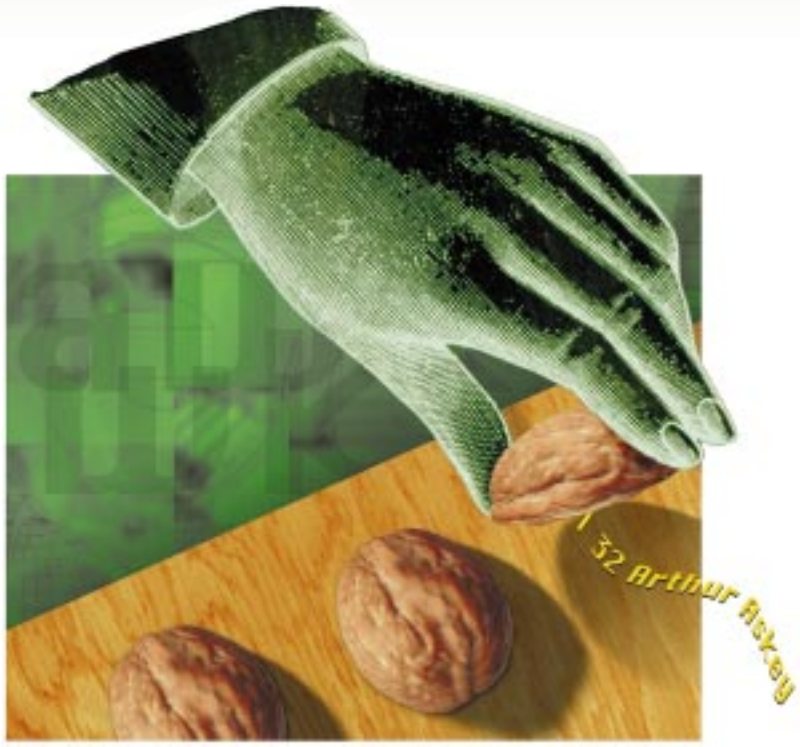
by Peter Collinson, **Hillside Systems**



MICHELLE FRIESENHAHN WILBY

Stupid Shell Tricks

Well, a piece of helpful email arrived when I was entering the panic state that my family recognizes as "the need to find something to write about." The mail message asked for help in processing some data. The data was a text file that looked like this:

```
aba1893 1 32 Arthur Askey
gcb2130 1 47 Geoffrey Boycott
dac2124 1 93 David Carter
jcc3245 1 43 John Clark
arc3337 1 39 Andrew Crewe
sjc3840 1 50 Simon Croft
afc6681 1 32 Anthony Cronshaw
```

etc.

The problem was: How could you swap the order of the text on the lines so that the full name of each person appeared at the start of each line, rather than at the end? Oh, by the way, columns one through three are *always* fixed length.

This problem can be solved a great many ways on UNIX, so I thought it would be illuminating to look at some of the different approaches and describe the little bits of underlying knowledge that you would need to solve the problem. Incidentally, while thinking about the problem, I've come to the conclusion that I didn't send the simplest method to my inquirer. But, hey, it worked. Eventually.

Your First Port of Call

If you're dealing with text in space-separated columns, then `awk` may be your first port of call. Like many UNIX tools, `awk` reads and writes text data, applying a program that specifies regular transformations to the information as it passes through. The `awk` input scanner splits each source line into a set of columns. By default a column is delineated by white space, some unknown number of spaces or tabs. A line in a basic `awk` program has the form:

match `{ commands }`

A complex program can consist of several *match* statements, each followed by a set of commands that are executed when the match succeeds. For large scripts, the commands are stored in a file and executed when needed. However, I think most people use `awk` to generate that useful single command that is entered on the command line on the fly.

The *match* part of the command is a test of some sort that is applied to the data in the current line. If the test is successful, then the commands within braces are executed; otherwise they are not. The *match* part of the `awk` command can be empty and, in this case, the commands are applied to every input line.

For our current task, we only need to know a couple of statements in the `awk` language. We are going to apply the `awk` commands to every line, so we need to know nothing about the *match* part of the program. Well, that's not quite true. The `awk` input scanner splits the lines of data into columns, and we need some way to refer to those columns in the

commands section of the program.

The authors of awk borrowed from the UNIX shell to provide a way to specify particular columns. When you supply arguments to a shell script, you can refer to them in the script using the "positional parameters" notation: The first argument is $1, the second $2 and so on. The dollar notation is used in awk to refer to a column in the input data, with $1 being the first column, $2 the second, etc.

The whole input line can be referred to using $0. So a simple awk program that just copies its input to its output is

```
$ awk '{print $0}' file
```

This is a shell "one-liner." The awk program has no match string and is quoted with single quotes to prevent the shell from attempting to expand the $0 argument. The program is a single print statement. Each line is read and then printed, and the print statement will add a new line.

We can instantly solve our problem by using the print statement and the column specification to re-order the columns in the original data:

```
$ awk '{print $4,$5,$1,$2,$3}' file
```

When the line is printed, the commas that separate the fields are replaced by the current output field separator. This will ensure that a space is placed to separate the output fields. Omitting the commas in a print statement is a common awk mistake. The language is aimed at processing strings, so it's legal to place two variables next to each other on the input line, concatenating their values. As a result, the program will still work, but the spaces that separate the columns will be stripped from the output.

If you want to do something a little more fancy with the output then you can replace the print statement with a formatted print statement:

```
$ awk '{printf "%s %s:%s:%s:%s\n",
                      $4,$5,$1,$2,$3}' file
```

I've split the line for printing purposes. You should type it in on a single line. Here, the format allows us to add colons to separate the columns on output. We must also add a new-line character (\n) to force each output line to be printed on a separate line.

You can specify the input separation character on the command line, and this can be helpful when processing colon-separated text databases such as the password file. This one-liner prints all the names from your password file, which, as you know, is a colon-separated text database:

```
$ awk -F: '{print $1}' /etc/passwd
```

Well, that solution worked OK. It would be somewhat harder to use awk if each person's name field contained more than just a first and last name. It can get tricky in awk when dealing with data that has a variable number of columns.

The Shell Solution

The shell offers a solution to our original problem too. The solution proposed here will work for the Bourne shell, sh, the Korn Shell, ksh, and GNU's bash; it will need rethinking for csh. The key to the shell solution is the read command. This shell built-in command allows us to read data and load it into a shell variable. So our simple copy input to output program becomes

```
cat file | while read line
do
    echo "$line"
done
```

Here, we are taking advantage of the ability of the shell to pipe data from a program into a while loop. Actually, we could have also said

```
while read line
do
    echo "$line"
done < file
```

which makes the while loop take its data from a file, but I find this less readable than using an extra process cat to pipe the data into the loop.

The while loop is controlled by a test that is placed after the word while on the line. The test is actually a command that is run and must complete successfully for the statements enclosed between do and done to be executed. Once the test command fails, the looping ceases. So in the example above, a successful read command will take one line of data from its input and load it into the line shell variable. If there is no data, read will fail and the loop will terminate.

The sole statement in the loop is an echo command that prints the contents of the line variable. I've quoted the argument to echo from habit. I prefer to quote shell variables that contain strings.

The read command can also be used to scan its input line, splitting the data into separate variables. The solution to our original problem is

```
cat file | while read c1 c2 c3 name
do
    echo "$name $c1 $c2 $c3"
done
```

Again, I've used quotes in the echo command to specify that the output should contain a single space between columns.

Notice that the read command has only four arguments and not five. The read command will scan the line, putting the first three words into the first three variables, and will place the remainder of the line into the last variable, name. This is a convenient feature, intended to ensure that data is never lost. I'm exploiting the feature to place the two (or more) space-separated parts of the person's name into a single variable.

The shell will normally split its input line into words using spaces, tabs and new lines to determine where each word starts and stops. You can change the separator character by meddling with the IFS shell variable. The variable con-

tains the set of characters that the shell uses to separate words; it's usually set to three default characters: space, tab and new-line. If you change the value of this variable, it's a good idea to save its original value and reset it afterwards. Here's the password file example written in a shell script:

```
oIFS="$IFS"
IFS=":"
cat /etc/passwd |
while read name rest
do
    echo "$name"
done
IFS="$oIFS"
```

The first statement saves the original version of the `IFS` variable so that we can reset it later. We then set the `IFS` variable to be just a colon character, and execute the loop. The `read` statement will now use a colon character to split the input line. The `name` variable will be set to the first colon-separated item in the password file, the login name. By setting `IFS` to just a colon character, we are assured that fields in the password file that contain embedded spaces will each be treated as a single field and will be placed in a single shell variable, should we be interested in them.

You must be careful when dealing with the `IFS` shell variable, because it's used all the time by the shell when parsing command arguments. When the shell is interpreting commands, it treats the whole line as a single string and goes through a well-defined sequence of text replacements. First, it replaces any shell variables by their value. Next, it executes any backquoted commands to load their output into the line. For example, look at the `echo` command in

```
prwd='/bin/pwd'
echo I am here: `$prwd`
```

In the `echo` command, the shell first replaces `$prwd` with its contents, creating a command enclosed in backquotes (`` `/bin/pwd` ``). Next, it looks for commands in backquotes, which of course it finds, and executes the command that the backquotes enclose. The output of the command replaces the backquoted section. In our example, the backquoted part of the line is replaced by the current working directory.

The shell now has a long string that needs to be broken up into nonblank words that each become an argument to the command that it is about to execute. The shell uses the contents of the `IFS` variable to define which characters are "blanks" and which are not. Having broken the line into several separate chunks, the shell looks for file names to expand using the standard shell `*` and `?` metacharacters. In the example, there are none and so the `echo` command will be called with four arguments: `I`, `am`, `here:` and the name of the current directory.

Now consider what happens when I say:

```
IFS=':'
prwd='/bin/pwd'
echo I am here: `$prwd`
```

Well, it might seem logical to think that the script would stop working because I have completely removed the space, tab and new-line characters from the `IFS` string. When loading commands, the shell continues to deal with the default set of characters as if they were present in the `IFS` variable. This makes sense. It allows us to set `IFS` to process data with specific separators without having to write scripts that use that particular separator in their code.

However, you *will* see different output from the `echo` command. The colon character after `here` is now a separator, and so will not be passed into the `echo` command as part of the argument list. The colon character is now behaving like a space, so it's perfectly OK, but perhaps not rational, to type

```
echo:I:am:here:`$prwd`
```

and you will see the same output as before. Being rational is a subjective judgment. I've come across people who have used the ability to set `IFS` characters to impose some other job control language style on the shell.

Incidentally, the `IFS` function has been used several times to attack and subvert shell scripts. The attacks usually center around definitions like

```
CP=/bin/cp
...
$CP old new
```

This looks very innocuous. However, because the `IFS` variable is in the environment, we can set it to something odd before we execute the script and obtain some interesting results. For example, if we call the script adding a slash character to the `IFS` string, then the `CP` variable will be defined as

```
bin cp
```

and the `$CP` line will be executed as

```
bin cp old new
```

We have suddenly introduced a new command into the script, one that the author didn't intend to be there, called `bin`. If the shell script is running as root, I can define a private program called `bin` that is actually a shell program, and with any luck I'll have an interactive shell running with superuser privileges.

The solutions are twofold: First, always place your definitions in quotes so they cannot be subverted by the `IFS` variable; second, always define a `PATH` variable that specifies a known set of search directories in publicly available scripts. I've also seen some scripts that explicitly set their `IFS` variable to ensure that what they are processing is what the author intended.

The sed Command

Let's return to the original problem. When I received the email, my first take on a solution was to use the `sed` command. The `sed` command is another UNIX supertool that reads data and transforms it using a set of commands. In the case of `sed`, the commands are based on those found in the

original UNIX editor, ed. Like ed, commands in sed have the general form:

address command

where the address field is optional. If the address field is missing, the command is applied to all the lines of data passing through the editor. In sed, the address is used to select the lines to which you wish to apply the commands. The command that I intend to use is the *substitute* command,

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

which looks on a line for old text and changes it to the new. Because we are going to swap the input data around, we will use a regular expression for matching purposes. Before we look at the matching operation, let's examine how we will create the data on the output line, taking the input data and reordering it.

The key to the reorder operation is what POSIX calls *back references*. If we place a portion of a regular expression inside the special character sequence

```
\( ... \)
```

(where the ellipsis denotes any regular expression), then we can refer to that portion of the regular expression again using a backslash followed by a single digit. So, if we create a substitute command like:

```
s/\(...\)...\(...\)/\2 \1/
```

The \1 will be replaced by the text that was matched in the first bracketed section on the left-hand side of the s command, and the \2 will be replaced by the text that was matched by the second section. Essentially, we've reordered the data on the line.

The solution to the problem is now a matter of writing a regular expression that will match the data that we want to include in each section. In fact, there are several ways to match the data in the example at the beginning of the article. I'm going to discuss two of them.

First, my correspondent pointed out that the data at the start of the line is fixed in length. In fact, there are 13 characters at the start of each line that need to be placed at the end, so we can use the repetition specification to match them:

```
.\{13\}
```

If we break this magic sequence down, we'll find a period (pronounced "dot") that will match any character. This is followed by the number 13 inside the quoted braces, which means match the previous character 13 times. Thus, the combination matches any sequence of 13 characters. This expression deals with the first part of the regular expression. The remainder of the line is of variable length, and so we can use the .* idiom, any character (dot) repeated zero or more times (star). We can now put the command together, ensuring that the pattern is anchored at the start of the line (^ the caret character) and also at the end of the line ($):

```
sed -e 's/^\(.\{13\}\)\(.*\)$/\2 \1/' file
```

Now, that's pretty unreadable, but as I've said before in previous columns, regular expressions are WORN–write once, read never. Take a deep breath and have a look at it. You should see the two sections that match the parts of the line in which we are interested, enclosed in \(...\) brackets allowing the section to be inserted on the right-hand side.

This was not the expression I used when I replied to the original mail. Repetition specifications were invented fairly recently, and so I don't readily think of them as a possibility when constructing regular expressions. Also, I was too lazy to count the number of characters at the start of each line.

I decided to match each of the columns in the data, using a square bracket repetition:

```
[^ ]*
```

Square brackets are used to match a single character position in the original data. You normally see them containing a list of characters that can appear at that position. When the list starts with a caret, it negates the list, matching any character except for the values in the list. So the list of options above matches any character except space. It's followed by a star, meaning that it will match zero or more repetitions. The combination will match any number of non-space characters.

Using this construction, we can match each individual column. But I'll leave the full solution to the problem as an exercise for the reader because the result will be too long to print here. As a hint, the password example is

```
sed -e 's/^\([^:]*\):.*$/\1/' /etc/passwd
```

The regular expression here matches:

^	The start of the line.
\(	The start of the bracketed expression.
[^:]*	Any character that is not a colon, repeated indefinitely.
\)	The end of the bracketed expression.
:	A colon.
.*	Anything else up to the end of the line.
$	The end of the line.

Finally

There are undoubtedly other ways of solving this problem. Incidentally, if you are fired by religious fervor to send me the appropriate Perl one-liner, please don't. I admit that I do tend to ignore Perl in this column because I try to talk about programs and features that everyone will find on their machine. Thanks to Hank Etlinger for the email that prompted this column.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* pc@cpg.com.