Unix Shell Scripts

A shell script is a readable file of Unix commands that is run ("executed") by the shell, just as anything you type in would be. If you only had to type in a series of commands once, there would be no advantage in putting them in a file, but very often you do need to repeat commands, and then having them in a file saves you a lot of time, and errors. Also, the shell allows options in a script file that would be pointless if you are typing at the terminal; these options make it easy to create scripts, based on existing programs, that will handle a wide range of activities.

In this section I present some examples, in which I use `this font` for what the computer produces, and for scripts and names of files, and this font for what you type. We use *italics* to denote something that does not have to be typed in literally.

Two Simple Examples

In the lecture last time, we showed that typing

```
%cat tmp | sort | uniq -c
```

would produce a list of the words in `tmp`, with a count of the number of occurrences of each word. We can turn this into a script by creating a file whose contents are

```
#!/bin/bash
#
#   program to count word occurrences
#
cat tmp | sort | uniq -c
```

which we might call (remember this for later) `sortcount`. If we then issue the command

```
% chmod +x sortcount
```

we will have made this file (the script) executable, so that typing

```
% sortcount
```

will produce the same output as typing in the original set of commands.

To explain the script: it starts with a #, which tells the shell doing the execution to in its turn invoke the shell whose pathname is given immediately thereafter: in this case bash, the Bourne Again shell. All subsequent lines starting with # are regarded as comments and ignored. It may seem otiose to comment something as short as this, but it will help someone else to understand what it does; and as a famous quote from one of the inventors of Unix has it "A year from now, *you* are someone else".

For a second example, here is a program that you run interactively by answering questions: actually, two programs, one that writes out random numbers to a file and another that computes the statistics of numbers in a file. (These programs are not available on the machines you have access to).

At the keyboard, what you would see as you type is

```
% white
 File name and length; tmp 1000
 Type of output; gaussian
 Square root of variance; 2
      1000 terms written to file   tmp
% stats
 Data file; tmp
 Begin, end term numbers; 1 1000

 no of data          1000              mean         -0.80659E-01
 variance          3.8632              stan dev       1.9655
 min x            -6.1448              max x          6.0661
%
```

The easiest way to capture this is to use *script*, which runs as

```
% script
Script started, output file is typescript
% white
........
 min x            -6.1448              max x          6.0661
```

```
% exit
exit
Script done, output file is typescript
%
```

and captures what was typed in as a file named `typescript`. After some editing this would look like

```
#!/bin/bash
#
#  program to create random numbers and find their statistics
#
white << XXX
tmp 1000
gaussian
2
XXX
stats << XXX
tmp
1 1000
XXX
```

The usage "`<<` *string*" means "redirect standard input to read from the following lines until you see one that begins with *string*". This usage is called a "here document", and allows you to include anything you would type into a program in a script instead. It is worth developing the habit of doing this for anything you will be running more than once–indeed you should do it even for command sequences you don't expect to rerun. In this usage, any set of characters will do for the string that follows the `<<` and also ends the document; some people use `EOF`, others a single exclamation point.

Command-Line Arguments

The examples of the previous section both show scripts that do exactly one thing; however useful this is as an alternative to retyping, it probably doesn't seem that impressive. However, once we add a few additional features we can do much more.

To start with, consider our script `sortcount` from the previous section; we change it to

```
#!/bin/bash
#
#   program to count word occurrences
#
cat $1 | sort | uniq -c
```

replacing the file name `tmp` with the symbols `$1`. These symbols, when interpreted by the shell, mean "put the *string* that is the first command-line argument here"; so if we type

```
% sortcount tmp
```

we get the same response has before. But we can type

```
% sortcount file
```

where *file* is any filename, and get the same sorting and counting results for that file. So we have written a script that does a specific task on any file; if you like, a new Unix command.

In a script, the symbol combination `$n`, where n is an integer from 1 through 9, means "put the n-th command-line argument here". For example, to make `sortcount` print the n most common words from a file, we would change it to

```
#!/bin/bash
#
#   program to count word occurrences
#
cat $1 | sort | uniq -c | sort -n -r | head -$2
```

which would be invoked as, say Here we have used a second `sort` to order the results by number of occurrences (the `-n` flag) with the largest first (the `-r` flag), and piped this to the `head` command to print out only the first `$2` lines. Note that we need a `-` for the second command-line argument.

As an example of running this, I created a file `tmpa`, which is just this section of the notes modified to put one word on each line. I can then run the script `sortcount` with the following invocation and results to get the 6 commonest words:

```
% sortcount tmpa 6
     41 the
     25 to
```

```
     25 a
     17 that
     16 of
     16 and
%
```

which is pretty typical for English prose.

The case Structure

If I forget to put arguments on the command line, and just type

```
% sortcount
```

nothing happens: the computer does not even respond with a prompt. What is really happening is that the lack of an argument has caused `cat` to read from the standard input, which is to say the terminal: the computer is waiting, with infinite patience, for me to type something. And I won't see any output, no matter how many lines I type, until I type Cntrl-D to indicate what, on the terminal, corresponds to an end-of-file signal when the program reads from a file.

This is at best momentarily bewildering, and at worst leads you to think the machine is broken. We can fix the problem with one new argument, and a new piece of scripting: the `case` syntax. (I call this syntax because it is not just one line). The modified script is

```
#!/bin/bash
#
#   program to count word occurrences
#
case $# in
2)
cat $1 | sort | uniq -c | sort -nr | head -$2
;;
*)
cat << XXX
   Usage:  sortcount  file number-lines
XXX
;;
esac
```

where we have made one minor change in the line that actually does something, which is to merge the flags for the `sort` command: this is almost always allowed.

The new part of the script is the first uncommented line:

```
case $# in
```

which introduces a new variable, `$#`, which is interpreted as "the number of command-line arguments". This can be anything; the `case` and `in` are required. The structure of a `case` is

```
case variable in
choice1)
     do something
;;
choice2)
     do something else
;;
choice3)
     or something else again
;;
  more choices, perhaps
esac
```

If the *variable* matches one of the *choices*, then the material between the) that follows that *choice* and the following `;;` will be executed, and not the rest; the `esac` ends the `case`. Note that the closing parenthesis on each *choice* line is also needed.

In this example, if there are two command-line arguments, `$#` matches the first choice, and the program runs as it should. The choice * means "anything else", and causes the here document below it to be sent to the screen, so if we just type the script name, we get

```
% sortcount
   Usage:  sortcount  file number-lines
%
```

so the script has reminded us of what it is for. Based on my own experience, it is very easy to give a script a simple and deeply evocative name that, one week later, I cannot remember the meaning of. Adding this additional structure to the script means that if I just type the script name, it will remind me what it is for and how to use it.

The do Structure

Suppose we want `sortcount` to be able to combine multiple files. This is easy if we introduce the do structure. We rewrite the program as

```
#!/bin/bash
case $# in
0)
cat << XXX
   Usage:  sortcount  file  file .....
XXX
;;
*)
   for clv in $*
   do
     cat $clv | sort | uniq -c | sort -nr | head -10
   done
;;
esac
```

where I have omitted the comment lines for brevity.

Here I have made the reminder part occur only if there are no command-line variables. If there are any, then the section after the `*)` is executed. In this section, the variable `clv` is a *shell variable*; you can use any name, but `clv` is a convenient shorthand for "command-line variable". The symbols `$` mean "a list of all the character strings on the command line". The overall structure amounts to "for all entries in the list, assign the value of the entry to `clv`, and then do whatever is between the `do` and `done` lines". In this case, `clv` takes on the values of strings in the command line; for each value, the file with that name is processed. Within the script `$clv` is how we refer to the variable. Note that because we are looping over all values in the command line, we can no longer get the number of lines for `head` from there, so we have made it a fixed value.

Now running the program looks like

```
% sortcount
   Usage:  sortcount  file  file .....
% sortcount text1 text2
    410 the
    221 and
    221 a
```

```
    187 of
    175 to
    131 I
    105 that
    100 you
     96 it
     93 in
%
```

where `text1` and `text2` are files of text.