



Cool Shell Tricks

The BASH shell is the product of many years of development work by a lot of people. It comes from the old days of Unix and was an important step in computer software evolution. It's a program that retains complete simplicity, yet packs in more features than most users could ever hope to use.

One of the best things about the shell is its sheer power. If you ever wonder if you can do a task differently (and more efficiently), you'll probably find that one of the many BASH developers has implemented a method to do so. Once you learn these techniques, you'll find you can whiz around the shell at blinding speed. It's just a matter of exploring the far reaches of the shell, and that's what you'll do in this chapter. Hold onto your hats, because it's an exciting ride!

Using Autocompletion

The Tab key is your best friend when using the shell, because it will cause BASH to automatically complete whatever you type. For example, if you want to run Ubuntu's web browser, you can enter `firefox` at the command line. However, to save yourself some time, you can type `fir`, and then press Tab. You'll then find that BASH fills in the rest for you. It does this by caching the names of the programs you might run according to the directories listed in your `$PATH` variable (see Chapter 13).

Of course, autocompletion has some limitations. On my Ubuntu test system, typing `loc` didn't autocomplete `locate`. Instead, it caused BASH to beep. This is because on a default Ubuntu installation, there is more than one possible match. Pressing Tab again shows those matches. Depending on how much you type (how much of an initial clue you give BASH), you might find there are many possible matches.

In this case, the experienced BASH user simply types another letter, which will be enough to distinguish the almost-typed word from the rest, and presses Tab again. With any luck, this should be enough for BASH to fill in the rest.

Autocompletion with Files and Paths

Tab autocompletion also works with files and paths. If you type the first few letters of a folder name, BASH will try to fill in the rest. This also obviously has limitations. There's no point in typing `cd myfol` and pressing Tab if there's nothing in the current directory that starts with the letters `myfol`. This particular autocomplete function works by looking at your current directory and seeing what's available.

Alternatively, you can specify an initial path for BASH to use in order to autocomplete. Typing `cd /ho` and pressing Tab will cause BASH to autocomplete the path by looking in the root directory (`/`). In other words, it will autocomplete the command with the directory `home`. In a similar way, typing `cd myfolder/myfo` will cause BASH to attempt to autocomplete by looking for a match in `myfolder`.

If you want to run a program that resides in the current directory, such as one you've just downloaded for example, typing `./`, followed by the first part of the program name, and then pressing Tab should be enough to have BASH autocomplete the rest. In this case, the dot and slash tell BASH to look in the current directory for any executable programs or scripts (programs with `X` as part of their permissions) and use them as possible autocomplete options.

BASH is clever enough to spot whether the command you're using is likely to require a file, directory, or executable, and it will autocomplete with only relevant file or directory names.

Viewing Available Options

The autocomplete function has a neat side effect. As we mentioned earlier, if BASH cannot find a match, pressing Tab again causes BASH to show all the available options. For example, typing `ba` at the shell, and then pressing Tab twice will cause BASH to show all the possible commands starting with the letters `ba`. On my test PC, this produces the following list of commands:

```
badblocks  banner  baobab  basename  bash  bashbug  batch
```

This can be a nice way of exploring what commands are available on your system. You can then use each command with the `--help` command option to find out what it does, or browse the command's man page.

When you apply this trick to directory and filename autocompletion, it's even more useful. For example, typing `cd` in a directory, and then pressing the Tab key twice will cause BASH to show the available directories, providing a handy way of retrieving a brief

directory listing. Alternatively, if you've forgotten how a directory name is spelled, you can use this technique to find out prior to switching into it.

Figure 17-1 shows a few examples of using this technique with BASH.

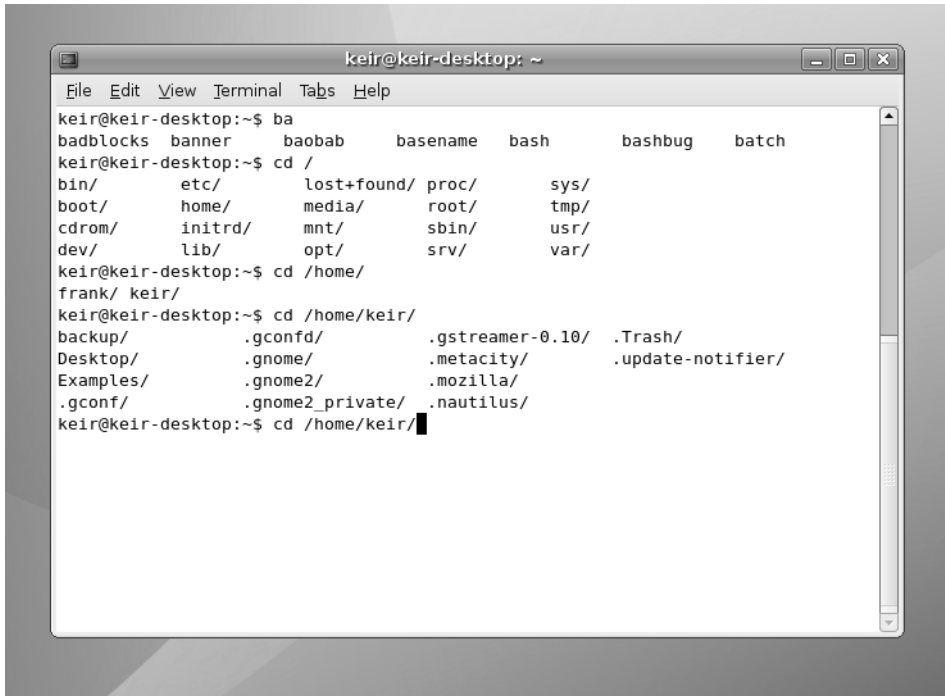


Figure 17-1. Autocompletion makes using BASH much easier.

Using Keyboard Shortcuts

Your other good friends when using BASH are the Ctrl and Alt keys. These keys provide shortcuts to vital command-line shell functions. They also let you work more efficiently when typing by providing what most programs call keyboard shortcuts.

Shortcuts for Working in BASH

Table 17-1 lists the most common keyboard shortcuts in BASH (there are many more; see BASH's man page for details). If you've explored the Emacs text editor, you might find these shortcuts familiar. Such keyboard shortcuts are largely the same across many of the software packages that originate from the GNU Project. Often, you'll find an option within many Ubuntu software packages that lets you use Emacs-style navigation, in which case, these keyboard shortcuts will most likely work equally well.

Table 17-1. *Keyboard Shortcuts in BASH*

Shortcut	Description
Navigation	
Left/right cursor key	Move left/right in text
Ctrl+A	Move to beginning of line
Ctrl+E	Move to end of line
Ctrl+right arrow	Move forward one word
Ctrl+left arrow	Move left one word
Editing	
Ctrl+U	Delete everything behind cursor to start of line
Ctrl+K	Delete from cursor to end of line
Ctrl+W	Delete from cursor to beginning of word
Alt+D	Delete from cursor to end of word
Ctrl+T	Transpose characters on left and right of cursor
Alt+T	Transpose words on left and right of cursor
Miscellaneous	
Ctrl+L	Clear screen (everything above current line)
Ctrl+U	Undo everything since last command ¹
Alt+R	Undo changes made to the line ²
Ctrl+Y	Undo deletion of word or line caused by using Ctrl+K, Ctrl+W, and so on ³
Alt+L	Lowercase current word (from the cursor to end of word)

¹ In most cases, this has the effect of clearing the line.

² This is different from Ctrl+U, because it will leave intact any command already on the line, such as one pulled from your command history.

³ This allows primitive cutting and pasting. Delete the text and then immediately undo, after which the text will remain in the buffer and can be pasted with Ctrl+Y.

Shortcuts for System Control

In terms of the control over your system offered by keyboard commands, as mentioned in Chapter 16, pressing Ctrl+Z has the effect of stopping the current program. It suspends the program until you switch back into it or tell it to resume in another way, or manually kill it.

In the same style, pressing Ctrl+C while a program is running will quit it. This sends the program's process a termination signal, a little like killing it using the `top` program. Ctrl+C can prove handy if you start a program running by accident and quickly want to end it, or if a command takes longer than you expected to work and you cannot wait for it to complete.

It's also a handy way of attempting to end crashed programs. Some complicated programs don't take too kindly to being quit in this way, particularly those that need to save data before they terminate. However, most should be okay.

Ctrl+D is another handy keyboard shortcut. This sends the program an end-of-file (EOF) message. In effect, this tells the program that you've finished your input. This can have a variety of effects, depending on the program you're running. For example, pressing Ctrl+D on its own at the shell prompt when no program is running will cause you to log out (if you're using a GUI terminal emulator like GNOME Terminal, the program will quit). This happens because pressing Ctrl+D informs the BASH shell program that you've finished your input. BASH then interprets this as the cue that it should log you out. After all, what else can it do if told there will be no more input?

While it might not seem very useful for day-to-day work, Ctrl+D is vital for programs that expect you to enter data at the command line. You might run into these as you explore BASH. If ever you read in a man page that a program requires an EOF message during input, you'll know what to press.

Using the Command History

The original hackers who invented the tools used under Unix hated waiting around for things to happen. After all, being a hacker is all about finding the most efficient way of doing any particular task.

Because of this, the BASH shell includes many features designed to optimize the user experience. The most important of these is the *command history*. BASH remembers every command you enter (even the ones that don't work!) and stores them as a list on your hard disk.

During any BASH session, you can cycle through this history using the up and down arrow keys. Pressing the up arrow key takes you back into the command history, and pressing the down arrow key takes you forward.

The potential of the command history is enormous. For example, rather than retype that long command that runs a program with command options, you can simply use the cursor keys to locate it in the history and press Enter.

Tip Typing `! -3` will cause BASH to move three paces back in the history file and run that command. In other words, it will run what you entered three commands ago.

On my Ubuntu test system, BASH remembers 1000 commands. You can view all of the remembered commands by typing `history` at the command prompt. The history list will scroll off the screen because it's so large, but you can use the scroll bars of the GNOME Terminal window to read it.

Each command in the history list is assigned a number. You can run any of the history commands by preceding their number with an exclamation mark (!), referred to as a *bang*, or sometimes a *shriek*. For example, you might type !923. On my test system, command number 923 in the BASH history is `cd ..`, so this has the effect of switching me into the parent directory.

Command numbering remains in place until you log out (close the GNOME Terminal window or end a virtual console session). After this, the numbering is reordered. There will still be 1000 commands, but the last command you entered before logging out will be at the end of the list, and the numbering will work back 1000 places until the first command in the history list.

Tip One neat trick is to type two bangs: `!!`. This tells BASH to repeat the last command you entered.

Rather than specifying a command number, you can type something like `!cd`. This will cause BASH to look in the history file, find the last instance of a command line that started with `cd`, and then run it.

Pressing `Ctrl+R` lets you search the command history from the command prompt. This particular tool can be tricky to get used to, however. As soon as you start typing, BASH will autocomplete the command based on matches found in the history file, starting with the last command in the history. What you type appears before the colon, while the auto-completion appears afterwards.

Because BASH autocompletes as you type, things can get a little confusing when you're working with the command history, particularly if it initially gets the match wrong. For example, typing `cd` will show the last instance of the use of `cd`, as in the example in Figure 17-2. This might not be what you're looking for, so you must keep typing the command you do want until it autocompletes correctly.

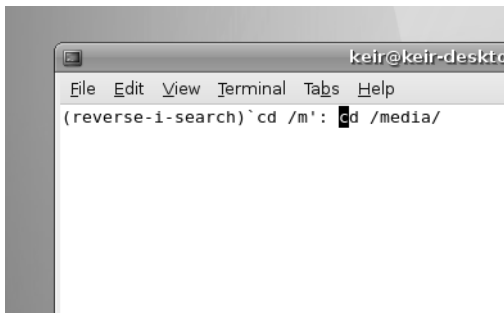


Figure 17-2. BASH history completion is very useful but can also be confusing.

Piping and Directing Output

It's not uncommon for a directory listing or output from another command to scroll off the screen. When using a GUI program like GNOME Terminal, you can use the scroll bars to view the output, but what if you are working at the bare command-line prompt?

By pressing Shift+Page Up and Shift+Page Down, you can “scroll” the window up to take a look at some of the old output, but very little is cached in this way, and you won't see more than a few screens. A far better solution is to pipe the output of the directory listing into a text viewer. Another useful technique is to redirect output to a file.

Piping the Output of Commands

Piping was one of the original innovations provided by Unix. It simply means that you can pass the output of one command to another, which is to say the output of one command can be used as input for another.

This is possible because shell commands work like machines. They usually take input from the keyboard (referred to technically as *standard input*) and, when they've done their job, usually show their output on the screen (known as *standard output*).

The commands don't need to take input from the keyboard, and they don't need to output to the screen. Piping is the process of diverting the output before it reaches the screen and passing it to another command for further processing.

Let's assume that you have a directory that is packed full of files. You want to do a long directory listing (`ls -l`) to see what permissions various files have. But doing this produces reams of output that fly off the screen. Typing something like the following provides a solution:

```
ls -l | less
```

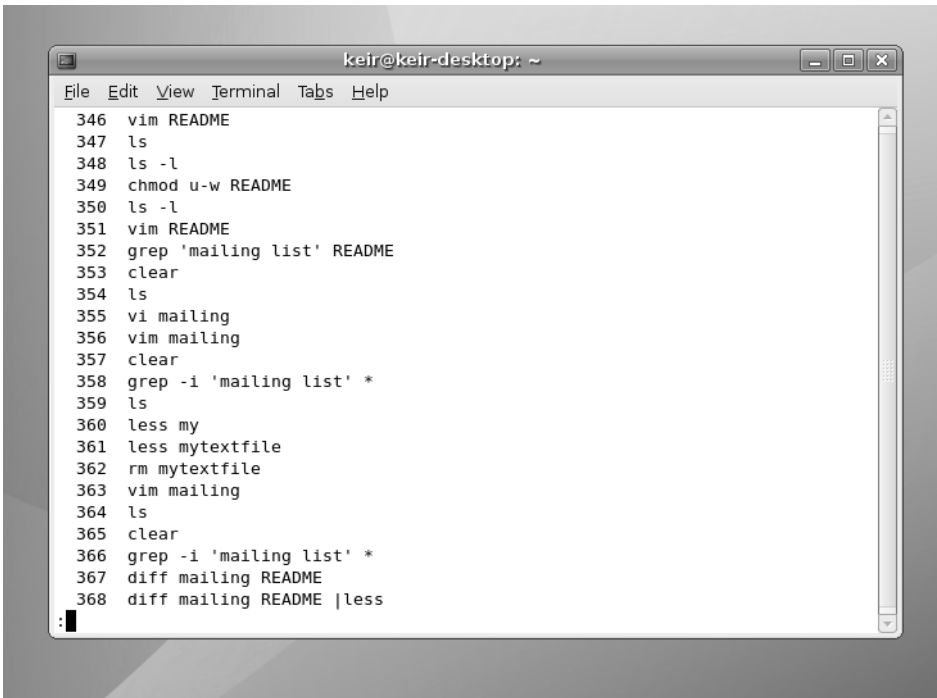
The `|` symbol between the two commands is the pipe. It can be found on most US keyboards next to the square bracket keys (near the Enter key; you'll need to hold down the Shift key to get it).

What happens in the example is that `ls -l` is run by the shell, but rather than sending the output to the screen, the pipe symbol (`|`) tells BASH to send it to the command that follows—to `less`. In other words, the listing is displayed within `less`, where you can read it at your leisure. You can use Page Up and Page Down or the arrow keys to scroll through it. Once you quit `less`, the listing evaporates into thin air; the piped output is never actually stored as a file.

In the previous section, you saw how you can use the `history` command to view the command history. At around 1000 entries, its output scrolls off the screen in seconds. However, you can pipe it to `less`, like so:

```
history | less
```

Figure 17-3 shows the result on my test PC.



```
keir@keir-desktop: ~
File Edit View Terminal Tabs Help
346 vim README
347 ls
348 ls -l
349 chmod u-w README
350 ls -l
351 vim README
352 grep 'mailing list' README
353 clear
354 ls
355 vi mailing
356 vim mailing
357 clear
358 grep -i 'mailing list' *
359 ls
360 less my
361 less mytextfile
362 rm mytextfile
363 vim mailing
364 ls
365 clear
366 grep -i 'mailing list' *
367 diff mailing README
368 diff mailing README | less
```

Figure 17-3. *Piping the output of the history command into the less command lets you read the output fully.*

You can pipe the output of any command. One of the most common uses is when searching for a particular string in the output of a command. For example, let's say you know that, within a crowded directory, there's a file with a picture of some flowers. You know that the word *flower* is in the filename, but can't recall any other details. One solution is to perform a directory listing, and then pipe the results to `grep`, which is able to search through text for a user-defined string (see Chapter 15):

```
ls -l | grep -i 'flower'
```

In this example, the shell runs the `ls -l` command, and then passes the output to `grep`. The `grep` command then searches the output for the word *flower* (the `-i` option tells it to ignore uppercase and lowercase). If `grep` finds any results, it will show them on your screen.

The key point to remember is that `grep` is used here as it normally is at the command prompt. The only difference is that it's being passed input from a previous command, rather than being used on its own.

You can pipe more than once on a command line. Suppose you know that the filename of the picture you want involves the words *flower* and *daffodil*, yet you're unsure of where they might fall in the filename. In this case, you could type the following:

```
ls -l | grep -i flower | grep -i daffodil
```

This will pass the result of the directory listing to the first `grep`, which will search the output for the word *flower*. The second pipe causes the output from `grep` to be passed to the second `grep` command, where it's then searched for the word *daffodil*. Any results are then displayed on your screen.

Redirecting Output

Redirecting is like piping, except that the output is passed to a file rather than to another command. Redirecting can also work the other way: the contents of a file can be passed to a command.

If you wanted to create a file that contained a directory listing, you could type this:

```
ls -l > directorylisting.txt
```

The angle bracket (>) between the commands tells BASH to direct the output of the `ls -l` command into a file called `directorylisting.txt`. If a file with this name exists, it's overwritten with new data. If it doesn't exist, it's created from scratch.

You can add data to an already existing file using two angle brackets:

```
ls -l >> directorylisting.txt
```

This will append the result of the directory listing to the end of the file called `directorylisting.txt`, although, once again, if the file doesn't exist, it will be created from scratch.

Redirecting output can get very sophisticated and useful. Take a look at the following:

```
cat myfile1.txt myfile2.txt > myfile3.txt
```

As you learned in Chapter 15, the `cat` command joins two or more files together. If the command were used on its own without the redirection, it would cause BASH to print `myfile1.txt` on the screen, immediately followed by `myfile2.txt`. As far as BASH is concerned, it has joined `myfile1.txt` to `myfile2.txt`, and then sent them to standard output (the screen). By specifying a redirection, you have BASH send the output to a third file. Using `cat` with redirection is a handy way of combining two files.

It's also possible to direct the contents of a file back into a command. Take a look at the following:

```
sort < textfile.txt > sortedtext.txt
```

The `sort` command simply sorts words into alphanumeric order (it actually sorts them according to the ASCII table of characters, which places symbols and numbers before

alphabetic characters). Directly after the `sort` command is a left angle bracket, which directs the contents of the file specified immediately after the bracket into the `sort` command. This is followed by a right angle bracket, which directs the output of the command into another file.

Tip To see a table of the ASCII characters, type `man ascii` at the command-line prompt.

There aren't many instances in day-to-day usage where you'll want to use the left angle bracket. It's mostly used with the text-based `mail` program (which lets you send e-mail from the shell), and in shell scripting, in which a lot of commands are combined together to form a simple program.

REDIRECTING STANDARD ERROR OUTPUT

Standard input and standard output are what BASH calls your keyboard and screen. These are the default input and output methods that programs use unless you specify something else, such as redirecting or piping output and input.

When a program goes wrong, its error message doesn't usually form part of standard output. Instead, it is output via *standard error*. Like standard output, this usually appears on the screen.

Sometimes, it's very beneficial to capture an error message in a text file. This can be done by redirecting the standard error output. The technique is very similar to redirecting standard output:

```
cdrecord --scanbus 2> errormessage.txt
```

The `cdrecord` command is used to burn CDs, and with the `--scanbus` command option, you tell it to search for CD-R/RW drives on the system, something which frequently results in an error message if your system is not properly configured.

After the initial command, you see the redirection. To redirect standard error, all you need to do is type `2>`, rather than simply `>`. This effectively tells BASH to use the second type of output: standard error.

You can direct both standard output and standard error to the same file. This is done in the following way:

```
cdrecord --scanbus > error.txt 2>&1
```

This is a little more complicated. The standard output from `cdrecord --scanbus` is sent to the file `error.txt`. The second redirect tells BASH to include standard error in the standard output. In other words, it's not a case of standard output being written to a file, and then standard error being added to it. Instead, the standard error is added to standard output by BASH, and then this is written to a file.

Summary

In this chapter, we've looked at some tricks and tips to help you use the BASH shell more effectively. You've seen how BASH can help by autocompleting commands, filenames, and directories. You also learned about keyboard shortcuts that can be used to speed up operations within the shell.

This chapter also covered the command history function and how it can be used to reuse old commands, saving valuable typing time. Finally, we looked at two key functions provided by BASH: redirection and piping. This involved the explanation of standard input, output, and error.

In Part 5 of the book, starting with the next chapter, we move on to discuss the multi-media functionality within Ubuntu.