by Peter Collinson, Hillside Systems

PAUL SCHULENBURG

Creating a Personal Environment

You will undoubtedly not be too surprised when I say that UNIX is a mature system. With more than 25 years of development, it has become more and more complicated as hordes of developers and users have added their own ideas or changed things to suit their needs. UNIX is stuffed with people's ideas; some good, some bad, some indifferent. You'll often find several different ways to achieve the same goal.

Sometimes, the parallel mechanisms are there because of evolution: some new and improved method has been developed and the old one has been left in place to supply backwards compatibility. On other occasions, the parallelism has been induced by separate development: two or more sets of developers have worked independently on the same problem and arrived at a different solution. Later, these solutions are integrated, and neither has been completely eliminated–again for fear of upsetting the customer base. In other cases, the parallelism has been engendered by cross-fertilization: one group has picked up on an idea from another but implemented it slightly differently for one reason or another.

The whole edifice that is UNIX has been constructed fairly slowly from a single coherent base and makes some sort of sense if you've watched it grow. However, the result can be deeply confusing to a novice who is presented with many different options but receives no guidance in deciding which path to choose.

Making it Work Differently

A minefield of confusion is created by the ability of a UNIX user to easily modify their view of the system to suit their needs. Tailorability is something that I push heavily in these pages because computers should be used to automate repetitive tasks. The machine should work for the human, and not vice versa.

I find myself constantly fiddling with my personal command set, adding new commands and removing old ones. For example, I have recently become a full-time `emacs` user and it has the habit of creating backup files in the current directory. The backup file is created with the same name as the original file and a tilde suffix (~). The editor can also regularly create autosave files, preserving the current editing status in a file whose name starts and ends with a hash sign (#). The safety features provided by these mechanisms are a *good* thing, but having to tidy up after you've stopped using the editor is a *bad* thing. Life is full of choices.

I suppose that I could have created a script that ran every night and removed these files, but it's better to intelligently clean up when you have finished the job rather than allow a robot to delete files in the middle of the night. A backup file that has been deleted is not a great deal of use. Consequently, when I'm done with a particular project, I realized that I was in the habit of typing:

```
$ rm *~ \#*
```

I've always been queasy about combining the wildcard star (`*`) with the `rm`

command. It's just too easy to have finger trouble at the wrong moment and lose everything. The solution, I decided, was to create an alias to this command in my shell. I now have a command that performs this cleanup task using less keystrokes and, more important, that is consistently safe. It only does one well-defined and tested task, and I cannot delete all the files on which I am working.

I said, somewhat glibly, "I created an alias." Aliases are one way to change the environment to create a new command. There are other ways. I could have written a script and made a file that contains a new command. Or, I could have created a shell function to do the task. Why did I use an alias and not a script or a function? Why do we have these systems in the first place?

Shells

We need to find explanations by looking at the history of shells and the user's ability to change their environment. What follows is the sequencing of features that is my personal history. Other people will have arrived at the same point via different routes, depending on which UNIX systems they used and in what order.

The first system I used was UNIX Version 6. Its simple shell provided the familiar command structure that we have inherited today:

```
% command options list-of-files
```

Although, you should appreciate that everything on the command line *after* the name is defined and processed by the command itself. When you typed a command into the shell that was not an absolute pathname, it had the problem of finding the file that contained the command you wished to execute. It looked in standard known places–the current directory and `/bin` and `/usr/bin`–before giving up and saying `Command not found`.

Incidentally, the shell supported scripting, although there was no built-in programming syntax. All the statements in the script were commands. Essentially, the file was executed by reading the next line and running the command. However, to write programs, we need to be able to jump about the file, perhaps moving back up to create a loop, or skipping a section to allow for a test to fail. The trick was to write a command that would reset the point at which the shell read the next command by using the `seek` system call to change the kernel's idea of what was the next character to be read from the file.

For example, to jump to a point in a file, you would plant a label (a shell comment) at the appropriate point in the script and execute a `goto` command whose argument was the name of the label. The `goto` command reread the file looking for the label, and left the standard input channel so that a subsequent read would read data that followed the label. After the `goto` command exited, the shell read commands from that channel. There must have been some magic that allowed the shell to pass the command file into the `goto` command to make this happen, but I've lost those details.

Version 7

All this went away with UNIX Version 7. It had a spanking new shell and solved the problem of locating the files that correspond to commands by creating the mechanisms that we still use today. The Bell Labs development team tended to seek general solutions to problems and recognized that there were advantages to be gained by enabling processes to inherit information from their parents. When you log in, the login program knows quite a bit about you, and if it can pass that information into all the programs you run, then they can each be tailored more easily.

The solution was to add a new feature to the process model, the programmed world in which each process runs. At the moment of birth, a process inherits a set of *name=value* pairs, known as the "environment," from its parent. The environment is passed automatically from parent to child, no special action is needed to transmit it unchanged. A process can elect to alter an existing value or add a new pair, and changes to the environment are passed to all of its descendants.

One of the environment pairs, the `PATH` variable, was appropriated by shells to provide a list of places to look for commands. The initial setting of the `PATH` variable is established at login to a set of default directories. It would be something like the following:

```
PATH=:/usr/bin:/bin
```

The list is separated by colons and starts with an empty entry, which means that the shell first looks in the current directory. I should say that executing commands in the current directory is now deemed to be a bad thing because it can allow Mr. Bad Guy to place a command into your execution path.

The Version 7 shell also allowed the user to execute a start-up file, `.profile`, in their shell at login. The intention was to allow you to establish your own settings for environment variables. It was now possible to add a statement that set a new `PATH` in your `.profile` file. Note that it's important for the `.profile` file to be run by your shell and not in a subprocess, otherwise the file commands cannot create settings in your shell.

The ability to set up your own directory that could hold private commands that were accessible wherever you were in the file system tree greatly enhanced the user's ability to tailor the system. It became viable to create personal commands. The notion of having a private `bin` directory became universal.

The Version 7 shell was written by Steve Bourne and his program supported a proper programming language based on Algol 68. The language allowed complex scripts to be created. It suddenly became possible to write real commands that would combine frequently used personal command sequences. You could even decode arguments to scripts, so the command sequences could be used as templates that were applied to different files. The scripts lived in your private `bin` directory, and you could make them operate indistinguishably from the standard command set on the system.

On my site, a U.K. university, personal commands began to flourish in the early '80s. We had scripts that generated multi-column output from `ls` by piping the output to the `pr` com-

mand. We had scripts that replaced the `rm` command by something that would copy unwanted files to a temporary directory to permit recovery of the files later (just like the Windows Recycle Bin). We even had people who implemented "Are you sure?" testing for the `rm` command (what is now the `-i` option).

Personal scripts could also create a support problem. Initially, many of my scripts that were used for system purposes didn't explicitly set their `PATH` or consciously use the absolute path to standard commands. When someone executed a system script, it inherited their `PATH` and so it was possible for it to pick up a private version of `ls` or `rm`. Suddenly, the command used by the system script was not behaving "properly" and the system script was broken. There were also wonderful unused opportunities for exploiting the script to compromise the system.

Experience is a good teacher. I now know that it's really a good idea to set a `PATH` (or use an absolute pathname to a command) to make sure the script is executing the correct commands. Also, you should never have a shell script that runs as root using the setuid bit to temporarily alter permissions.

From the user's point of view, it's also not a good idea to establish a private command that replaces a standard command with something that doesn't replicate the original functionality. First, it may confuse you at some later date when you've forgotten it's there. Second, it ruins your personal portability to other systems. For example, if you create a command (or an alias) that always adds the interactive switch into the `rm` command, and start relying on it, then you may be embarrassed when it's not there.

And Then Came...

At the outset, the released version of the Bourne shell for Version 7 didn't have shell functions, nor built-in commands. Both of these features came with the release of UNIX System III, whose version of the Bourne shell supported functions and had the `echo` and `if` commands built into the code.

When System III materialized, my university had converted to using BSD systems because they supported job control. However, to use job control, you needed to run C shell (`csh`). The actual timing of the invention of `csh` by Bill Joy is not too clear. To me, it has always betrayed its origins in the Version 6 shell by being very line-based. I am told that Bill started from scratch but was influenced by the early UNIX line-based shells. I have always believed that it was a parallel development with the Bourne shell, but by the time I started using it to talk to the machine in the early '80s, it had been integrated with Version 7 and had been influenced by the Bourne shell.

However, `csh` contains considerably more internal intelligence than was present in previous shells. The folks at Bell Labs continuously accused the Berkeley team of creeping featurism, and `csh` was not lacking in features. But it didn't creep; most of the features made it faster and easier to use. Nearly all of the features are considered standard on shells today.

The intelligence `csh` possessed gave it a set of built-in commands that made some common operations happen more quickly. For example, the `echo` command was part of the shell, saving the creation of a new process for this commonly used shell print command.

There were many new features too. The shell has better variable handing capabilities in the form of arrays, and arrays are used to manage the `PATH` list. Some `csh` variables are automatically mirrored into the environment, and a change to the `path` array is automatically copied into the `PATH` environment variable with appropriate translations.

The shell also supports command history. It remembers the commands the user types and allows the user to retrieve that information. Storage of command history is a huge win, minimizing keystrokes for maximal effect.

Command aliases were introduced by `csh`. Aliases can be defined in the shell start-up files (of which more later) and are intended to allow the creation of shortcuts for commands. If you want to use a command that lives in an unusual place, but don't want to access any other commands in the target directory, then you can define an alias:

```
alias emacs /usr/local/emacs/bin/emacs
```

Now when you type the word `emacs` as a command, it's replaced by the full path. Aliases like this are a win because it means you don't have to search a directory to start a new command.

You can also use aliases to add common prefixes into commands:

```
alias rmi rm -i
```

Now when I type

```
% rmi file
```

the command that is executed is

```
% rm -i file
```

and the `-i` flag to `rm` will make it ask for confirmation before the file is deleted.

You can add wildcard expansions into aliases, but you need to quote the right-hand side of the definition to prevent the wildcards from being expanded at definition time. My desire to clean up `emacs` backup files can be specified as

```
alias cleanup 'rm *~ #*'
```

The hash (#) sign is a comment character in the shell and needs escaping in quotes to ensure that it appears in the command.

This type of aliasing feature was not available in Bourne shell but was picked up by David Korn for `ksh`, although he changed the definition syntax to add an equals sign (=):

```
alias cleanup='rm *~ #*'
```

The lack of spaces around the equals sign is significant. You'll also find this alias syntax in `bash`, GNU's Bourne Again shell.

The `csh` and `ksh` syntax I described above doesn't allow you to take an argument from the end of a command and insert it into the center of the string, creating a complex command, say,

```
% lm /bin
```

as a shorthand for

```
ls -l /bin | more
```

Of course, we want to be able to type any directory or file name instead of `/bin` and have the name pasted into the middle of the command. In fact, `csh` does permit this type of expansion by accessing its command history inside the alias definition (see *UNIX Power Tools* for more on this, a full reference can be found at the end of this article).

If you start creating aliases, then you will want to place them into your shell start-up file so that you don't have to retype them whenever you start a new shell. As we've seen, the early shells had a start-up file that was executed when you logged in, so obviously we could place the alias definitions in that. However, you'll often start new shells running from other commands, perhaps from inside your editor. These new shells will not run the start-up file because you are already logged in and so they will not see and establish the aliases.

To solve this problem, `csh` created a new "dot" file, `.cshrc`, that's run whenever a shell is started. Anything that needs to be run when you log in is placed in the `.profile` file, and anything that is needed by all shell invocations is placed into `.cshrc`. Both `ksh` and `bash` have adopted similar tactics, allowing the user to have two setup files.

When `csh` was introduced, people went mad on creating aliases, and `.cshrc` files containing hundreds of lines became common. Actually, the physical size of `.cshrc` files became a worry; significant processing time was added to the shell start-up. To counteract this, `csh` gained a flag (`-f`) that said: "Don't read the `.cshrc` file." I use aliases for commands that cannot be done any other way and for a few commands I use every day (like `h` for history). Less frequently used commands are placed in my private `bin`. Of course, with the advent of seriously fast computers, many of these speed concerns have vanished.

Functions

While `csh` was creating aliases, the Bourne shell camp was inventing functions. You'll recall that these were not present in early Version 7 releases and consequently didn't make it into Version 32V, on which the Berkeley systems are based. For some time, there were two versions of Bourne shell in common use; one with functions and one without. It took several years before shell programmers could safely write portable scripts that used shell functions.

Functions in any programming language enable the creation of a named sequence of statements that can be used and reused in the program. The statements obtain data from variables, and their values can be passed into the function as arguments. The statements may compute an answer, and the function will return it back to calling code. The function becomes a recipe for making something happen on different data.

Typing a command into any UNIX shell has similar properties to functions in a programming language. You provide a command with arguments, and it returns success or failure. Functions in the shell should behave like any command but are written in the shell's own programming language. They can then be used in any context where a regular command may appear.

Functions allow you to create a complex command definition that is eventually executed in the current shell. They are programmed in the same way as a regular command file. For example, they use the same argument-decoding syntax. With one exception (the `exit` statement), you can place script code from command files into functions with few surprises. For example, here's the `lm` command:

```
lm()
{
    ls -l "$@" | more
}
```

The `"$@"` is magic shell syntax that expands to arguments to the function to a list, while preserving any quoted arguments from the command line. We can now call this new command with any sensible set of parameters from the command line. The arguments will be passed into the `ls` command that will run with its output being sent to `more`.

The main coding difference between functions and command files is that functions must not use `exit` to force a termination of the command. Because the function is executed in the current running shell, a call to `exit` will cause the shell itself to terminate. In functions, the `return` statement is used wherever you would use an `exit` statement in the script.

Which Shall I Use?

In summary, we have three ways of creating private commands for the shell: aliases, functions and command files. I use a somewhat random mixture of all three. I have some command files in my own `bin` directory, but the number of these is small. I tend to use the `bin` directory as a place to put symbolic links to commands that I use infrequently but which live in odd places on the file system. Also, I have some symbolic links to the `rlogin` program that are named for the names of machines on my network. I can use `rlogin` to one of several machines by simply typing the machine name.

My `.bashrc` file contains several functions and aliases. I stick to the portable subset of `bash` commands so my `.bashrc` file is executable from `ksh`. Actually, on Solaris 2.6, `/bin/sh` is really `ksh`. Aliases are good for adding odd flags to standard commands to make them work the way you think they should. My functions are usually pretty complex and often can be done quite well by command files. As I mentioned above, there are whole sections on aliasing, functions and command execution in *UNIX Power Tools* by Jerry Peek, Tim O'Reilly, Mike Loukides et al. The book is into its second edition and is published by O'Reilly & Associates Inc. (ISBN 1-56592-260-3).

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`.