The Answer Girl

COMMAND LINE JUGGLER

The Linux command line can do a great deal more than the good old DOS command .com. Many of its treasures are hard to find, though, but Patricia Jung is at hand to help root them out

It's well known that typing in a command at the **shell prompt** of an **X terminal** or a console often produces more rapid results that a whole raft of mouse clicks in a GUI application. With the arrow keys, it's possible to retrieve and edit commands already entered from the History – the store for used commands – and with a Return, send them on their way again. The Tab key, for adding to commands and file names in the **bash**, is also part of the general education of a Linux user.

In most cases, that's as far as it goes. By the time you've spent five minutes digging around with up and down arrows for an old command, which it would have been quicker to type in afresh, you start wondering if the shell has any more shortcuts to offer.

Driven by this thought, the Answer Girl discovered in previous issues the event designator *!#*, which simply repeats whatever one has already entered in the current command line. With modifications such as *:1* you can restrict the selection to the second word of this expression (the count starts at zero), so that an

```
echo hello !#:1
```

executes the command *echo hello hello*. Even if one adds on another *:p* and merely outputs the command thus created (*p*rint), but does not actually allow it to proceed, this is something which is still nice to know. In reality, those of us who are less practised will usually have typed in the command completely, before remembering the very crude syntax.

Impact point

Anyone who has ever looked over the shoulder of some guru as he or she is typing may have noticed that the exclamation mark peppers the command line quite liberally, and usually in the form:

```
!commandstart
```

A *!man*, for example, recalls the manpage in which you have most recently been rummaging, while *!ssh* re-executes the most recently entered *ssh* command. It's quite likely that you won't really trust this thing, and would rather know in advance what command was going to pass. So why don't we just find out what happens when we link the modifier already mentioned *:p* with the exclamation mark search:

```
pjung@chekov:~$ !ssh :p
ssh bashir :p
pjung@bashir's password:
```

That was in fact wrong, since the target computer *bashir* is asking for the password, *:p* has apparently not stopped the last *ssh* command, *ssh bashir*, being executed. But wait! Why does it say, in the first answer line of the shell, that it is now executing the command *ssh bashir :p*? Because all we did there was to add the character string " *:p*" to the command line concerned. That may not have been what we wanted, but it's good to know it works.

The Answer Girl

The fact that the world of everyday computing, even under Linux, is often good for surprises, is a bit of a truism: Time and again things don't work, or at least not as they're supposed to. The Answer-Girl in Linux Magazine shows how to deal elegantly with such little problems.

Shell prompt The prompt for the shell. Only when a character string, which can of course include a username and/or computer name, but also the working directory and typically ends in $, > or (in the case of root) in #, can be seen on the command line, will the shell accept commands. Obviously, the prompt can be individually adapted.

Shell The command line interpreter, which prepares user commands for execution by the kernel and passes them to it. From the viewpoint of a command line user the shell encompasses the kernel like a mussel, hence the name.

Ctrl+C will in any case ensure that the wrongly invoked command is stopped. But where was the error? A simple space, because

```
pjung@chekov:~$ !ssh:p
ssh bashir :p
```

actually shows that the last-sent (as the result of our failed attempt) *ssh*-command was called *ssh bashir :p*. But we don't need the *:p*. Anyone who now carries on bravely and types in *ssh bashir*, though a little puzzled, can use

```
pjung@chekov:~$ !ssh:0-1:p
ssh bashir
```

to display the command created when we take away from the last-used *ssh*-command the reset (*ssh*) and the first word (*bashir*).

```
!ssh:0-1
```

will now call up this command line.

Glimpses of history

Unfortunately, the bash does not save the exclamation mark variations of the commands in its history like this. Instead, using the arrow keys or the shell built-in *history*, one finds only the version already replaced by the shell (the *history* function interprets a numerical argument as the number of recent commands to be listed):

```
pjung@chekov:~$ history 3
 955 date
 956 ssh bashir :p
 957 ssh bashir
```

It should be no problem to re-use the command lines output by this, using the numbers. If, on the other hand, *!#* relates to the current command line, while *!!* relates to the previous one, it seems a good idea to have a go at

```
pjung@chekov:~$ !955
date
Thu Jan  3 14:03:49 CET 2002
```

X terminal: A GUI program which provides a command line. It doesn't matter whether the program is called xterm, konsole or aterm, there is always a shell running in it too.

bash: The standard shell under Linux. Its name, "Bourne Again Shell", indicates that it is compatible with the traditional Bourne Shell, sh, but also comes with a whole lot of functionality which the other does not have.

Built-in

The work begins for the shell when the user presses Return on the command line. It checks to see if anything in what has been typed in needs to be replaced or supplemented (the exclamation mark constructs are one good example). It is only after this preliminary work that it will charge the kernel with executing the corresponding processes. The first word of the edited command line is the command which is to be started.

The bash and related shells first check to see if there is an alias of this name. If not, they check whether they are dealing with a shell function. These can be functions implemented in the shell itself, the *Shell-Built-ins*, or else they can be self-defined. Unlike aliases, functions can not only be given arguments along the way, but can also edit these. Only when the shell finds neither alias nor function does an external program come into play. The Bash-Built-in *type* gives the user the option of finding out whether a command is really an independent binary or "only" a command built into the shell. With surprising results, such as:

```
pjung@chekov:~$ type cd
cd is a shell builtin
```

So let's put it to the test: The change directory command (change directory) is not in fact an executable file, but a shell-built-in, which we can overwrite with a self-defined shell function:

```
pjung@chekov:~$ cd(){ echo Do you want to change to $1? ↵
Nothing to it... ; }
```

As in other programming languages, after the function name comes a set of round brackets as an indicator that this is a function. However, these brackets can be left empty in the bash itself, if the function deals with (command line) arguments.

Curly brackets contain the commands to be executed when the function is invoked. What matters most here is that each command must end with a semicolon, and don't forget the space after *{*. With *$1* we can go back to the first command line argument of *cd*. If we now feel the urge to change the directory, the computer digs in its heels:

```
trish@checkov:~$ cd /mnt/cdrom
Do you want to change to /mnt/cdrom? Nothing to it...
```

By way of comparison: It is not possible to evaluate the parameter variable *1* with a *cd* alias:

```
trish@linux:~$ alias cd="echo Do you want to change ↵
to $1? Nothing to it..."
trish@linux:~$ cd /tmp
Do you want to change to  ? Nothing to it... /tmp
```

Here the shell takes the entire *cd /tmp* command and does nothing but replace *cd* with *echo Do you want to change to $1? Nothing to it...*. *echo Do you want to change to $1? Nothing to it.../tmp* is executed. With *unalias cd* we cancel the alias. If we now enter the *cd* command, the shell again goes for the function defined by ourselves. To get rid of this and be able to change directory in the normal way again with the built-in, there is fortunately another built-in named *unset*: *unset cd* lays the ghost of a shell variable.

Emacs After vi, Emacs is the second most common standard text editor, installed on almost every Unix system. As with vi, there also exist various implementations of this editor, of which the most popular must be the GUI application xemacs. Anyone who has familiarised themselves with its operation, which sometimes takes quite a bit of getting used to, finds they have acquired an extremely versatile tool which can be expanded in the programming language Lisp, which, with the aid of various modules written in Emacs-Lisp, covers all possible areas of application from the programming environment to email and news programs.

Foreground process: If one calls up a command on the command line, this shell will remain blocked until this foreground command comes to an end. With command line commands such as *ls* this is normally no problem, but anyone wanting to start a GUI program will not be keen to see the shell put out of action for the duration of its use. This is why commands can be sent into the background: if you add an *&* to the command, it no longer blocks the invoking shell.

and behold, it works. The event designator does not even have to be in the first position here: *ping !956:1* for example simply grabs for itself the first argument from the 956th command in the history and thereby executes the command *ping bashir*.

Almost like Emacs

It's usually simpler if you get – as with the arrow keys – an old command on the instruction line and then you can edit this to your heart's content. In charge of this is – *man bash* gives a clue – the Readline library, which in turn ensures that **Emacs** users can use familiar Emacs key shortcuts to edit the command line. (Another possible mode, and one which can be activated in the current shell with *set +o vi*, is vi mode, which is scarcely used, even by hardcore-vi advocates.) All you have to watch out for here is the fact that not everything that you can do in an editor with expanded options is also useful for a line editor, like the one the shell offers with the command line. The Emacs mode of the shell thus covers only a tiny fraction of the options of its namesake.

But let's try a few things out. Since Emacs uses Ctrl+R to search backwards (*reverse*), we should also be able to do something in the bash with this key combination. Let's first try to ferret the *ping* command on *bashir* out of the example history. As a matter of fact a Ctrl+R *ba* produces the result:

```
(reverse-i-search)`ba': man bash
```

The last command typed in to contain the character string *ba*. Pressing the Return key to send off this command is not an option at this point, since we have not yet even found the command line we are seeking. So we complement our earlier search term *ba* with *shi*, and soon the shell suggests

```
(reverse-i-search)`bashi': ping bashir
```

If this is not to our taste, either, the Emacs cancel command Ctrl+X Ctrl+C (cancel without saving) will help out. But why do it the hard way, when there's a simpler way: a simple Ctrl+C (familiar as the key command to end **foreground processes**) works here, too.

But what can you do when the command found, although largely matching our expectations, does not do so completely? An (again, not quite conforming to Emacs) Esc makes sure that the command now found appears in the command line for editing.

The target computer is not called *bashir, but bahsir*? In Emacs Ctrl+T swaps two mixed-up letters. So place the cursor on the *h* in *bashir* and press Ctrl+T – the *h* and the *s* before it then swap places.

What works with a letter should also work with entire words. Here one can be guided by the rule of thumb that similar actions (sometimes) also have similar shortcuts: The *t* as in "*t*rade" stays, but instead of Ctrl you should press Alt. If the cursor is over *bahsir*, with an Alt+T this word trades places with its predecessor: so *ping bahsir* becomes *bahsir ping*. Another Alt+T will also swap them both back again. All you need to watch out for here is that hyphens and dots also count as "word separators": If, say, you have entered the name of a file (for example *index.html) at the prompt and you now realise that you have lazily forgotten which command to apply to it, you can write the vi* (or *emacs* or *less*...) after it:

```
pjung@chekov:~$ index.html vi
```

and press Alt+T. The result, though, is not *vi index.html*, but

```
pjung@chekov:~$ index.vi html
```

The file name ending, separated by a dot from the basic name *index*, counts as a word and is consequently swapped for the character string *vi*. This clearly mistaken swap action is one we would like to reverse. In Emacs this is done using Ctrl+X+U, and lo and behold, the bash again puts the old *index.html vi* after it for show.

Pressing the backspace key twice will now ensure that the *vi* at the end disappears again, but as soon as whole words to be eradicated start getting a bit longer, a key shortcut for deleting the word before the cursor will save a bit of strain on your wrists. So we make a proper job of it and set about searching for the corresponding key combination.

All is meta

Except, what exactly are we looking for? The *bash* manpage unfortunately does not contain such a thing as a key shortcut table. But there's something, Readline library ..., if this is responsible for manipulation options of the command line, then there should be something to find under this subject.

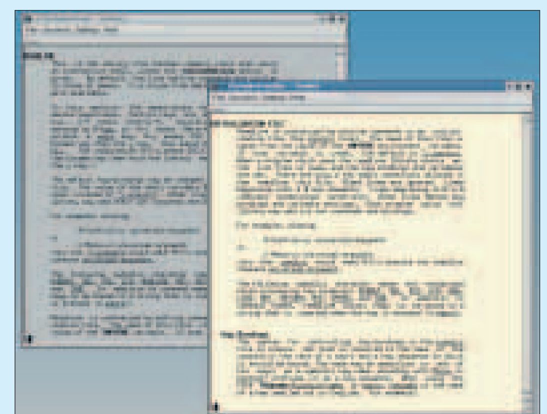

Figure 1: The READLINE section of the bash manpage (left) is almost completely cribbed from the readline-manpage

There is in fact a section called *READLINE*, which is also a neat explanation as to why it is so difficult to trawl this manpage for key shortcuts as an inexperienced user: the documentation uses Emacs syntax for its details.

This means that *C stands for Ctrl, while M* designates a mysterious *Metakey*. However, there is no such thing on PC keyboards. Depending on the pre-configuration of your computer, the Alt and/or Esc key, as mentioned in the manpage, takes over its function. And it really works: instead of Alt+T, Esc+T can swap two words, too.

However, as the manpage then makes clear, all these details are subject to change: if documented key combinations have effects different from those described, then this is presumably due to individual setting in the Readline configuration files: unless the environment variable *INPUTRC* says otherwise, the personal configuration file *~/.inputrc* goes into action. There is also the option of a global configuration file not mentioned in the Manpages of some distributions */etc/inputrc*.

Foreign characters are a matter for Readline

Inputrc? Anyone who has ever tried, in a badly pre-configured distribution, to get the accented characters on a keyboard to show up in the text console, will find this name rings a bell. Three mysterious lines (Listing 1) in the */etc/inputrc* have already helped many people at this point – but only now is it becoming clear what they mean: the Readline library can be correctly configured with the three variables set therein.

But back to business: what we are looking for is obviously a Readline command, which deletes a word backwards. In fact the seemingly-appropriate sub-section *Commands for Changing Text* has nothing that fits, but in *Killing and Yanking* ("deleting and re-inserting", where *yank* in the literal (and figurative) sense "yanks" strings already deleted from an ominous waste paper basket, the "kill ring") we get lucky:

```
backward-kill-word (M-Rubout)
   Kill the word behind the cursor. [...]
```

If only we knew what a *Rubout* key is... fortunately the very first hit in a Google search for *Rubout key* (Figure 2) informs us that it is just another name for the Backspace key. As a matter of fact Esc+Backspace works as desired – but not Alt+Backspace, which is a pity. The manpage agrees though, providing Esc as substitute for the Metakey but not the Alt which is an option in ordinary Emacs defaults.

The annoying *vi* string from the "*index.html vi*" example command line is thus gone – now we just have to get back as quickly as possible to the start of the line in order to re-insert it there with *C-y*, thus Ctrl+Y ("yank"). The appropriate Readline command is in the section *Commands for Moving* and is easy to remember with Ctrl+A. We also learn just in passing that Ctrl+E sends the cursor to the line end, like the useful option of jumping one word forward with *M-f*, and one word backward with *M-b*.

Now all that's actually missing is an overview listing all the pre-set key shortcuts followed by their meaning. This does in fact exist – but not in all distributions. Anyone who uses *man readline* to find an individual manpage on *readline(3)*, need only look in the section called *DEFAULT KEY BINDINGS*. But before the rest of you start cursing your own distributors, let me tell you: This section is almost all the *readline* manual has over the bash manual. Who has copied from whom here?

Listing 1: Accented characters

In the console, accented characters only function if the Readline variables are correctly set:

```
set meta-flag on
# The variable meta-flag now activated ensures that
# the Bash never cuts off the eighth bit of a letter.
# Namely, accented characters can only be shown in 8-bit, but not in
# 7-bit ASCII.

set output-meta on
# 8 bit characters are now shown correctly (and not as
# comical escape-sequences).

set convert-meta off
# convert-meta is activated by default and then ensures
# that 8-bit characters are converted into an escape-character and
# a 7-bit ASCII character. Foreign characters obviously get messed up
# when this happens, which is why this option should be deselected.
```

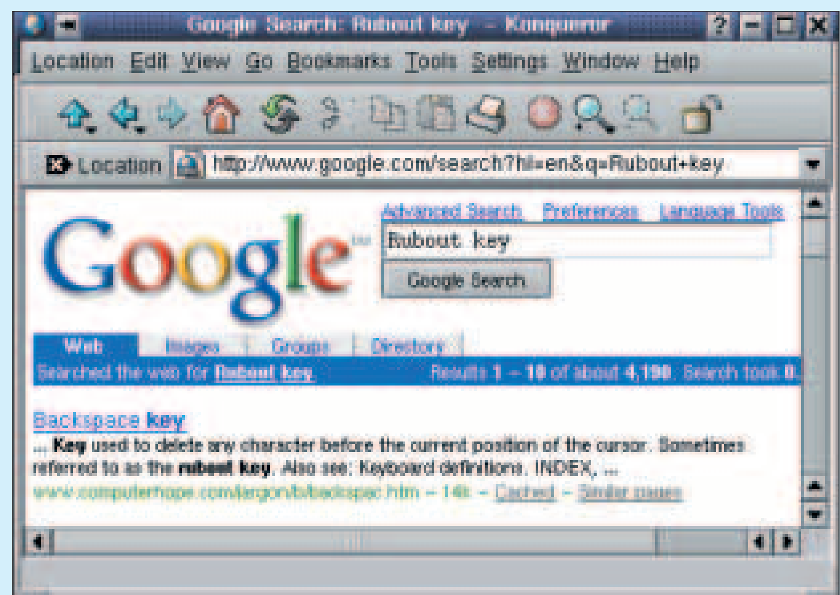

Figure 2: Rubout is just the Backspace key