored on a UNIX computer as executable files. How are these files executed? This is one of the major responsibilities of a shell. The command line interface at which you type commands is provided by the particular shell program you are using (under Linux you will usually be using a shell called `bash`). When you type a command at this interface and hit enter the shell performs the following steps

- wait for the user to enter a command
- perform a number of tasks if the command contains any special characters
- find the executable file for the command, if the file can't be found generate an error message
- fork off a child process that will execute the command,
- wait until the command is finished (the child process dies) and then return to the top of the list

Different shells

There are many different types of shells. Table 6.1 provides a list of some of the more popular UNIX shells. Under Linux most users will be using `bash`, the **B**ourne **A**gain **Sh**ell. `bash` is an extension of the Bourne shell and uses the Bourne shell syntax. All of the examples in this text are written using the `bash` syntax.

All shells fulfil the same basic responsibilities. The main differences between shells include

- the extra features provided
  Many shells provide command history, command line editing, command completion and other special features.
- the syntax
  Different shells use slightly different syntax for some commands.

Shell	Program name	Description
Bourne shell	`sh`	the original shell from AT&T, available on all UNIX machines
C shell	`csh`	shell developed as part of BSD UNIX
Korn shell	`ksh`	AT&T improvement of the Bourne shell
Bourne again shell	`bash`	Shell distributed with Linux, version of Bourne shell that includes command line editing and other nice things

Table 6.1
Different UNIX shells

The C shell and its various versions have been popular in some fields. However, there are a number of problems with the C shell. The 85321 Website contains a pointer to a document entitled "C Shell Considered Harmful". If you really want to know why we use the Bourne shell syntax read this document.

Starting a shell

When you log onto a UNIX machine the UNIX login process automatically executes a shell for you. Which shell is executed is defined in the last field of your entry in the `/etc/passwd` file.

The last field of every line of `/etc/passwd` specifies which program to execute when the user logs in. The program is usually a shell (but it doesn't have to be).

Exercises

- What shell is started when you login?

The shell itself is just another executable program. This means you can choose to run another shell in the same way you would run any other command by simply typing in the name of the executable file. When you do the shell you are currently running will find the program and execute it.

To exit a shell any of the following may work (depending on how your environment is set up).

- logout
- exit
- CTRL−D
  By default control D is the end of file (EOF) marker in UNIX. By pressing CTRL−D you are telling the shell that it has reached the end of the file and so it exits. In a later chapter which examines shell programming you will see why shells work with files.

For example

The following is a simple example of starting other shells. Most different shells use a different command−line prompt.

```
bash$ sh
$ csh
% tcsh
> exit
%
$
bash$
```

In the above my original login shell is `bash`. A number of different shells are then started up. Each new shell in this example changes the prompt (this doesn't always happen). After starting up the `tcsh` shell I've then exited out of all the new shells and returned to the original `bash`.

Parsing the command line

The first task the shell performs when you enter a command is to parse the command line. This means the shell takes what you typed in and breaks it up into components and also changes the command−line if certain special characters exist. Special characters are used for a number of purposes and are used to modify the operation of the shell.

Table 6.2 lists most of the special characters which the shell recognises and the meaning the shell places on these characters. In the following discussion the effect of this meaning and what the shell does with these special characters will be explained in more detail.

Character(s)	Meaning
white space	Any white space characters (tabs, spaces) are used to separate arguments multiple white space characters are ignored
newline character	used to indicate the end of the command−line
’ " \	special HREF="#Quotes"§MACROBUTTON HtmlResAnchor quote characters that change the way the shell interprets special characters
&	Used after a command, tells the shell to run the command in the background
< >> << ‘ |	I/O redirection characters
* ? [ ] [^	filename substitution characters
$	indicate a shell variable
;	used to separate multiple commands on the one line

Table 6.2
Shell special characters

The Command Line

The following section examines, and attempts to explain, the special shell characters which influence the command line. This influence includes

- § breaking the command line into arguments
- § allows more than one command to a line
- § allows commands to be run in the background

Arguments

One of the first steps for the shell is to break the line of text entered by the user into arguments. This is usually the task of whitespace characters.

What will the following command display?

```
echo hello          there my friend
```

It won’t display

```
hello        there my friend
```

instead it will display

```
hello there my friend
```

When the shell examines the text of a command it divides it into the command and a list of arguments. A white space character separates the command and each argument. Any duplicate white space characters are **ignored**. The following diagram demonstrates.

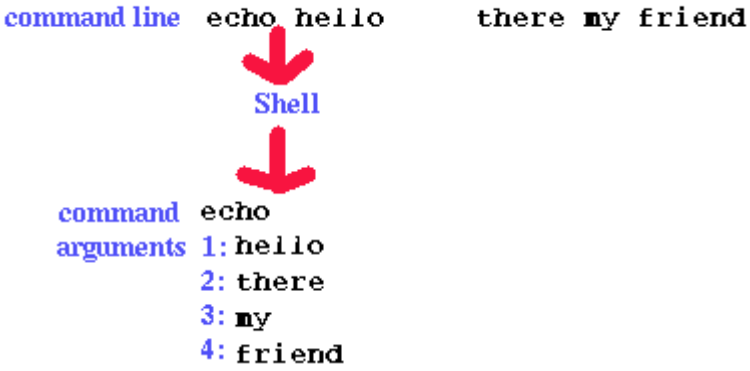

Figure 6.1
Shells, white space and arguments

Eventually the shell will execute the command. The shell passes to the command a list of arguments. The command then proceeds to perform its function. In the case above the command the user entered was the `echo` command. The purpose of the `echo` command is to display each of its arguments onto the screen separated by a single space character.

The important part here is that the `echo` command never sees all the extra space characters between `hello` and `there`. The shell removes this whilst it is performing its parsing of the command line.

One command to a line

The second shell special character in Table 6.2 is the newline character. The newline character tells the shell that the user has finished entering a command and that the shell should start parsing and then executing the command. The shell makes a number of assumptions about the command line a user has entered including

- there is only one command to each line
- the shell should not present the next command prompt until the command the user entered is finished executing.

This section examines how some of the shell special characters can be used to change these assumptions.

Multiple commands to a line

The `;` character can be used to place multiple commands onto the one line.

```
ls ; cd /etc ; ls
```

The shell sees the `;` characters and knows that this indicates the end of one command and the start of another.

Commands in the background

By default the shell will wait until the command it is running for the user has finished executing before presenting the next command line prompt. This default operation can be changed by using the `&` character. The `&` character tells the shell that it should immediately present the next command line prompt and run the command in the background.

This provides major benefits if the command you are executing is going to take a long time to complete. Running it in the background allows you to go on and perform other commands without having to wait for it to complete.

However, you won't wish to use this all the time as some confusion between the output of the command running in the background and shell command prompt can occur.

For example

The `sleep` command usually takes on argument, a number. This number represents the number of seconds the `sleep` command should wait before finishing. Try the following commands on your system to see the difference the `&` character can make.

```
bash$ sleep 10
bash$ sleep 10 &
```

Filename substitution

In the great majority of situations you will want to use UNIX commands to manipulate files and directories in some way. To make it easier to manipulate large numbers of commands the UNIX shell recognises a number of characters which should be replaced by filenames.

This process is called ether filename substitution or filename globbing.

For example

You have a directory which contains HTML files (an extension of `.html`), GIF files (an extension of `.gif`), JPEG files (an extension `.jpg`) and a range of other files. You wish to find out how big all the HTML files are.

The hard way to do this is to use the `ls –l` command and type in all the filenames.

The simple method is to use the shell special character `*`, which represents any 0 or more characters in a file name

```
ls –l *.html
```

In the above, the shell sees the `*` character and recognises it as a shell special character. The shell knows that it should replace `*.html` with any files that have filenames which match. That is, have 0 or more characters, followed by `.html`

> **UNIX doesn't use extensions**
>
> MS−DOS and Windows treat a file''s extension as special. UNIX does not do this. Refer to the previous chapter and its discussion of magic numbers.

Table 6.3 lists the other shell special characters which are used in filename substitution.

Character	What it matches
*	0 or more characters
?	1 character
[ ]	matches any one character between the brackets
[^]	matches any one character NOT in the brackets

Table 6.3
Filename substitution special characters

Some examples of filename substitution include

§ `cat *`
`*` will be replaced by the names of all the files and directories in the current directory. The `cat` command will then display the contents of all those files.

§ `ls a*bc`
`a*bc` matches all filenames that start with `a`, end with `bc` and have any characters in between.

§ `ls a?bc`
`a?bc` matches all filenames that start with `a`, end with `bc` and have only ONE character in between.

§ `ls [ic]???`
`[ic]???` matches any filename that starts with either a `i` or `c` followed by any other three letters.

§ `ls [^ic]???`
Same as the previous command but instead of any file that starts with `i` or `c` match any file that DOESN'T start with `i` or `c`.

Exercises

- Given the following files in your current directory:

```
$ ls
feb86 jan12.89
jan19.89 jan26.89
jan5.89 jan85 jan86 jan87
jan88 mar88 memo1 memo10
memo2 memo2.sv
```

What would be the output from the following commands?

```
echo *
echo *[^0-9]
echo m[a-df-z]*
echo [A-Z]*
echo jan*
echo *.*
echo ?????
echo *89
echo jan?? feb?? mar??
echo [fjm][ae][bnr]
```

Removing special meaning

There will be times when you won't want to use the shell special characters as shell special characters. For example, what happens if you really do want to display

```
hello        there my friend
```

How do you do it?

It's for circumstances like this that the shell provides shell special characters called **quotes**. The quote characters `' " \` tell the shell to ignore the meaning of any shell special character.

To display the above you could use the command

```
echo 'hello        there my friend'
```

The first quote character `'` tells the shell to ignore the meaning of any special character between it and the next `'`. In this case it will ignore the meaning of the multiple space characters. So the `echo` command receives one argument instead of four separate arguments. The following diagram demonstrates.

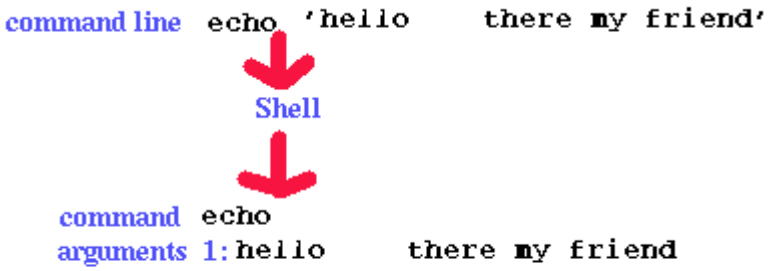

Figure 6.2
Shells, commands and quotes

Table 6.4 lists each of the shell quote characters, their names and how the influence the shell.

Character	Name	Action
'	single quote	the shell will ignore all special characters contained within a pair of single quotes
"	double quote	the shell will ignore all special characters EXCEPT $ ' \ contained within a pair of double quotes
\	backslash	the shell ignores any special character immediately following a backslash

Table 6.4
Quote characters

Examples with quotes

Try the following commands and observe what happens

- `echo I'm David.`
  This causes an error because the ' quote character must be used as one of a pair. Since this line doesn't have a second ' character the shell continues to ignore all the shell special characters it sees, **including** the new line character which indicates the end of a command.
- `echo I\'m David.`
  This is the "correct" implementation of what was attempted above. The \ quote character is used to remove the special meaning of the ' character so it is used as a normal character
- `echo *`
- `echo '*'`
- `echo \*`
  The previous three show two different approaches to removing the special meaning from a single character.
- `echo one two three four`

§ `echo 'one two three four'`

§ `echo "one two three four"`

§ `echo hello there \`
`my name is david`
Here the `\` is used to ignore the special meaning of the newline character at the end of the first line. This will only work if the newline character is immediately after the `\` character. Remember, the `\` character only removes the special meaning from the next character.

§ `echo files = ; ls`

§ `echo files = \; ls`
Since the special meaning of the `;` character is removed by the `\` character means that the shell no longer assumes there are two commands on this line. This means the `ls` characters are treated simply as normal characters, not a command which must be executed.

Exercises

- Create files with the following names
  `stars*`
  `-top`
  `hello my friend`
  `"goodbye"`
  Now delete them.
- As was mentioned in the previous chapter the `{}` and `;` used in the `exec` and `ok` actions of the `find` command must be quoted. The normal way of doing this is to use the \ character to remove the special meaning. Why doesn't the use of the single quote character work. e.g. why the following command doesn't work.
  `find . -name \*.bak -ok rm '{} ;'`

Input/output redirection

As the name suggests input/output (I/O) redirection is about changing the source of input or destination of output. UNIX I/O redirection is very similar (in part) to MS−DOS I/O redirection (guess who stole from who). I/O redirection, when combined with the UNIX philosophy of writing commands to perform one task, is one of the most important and useful combinations in UNIX.

How it works

All I/O on a UNIX system is achieved using files. This includes I/O to the screen and from a keyboard. Every process under UNIX will open a number of different files. To keep a track of the files it has, a process maintains a file descriptor for every file it is using.

File descriptors

A file descriptor is a small, non−negative integer. When a process reads/writes to/from a file it passes the kernel the file descriptor and asks it to perform the operation. The kernel knows which file the file descriptor refers to.

Standard file descriptors

Whenever the shell runs a new program (that is when it creates a new process) it automatically opens three file descriptors for the new process. These file descriptors are assigned the numbers `0`, `1` and `2` (numbers from then on are used by file descriptors the process uses). The following table summarises their names, number and default destination.

Name	File descriptor	Default destination
standard input (stdin)	0	the keyboard
standard output (stdout)	1	the screen
standard error (stderr)	2	the screen

Table 6.5
Standard file descriptors

By default whenever a command asks for input it takes that input from standard input. Whenever it produces output it puts that output onto standard output and if the command generates errors then the error messages are placed onto standard error.

For example, the ls command displays an error message when it can't find the file it was given.

```
[root@faile 85321]# ls /fred
ls: /fred: No such file or directory
```

The "No such file or directory" message is sent to standard error.

Changing direction

By using the special characters in the table below it is possible to tell the shell to change the destination for standard input, output and error.

For example

```
cat /etc/passwd > hello
```

tells the shell rather than send the contents of the `/etc/passwd` file to standard output, it should send it to a file called `hello`.

Character(s)	Result
`Command < file`	Take standard input from file
`Command > file`	Place output of `command` into `file`. Overwrite anything already in the file.
`Command >> file`	Append the output of `command` into `file`.
`command << label`	Take standard input for `command` from the following lines until a line that contains `label` by itself
`‘command‘`	execute `command` and replace `‘command‘` with the output of the command
`command1 | command2`	pass the output of `command1` to the input of `command2`
`command1 2> file`	redirect standard error of `command1` to `file`. The `2` can actually be replaced by any number which represents a file descriptor
`command1 >& file_descriptor`	redirect output of `command1` to a `file_descriptor` (the actual number for the file descriptor)

Table 6.6
I/O redirection constructs

Using standard I/O

Not all commands use standard input and standard output. For example the `cd` command doesn’t take any input and doesn’t produce any output. It simply takes the name of a directory as an argument and changes to that directory. It does however use standard error if it can’t change into the directory.

It doesn’t make sense to redirect the I/O of some commands

For example, the cp command doesn’t produce any output. It may produce errors if it cannot copy the requested file but otherwise there is no output. So

cp /etc/passwd /tmp/passwd > output.dat

does not make sense.

Filters

On the other hand some commands will always take their input from standard input and put their output onto standard output. All of the HREF="../chap2/sec4p5.html"§MACROBUTTON HtmlResAnchor filters discussed earlier in the textbook act this way.

As an example lets take the `cat` command mentioned previously. If you execute the `cat` command without supplying it with any parameters it will take its input from standard input and place its output onto standard output.

Try it. Execute the command `cat` with no arguments. Hit `CTRL-D`, on a line by itself,to signal the end of input. You should find that `cat` echoes back to the screen every line you type.

Try the same experiment with the other filters mentioned earlier.

I/O redirection examples

- `ls > the.files`
  Create a file `the.files` that contains the list of files in the current directory.
- ls /fred 2> /dev/null
  Send any error messages to the null device (throw it away).
- `cat the.files | more`
  Same effect as the command `more the.files`. Display the content of the file `the.files` one page at a time.
- `ls /etc >> the.files`
  Add the list of files in from the `/etc` directory onto the end of the file `the.files`.
- `echo number of lines in the.files = ‘wc −l the.files‘`
  Execute the command `wc −l the.files`. Replace it with its output and then execute the `echo` command. Will display output similar to `number of lines in the.files = 66`
- `cat << finished > input`
  Ask the user to type in information until they enter a line with just `finished` on it. Then copy all the information entered by the user into the file called `input`
- `cd /etc > output.file`
  Create an empty file called `output.file`. The `cd` command generates no output so redirecting its output creates an empty file.
- `ls | cd`
  An error message. `cd` doesn’t accept input so when the shell tries to send the output of the `ls` command to the `cd` command it doesn’t work.
- `echo ‘wc −l /etc/passwd‘`
  Execute the `wc` command and pass its output to the `echo` command as arguments.

Redirecting standard error

There will be times where you wish to either throw standard error away, join standard error and standard output, or just view standard error. This section provides examples of how this can be accomplished using I/O redirection.

the file `xx` doesn’t exist
display an error message on standard error

redirect standard output to the file
`errors`, no change

redirect standard error to the file
`errors` nothing on the screen

```
$ ls xx
/bin/ls: xx: No such file or
directory
```

```
$ ls xx > errors
/bin/ls: xx: No such file or
directory
```

```
$ ls xx 2> errors
```

file `chap1.ps` does exist so
we get output but the errors still go to the
file

try to send both stdout and stderr to the
`errors` file, but stdout doesn't go

try a different order and it
does work, **why?**

```
$ ls chap1.ps xx 2> errors
chap1.ps
```

```
$ ls chap1.ps xx >& 2 2> errors
chap1.ps
```

```
$ ls chap1.ps xx 2> errors >& 2
$
```

Evaluating from left to right

The shell evaluates arguments from left to right, that is it works with each argument starting with those from the left. This can influence how you might want to use the I/O redirection special characters.

For example

An example of why this is important is when you want to send both standard output and standard error of a command to the same file.

Lets say we are attempting to view the attributes of the two files `chap1.ps` and `xx`. The idea is that the file `xx` does not exist so the `ls` command will generate an error when it can't find the file. Both the error and the file attributes of the `chap1.ps` file are meant to be sent to a file called `errors`. So we try to use

- 2>&1
  This should redirect file descriptor 2 to standard output (refer back to Table 6.6). It should make standard error (file descriptor 2) go to the same place as standard output (file descriptor 1)
- > output.and.errors
  This sends standard output to the file output.and.errors (we hope).

Lets try and find out.

```
$ ls –l chap1.ps xx 2>&1  > output.and.errors
ls: xx: No such file or directory
[david@faile tmp]$ cat output.and.errors
–rw–rw–r--   1 david    david           0 Jan  9 16:23 chap1.ps
```

As you can see it doesn't work. The error message still appears on the screen and doesn't get sent to the output.and.errors file.

Can you explain why?

The reason it doesn't work is that the shell evaluates the arguments of this command from left to right. The order of evaluation goes like this

§ `ls`
  The first argument tells the shell what command should be executed.

§ `–l`
  The shell won't recognise any special characters in this argument so it will pass it on directly to the command.

§ `chap1.ps`
  Again the shell won't see any shell special characters and so passes this argument directly onto the command.

§ `xx`
  Same again.

§ `2>&1`
  Now some action. The shell recognises some special characters here. It knows that >& are I/O redirection characters. These characters tell the shell that it should redirect standard error for this command to the same place as standard output. The current location for standard output is the terminal (the screen). So standard error is redirected to the terminal. **No change from normal.**

§ `>`
  Again the shell will see a shell special character. In this case, the shell knows that standard output should be redirected to the location specified in the next argument.

§ `output.and.errors`
  This is where the shell will send the standard error of the command, a file called `output.and.errors`.

The outcome of this is that standard output still goes to the terminal and standard error goes to the file `output.and.errors`.

What we wanted is for both standard output and standard error to go to the file. The problem is the order in which the shell evaluated the arguments. The solution is to switch the I/O redirection shell characters.

```
[david@faile tmp]$ ls −l chap1.ps xx > output.and.errors 2>&1
[david@faile tmp]$ cat output.and.errors
ls: xx: No such file or directory
−rw−rw−r−−   1 david    david           0 Jan  9 16:23 chap1.ps
```

Changing the order means that standard output is redirected to the file `output.and.errors` FIRST and then standard error is redirected to where standard output is pointing (the same file).

Everything is a file

One of the features of the UNIX operating system is that almost everything can be treated as a file. This combined with I/O redirection allows you to achieve some powerful and interesting results.

You've already seen that by default stdin is the keyboard and stdout is the screen of your terminal. The UNIX operating system treats these devices as files (remember the shell sets up file descriptors for standard input/output). But which file is used?

`tty`

The `tty` command is used to display the filename of the terminal you are using.

```
$ tty
/dev/ttyp1
```

In the above example my terminal is accessed through the file `/dev/ttyp1`. This means if I execute the following command

```
cat /etc/passwd > /dev/ttyp1
```

standard output will be redirected to `/dev/ttyp1` which is where it would've gone anyway.

Exercises

- What would the following command do?
  `ls > ‘tty‘`

Device files

`/dev/ttyp1` is an example of a device file. A device file is a interface to one of the kernel’s device drivers. A device driver is a part of the Linux kernel. It knows how to talk to a specific hardware device and presents a standard programming interface that is used by software.

When you redirect I/O to/from a device file the information is passed through the device file, to the device driver and eventually to the hardware device or peripheral. In the previous example the contents of the `/etc/passwd` file were sent through the device file `/dev/ttyp1`, to a device driver. The device driver then displayed it on an appropriate device.

`/dev`

All of the system’s device files will be stored under the directory `/dev`. A standard Linux system is likely to have over 600 different device files. The following table summarises some of the device files.

filename	purpose	filename	purpose
`/dev/hda`	The first IDE disk drive	`/dev/hda1`	the first partition on the first IDE disk drive
`/dev/sda`	The first SCSI disk drive	`/dev/sda1`	the first partition on the first SCSI drive
`/dev/audio`	Sound card	`/dev/cdrom`	CD−ROM drive
`/dev/fd0`	First floppy drive	`/dev/ttyS1`	the second serial port

Table 6.7
Example device files

Redirecting I/O to device files

As you’ve seen it is possible to send output or obtain input from a device file. That particular example was fairly boring, here’s another.

```
cat HREF="../../sounds/beam.au"§MACROBUTTON HtmlResAnchor beam.au >
/dev/audio
```

This one sends a sound file to the audio device. The result (if you have a sound card) is that the sound is played.

When not to

If you examine the file permissions of the device file `/dev/hda1` you’ll find that only the `root` user and the group `disk` can write to that file. You should not be able to redirect I/O to/from that device file (unless you are the root user).

If you could it would corrupt the information on the hard−drive. There are other device files that you should not experiment with. These other device file should also be protected with appropriate file permissions.

`/dev/null`

`/dev/null` is the UNIX "garbage bin". Any output redirected to `/dev/null` is thrown away. Any input redirected from `/dev/null` is empty. `/dev/null` can be used to throw away output or create an empty file.

```
cat /etc/passwd > /dev/null
cat > newfile < /dev/null
```

The last command is one way of creating an empty file.

Exercises

- Using I/O redirection how would you perform the following tasks
  − display the first field of the `/etc/passwd` file sorted in descending order
  − find the number of lines in the `/etc/passwd` file that contain the word `bash`

Shell variables

The shell provides a variable mechanism where you can store information for future use. Shell variables are used for two main purposes: shell programming and environment control. This section provides an introduction to shell variables and their use in environment control. A later chapter discusses shell programming in more detail.

Environment control

Whenever you run a shell it creates an environment. This environment includes pre−defined shell variables used to store special values including

- the format of the prompt the shell will present to you
- your current path
- your home directory
- the type of terminal you are using
- and a great deal more.

Any shell variable you create will be stored within this environment. A later section in this chapter goes into more detail about environment control.

The `set` command

The `set` command can be used to view you shell's environment. By executing the `set` command without any parameters it will display all the shell variables currently within your shell's environment.

Using shell variables

There are two main operations performed with shell variables

- § assign a variable a value
- § use a variable's value

Assigning a value

Assigning value to a shell variable is much the same as in any programming language *variable_name=value*.

```
my_variable=hello
theNum=5
myName="David Jones"
```

A shell variable can be assigned just about any value, though there are a few guidelines to keep in mind.

A space is a shell special character. If you want your shell variable to contain a space you must tell the shell to ignore the space's special meaning. In the above example I've used the double quotes. ***For the same reason there should never be any spaces around the = symbol.***

Accessing a variable's value

To access a shell variable's value we use the `$` symbol. The `$` is a shell special character that indicates to the shell that it should replace a variable with its value.

For example

```
dinbig$ myName="David Jones"
dinbig$ echo My name is $myName
My name is David Jones
dinbig$ command=ls
dinbig$ $command
Mail ethics.txt papers
dinbig$ echo A$empty:
A:
```

Uninitialised variables

The last command in the above example demonstrates what the value of a variable is when you haven't initialised it. The last command tries to access the value for the variable `empty`.

But because the variable `empty` has never been initialised it is totally empty. Notice that the result of the command has nothing between the `A` and the `:`.

Resetting a variable

It is possible to reset the value of a variable as follows

```
myName=
```

This is totally different from trying this

```
myName=' '
```

This example sets the value of `myName` to a space character **NOT** nothing.

The `readonly` command

As you might assume the `readonly` command is used to make a shell variable readonly. Once you execute a command like

```
readonly my_variable
```

The shell variable `my_variable` can no longer be modified.

To get a list of the shell variables that are currently set to read only you run the `readonly` command without any parameters.

The `unset` command

Previously you've been shown that to reset a shell variable to nothing as follows

```
variable=
```

But what happens if you want to remove a shell variable from the current environment? This is where the `unset` command comes in. The command

```
unset variable
```

Will remove a variable completely from the current environment.

There are some restrictions on the `unset` command. You cannot use `unset` on a read only variable or on the pre−defined variables `IFS, PATH, PS1, PS2`

Arithmetic

UNIX shells do not support any notion of numeric data types such as integer or real. All shell variables are strings. How then do you perform arithmetic with shell variables?

One attempt might be

```
dinbig:~$ count=1
dinbig:~$ Rcount=$count+1
```

But it won't work. Think about what happens in the second line. The shell sees `$count` and replaces it with the value of that variable so we get the command `count=1+1`. Since the shell has no notion of an integer data type the variable `count` now takes on the value `1+1` (just a string of characters).

The `expr` command

The UNIX command `expr` is used to evaluate expressions. In particular it can be used to evaluate integer expressions. For example

```
dinbig:~$ expr 5 + 6
11
dinbig:~$ expr 10 / 5
2
dinbig:~$ expr 5 \* 10
50
dinbig:~$ expr 5 + 6 * 10
expr: syntax error
```

```
dinbig:~$ expr 5 + 6 \* 10
65
```

Note that the shell special character * has to be quoted. If it isn't the shell will replace it with the list of all the files in the current directory which results in expr generating a syntax error.

Using expr

By combining the expr command with the grave character ‘ we have a mechanism for performing arithmetic on shell variables. For example

```
count=1
count=‘expr $count + 1‘
```

expr restrictions

The expr command only works with integer arithmetic. If you need to perform floating point arithmetic have a look at the bc and awk commands.

The expr command accepts a list of parameters and then attempts to evaluate the expression they form. As with all UNIX commands the parameters for the expr command must be separated by spaces. If you don't expr interprets the input as a sequence of characters.

```
dinbig:~$ expr 5+6
5+6
dinbig:~$ expr 5+6 \* 10
expr: non-numeric argument
```

Alternatives to expr for arithmetic

The expr command is the traditional approach for perform arithmetic but it is by no means the best and has at least two major draw backs including

- it doesn't handle decimal points
  If you want to add 5.5 and 6.5 you can't do it with expr.
  One solution to this is the bc command

```
[david@faile tmp]$ echo 5.5 + 5 | bc
10.5
```

- every use requires the creation of a new process
  Chapter 5 includes a discussion of why this can be a problem and cause shell scripts to be very slow.
  An alternative to this is to use the arithemetic capabilities provided by many of the modern shells including bash. This is what is used in the add2 script mentioned in the previous chapter.

```
[david@faile tmp]$ echo $[ 5 + 5]
10
```

Valid variable names

Most programming languages have rules that restrict the format of variable names. For the Bourne shell, variable names must

§ start with either a letter or an underscore character,

§ followed by zero or more letters, numbers or underscores

{}

In some cases you will wish to use the value of a shell variable as part of a larger word. Curly braces `{ }` are used to separate the variable name from the rest of the word.

For example

You want to copy the file `/etc/passwd` into the directory `/home/david`. The following shell variables have been defined.

```
directory=/etc/
home=/home/david
```

A first attempt might be

```
cp $directorypasswd $home
```

This won't work because the shell is looking for the shell variable called `directorypasswd` (there isn't one) instead of the variable `directory`.

The correct solution would be to surround the variable name `directory` with curly braces. This indicates to the shell where the variable stops.

```
cp ${directory}passwd $home
```

Environment control

Whenever you run a shell it creates an environment in which it runs. This environment specifies various things about how the shell looks, feels and operates. To achieve this the shell uses a number of pre−defined shell variables. Table 6.8 summarises these special shell variables.

Variable name	Purpose
HOME	your home directory
SHELL	the executable program for the shell you are using
UID	your user id
USER	your username
TERM	the type of terminal you are using
DISPLAY	your X−Windows display
PATH	your executable path

Table 6.8
Environment variables

PS1 and PS2

The shell variables `PS1` and `PS2` are used to store the value of your command prompt. Changing the values of `PS1` and `PS2` will change what your command prompt looks like.

```
dinbig:~$ echo :$PS1: and :$PS2:
:\h:\w\$ : and :> :
```

PS2 is the secondary command prompt. It is used when a single command is spread over multiple lines. You can change the values of PS1 and PS2 just like you can any other shell variable.

bash extensions

You'll notice that the value of PS1 above is \h:\w\$ but my command prompt looks like dinbig:~$.

This is because the bash shell provides a number of extra facilities. One of those facilities is that it allows the command prompt to contain the hostname \h(the name of my machine) and the current working directory \w.

With older shells it was not possible to get the command prompt to display the current working directory.

- Many first time users of older shells attempt to get the command prompt to contain the current directory by trying this
  PS1='pwd'
  The pwd command displays the current working directory. Explain why this will not work. (HINT: When is the pwd command executed?)

Variables and sub−shells

Every time you start a new shell, the new shell will create a new environment separate from its parent's environment. The new shell will not be able to access or modify the environment of its parent shell.

For example

Here's a simple example.

```
dinbig:~$ myName=david                  create a shell variable
dinbig:~$ echo $myName                  use it
david
dinbig:~$ bash                          start a new shell
dinbig:~$ echo my name is $myName       try to use the parent shell's variable
my name is
dinbig:~$ exit                          exit from the new shell and return to the
                                        parent
dinbig:~$ echo $myName                  use the variable again
david
```

As you can see a new shell cannot access or modify the shell variables of its parent shells.

export

There are times when you may wish a child or sub−shell to know about a shell variable from the parent shell. For this purpose you use the export command. For example,

```
dinbig:~$ myName=David Jones
dinbig:~$ bash
dinbig:~$ echo my name is $myName
```

```
my name is
dinbig:~$ logout
dinbig:~$ export myName
dinbig:~$ bash
dinbig:~$ echo my name is $myName
my name is david
dinbig:~$ exit
```

Local variables

When you export a variable to a child shell the child shell creates a local copy of the variable. Any modification to this local variable cannot be seen by the parent process.

There is **no** way in which a child shell can modify a shell variable of a parent process. The `export` command only passes shell variables to child shells. It cannot be used to pass a shell variable from a child shell back to the parent.

For example

```
dinbig:~$ echo my name is $myName
my name is david
dinbig:~$ export myName
dinbig:~$ bash
dinbig:~$ myName=fred     # child shell modifies variable
dinbig:~$ exit
dinbig:~$ echo my name is $myName
my name is david
# there is no change in the parent
```

Advanced variable substitution

The shell provides a number of additional more complex constructs associated with variable substitution. The following table summarises them.

Construct	Purpose
${*variable*:-*value*}	replace this construct with the variable's value if it has one, if it doesn't, use *value* but don't make *variable* equal to *value*
${*variable*:=*value*}	same as the above but if *variable* has no value assign it *value*
${*variable*:?*message*}	replace the construct with the value of the variable if it has one, if it doesn't then display *message* onto stderr if *message* is null then display *prog*: *variable*: parameter null or not set on stderr
${*variable*:+*value*}	if *variable* has a value replace it with *value* otherwise do nothing

Table 6.9
Advanced variable substitution

For example

```
dinbig:~$ myName=
dinbig:~$ echo my name is $myName
my name is
dinbig:~$ echo my name is ${myName:-"NO NAME"}
my name is NO NAME
dinbig:~$ echo my name is $myName
my name is
```

```
dinbig:~$ echo my name is ${myName:="NO NAME"}
my name is NO NAME
dinbig:~$ echo my name is $myName
my name is NO NAME
dinbig:~$ herName=
dinbig:~$ echo her name is ${herName:?"she hasn't got a name"}
bash: herName: she hasn't got a name
dinbig:~$ echo her name is ${herName:?}
bash: herName: parameter null or not set
```

Evaluation order

In this chapter we've looked at the steps the shell performs between getting the user's input and executing the command. The steps include

- I/O redirection
  Where the shell changes the direction in which I/O is being sent.
- variable substitution
  The shell replaces shell variables with the corresponding values.
- filename substitution
  This is where the shell replaces globbing characters with matching filenames.

An important question is in what order does the shell perform these steps?

Why order is important

Look at the following example

```
dinbig:~$ pipe=\|
dinbig:~$ echo $pipe
|
dinbig:~$ star=\*
dinbig:~$ echo $star
Mail News README VMSpec.ps.bak acm.bhx acm2.dot
```

In the case of the `echo $start` command the shell has seen `$star` and replaced it with its value `*`. The shell sees the `*` and replaces it with the list of the files in the current directory.

In the case of the `echo $pipe` command the shell sees `$pipe` and replaces it with its value `|`. It then displays `|` onto the screen.

Why didn't it treat the `|` as a special character? If it had then `echo |` would've produced something like the following.

```
[david@faile tmp]$ echo |
>
```

The >, produced by the shell not typed in by the user, indicates that the shell is still waiting for input. The shell is still expecting another command name.

The reason this isn't produced in the previous example is related to the order in which the shell performs its analysis of shell special variables.

The order

The order in which the shell performs the steps is

- I/O redirection
- variable substitution

- filename substitution

For the command

```
echo $PIPE
```

the shell performs the following steps

- check for any I/O redirection characters, there aren't any, the command line is currently `echo $PIPE`
- check for variables, there is one `$PIPE`, replace it with its value, the command line is now `echo |`
- check for any wildcards, there aren't any

So it now executes the command `echo |`.

If you do the same walk through for the `echo $star` command you should see how its output is achieved.

The `eval` command

What happens if I want to execute the following command

```
ls $pipe more
```

using the shell variable `pipe` from the example above?

The intention is that the `pipe` shell variable should be replaced by its value `|` and that the `|` be used to redirect the output of the `ls` command to the `more` command.

Due to the order in which the shell performs its evaluation this won't work.

Doing it twice

The `eval` command is used to evaluate the command line twice. `eval` is a built−in shell command. Take the following command (using the `pipe` shell variable from above)

```
eval ls $pipe more
```

The shell sees the `$pipe` and replaces it with its value, `|`. It then executes the `eval` command.

The `eval` command repeats the shell's analysis of its arguments. In this case it will see the `|` and perform necessary I/O redirection while running the commands.

Conclusion

The UNIX command line interface is provided by programs called shells. A shell's responsibilities include

- providing the command line interface
- performing I/O redirection
- performing filename substitution
- performing variable substitution

- and providing an interpreted programming language

A shell recognises a number of characters as having special meaning. Whenever it sees these special characters it performs a number of tasks that replace the special characters.

When a shell is executed it creates an environment in which to run. This environment consists of all the shell variables created including a number of pre−defined shell variables that control its operation and appearance.

Review Questions

6.1

What is the effect of the following command sequences?

- `ls | wc −l`
- `rm ???`
- `who | wc −l`
- `mv progs/* /usr/steve/backup`
- `ls *.c | wc −l`
- `rm *.o`
- `who | sort`
- `cd ; pwd`
- `cp memo1 ..`
- `ls −l | sort +4n`

6.2

What is the output of the following commands? Are there any problems? How would you fix it?

- `echo this is a star *`
- `echo ain\\\\’t you my friend`
- `echo "** hello **"`
- `echo "the output of the ls command is ‘ls‘"`
- `echo ‘the output of the pwd command is ‘pwd‘‘`

6.3

Which of the following are valid shell variable names?

- `XxXxXxXx`
- `_`
- `12345`
- `HOMEDIR`
- `file.name`
- `_date`
- `file_name`

§ `x0-9`

§ `file1`

§ `Slimit`

6.4

Suppose your HOME directory is `/usr/steve` and that you have sub-directory as shown in figure 6.3.

Assuming you just logged onto the system and executed the following commands:

```
docs=/usr/steve/documents
let=$docs/letters
prop=$docs/proposals
```

write commands to do the following using these variables

§ List the contents of the `documents` directory

§ Copy all files from the `letters` directory to the `proposals` directory

§ Move all files with names that contain a capital letter from the `letters` directory to the current directory.

§ Count the number of files in the `memos` directory.

What would be the effect of the following commands?

§ `ls $let/..`

§ `cat $prop/sys.A >> $let/no.JSK`

§ `echo $let/*`

§ `cp $let/no.JSK $prop`

§ `cd $prop`

§ `files_in_prop=‘echo $prop*‘`

§ `cat ‘echo $let\*‘`

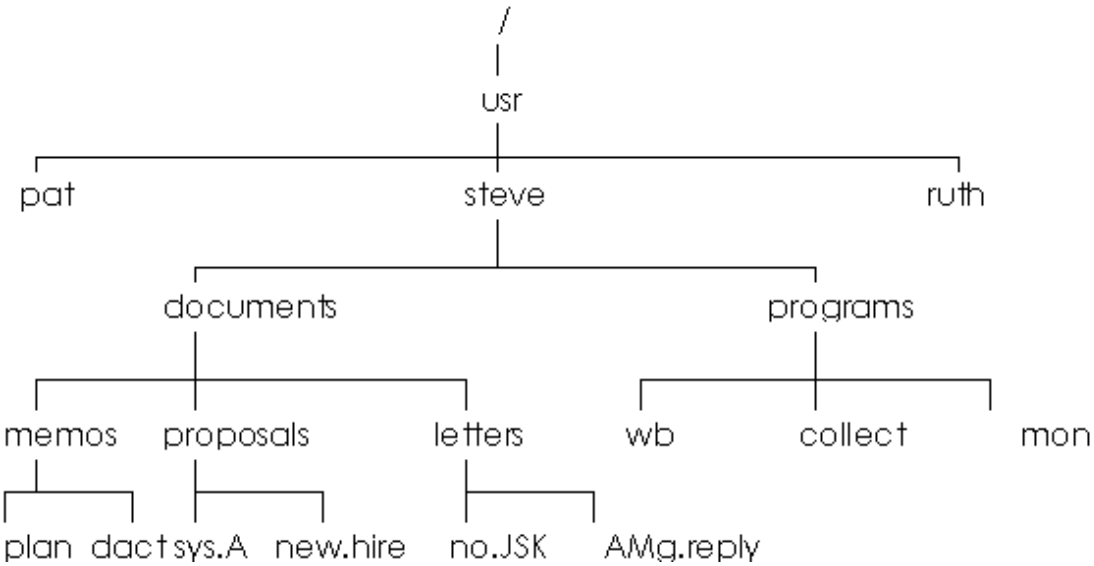

Figure 6.3

Review Question 6.4