8

ADVANCED KEYBOARD TRICKS

I often kiddingly describe Unix as “the operating system for people who like to type.” Of course, the fact that it even has a command line is a testament to that; however, command line users don’t like to type *that* much. Why else would so many commands have such short names

like `cp`, `ls`, `mv`, and `rm`? In fact, one of the most cherished goals of the command line is laziness—doing the most work with the fewest number of keystrokes. Another goal is never having to lift your fingers from the keyboard and reach for the mouse. In this chapter, we will look at `bash` features that make keyboard use faster and more efficient.

The following commands will make an appearance:

`clear` Clear the terminal screen

`history` Display or manipulate the history list

Command Line Editing

`bash` uses a library (a shared collection of routines that different programs can use) called *Readline* to implement command line editing. We have already seen some of this. We know, for example, that the arrow keys move the cursor, but there are many more features. Think of these as additional

tools that we can employ in our work. It's not important to learn all of them, but many of them are very useful. Pick and choose as desired.

NOTE

Some of the key sequences covered in this chapter (particularly those that use the ALT key) may be intercepted by the GUI for other functions. All of the key sequences should work properly when using a virtual console.

Cursor Movement

Table 8-1 describes the keys used to move the cursor.

Table 8-1: Cursor Movement Commands

Key	Action
CTRL-A	Move cursor to the beginning of the line.
CTRL-E	Move cursor to the end of the line.
CTRL-F	Move cursor forward one character; same as the right arrow key.
CTRL-B	Move cursor backward one character; same as the left arrow key.
ALT-F	Move cursor forward one word.
ALT-B	Move cursor backward one word.

Key	Action
CTRL-L	Clear the screen and move the cursor to the top-left corner. The `clear` command does the same thing.

Modifying Text

Because it's possible we might make a mistake while composing commands, we need a way to correct them efficiently. Table 8-2 describes keyboard commands that are used to edit characters on the command line.

Table 8-2: Text Editing Commands

Key	Action
CTRL-D	Delete the character at the cursor location.
CTRL-T	Transpose (exchange) the character at the cursor location with the one preceding it.
ALT-T	Transpose the word at the cursor location with the one preceding it.
ALT-L	Convert the characters from the cursor location to the end of the word to lowercase.
ALT-U	Convert the characters from the cursor location to the end of the word to uppercase.

Cutting and Pasting (Killing and Yanking) Text

The Readline documentation uses the terms *killing* and *yanking* to refer to what we would commonly call cutting and pasting (see Table 8-3). Items

that are cut are stored in a buffer (a temporary storage area in memory) called the *kill-ring*.

Table 8-3: Cut-and-Paste Commands

Key	Action
CTRL-K	Kill text from the cursor location to the end of line.
CTRL-U	Kill text from the cursor location to the beginning of the line.
ALT-D	Kill text from the cursor location to the end of the current word.
ALT-BACKSPACE	Kill text from the cursor location to the beginning of the current word. If the cursor is at the beginning of a word, kill the previous word.
CTRL-Y	Yank text from the kill-ring and insert it at the cursor location.

THE META KEY

If you venture into the Readline documentation, which can be found in the "READLINE" section of the `bash` man page, you will encounter the term *meta key*. On modern keyboards, this maps to the ALT key; however, this wasn't always the case.

Back in the dim times (before PCs but after Unix), not everybody had their own computer. What they might have had was a device called a *terminal*. A terminal was a communication device that featured a text display screen and a keyboard and just enough electronics inside to display text characters and move the cursor around. It was attached (usually by serial cable) to a larger computer or the communication network of a

larger computer. There were many different brands of terminals, and they all had different keyboards and display feature sets. Because they all tended to at least understand ASCII, software developers wanting portable applications wrote to the lowest common denominator. Unix systems have an elaborate way of dealing with terminals and their different display features. Because the developers of Readline could not be sure of the presence of a dedicated extra control key, they invented one and called it *meta*. While the ALT key serves as the meta key on modern keyboards, you can also press and release the ESC key to get the same effect as holding down the ALT key if you're still using a terminal (which you can still do in Linux!).

Completion

Another way that the shell can help you is through a mechanism called *completion*. Completion occurs when you press the TAB key while typing a command. Let's see how this works. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop   ls-output.txt  Pictures  Templates  Videos
Documents  Music         Public
```

try typing the following, but *don't press* ENTER:

```
[me@linuxbox ~]$ ls l
```

Now press TAB.

```
[me@linuxbox ~]$ ls ls-output.txt
```

See how the shell completed the line for you? Let's try another one. Again, don't press ENTER.

```
[me@linuxbox ~]$ ls D
```

Press TAB.

```
[me@linuxbox ~]$ ls D
```

No completion, just nothing. This happened because `D` matches more than one entry in the directory. For completion to be successful, the "clue" you give it has to be unambiguous. If we go further, as with the following:

```
[me@linuxbox ~]$ ls Do
```

and then press TAB:

```
[me@linuxbox ~]$ ls Documents
```

the completion is successful.

While this example shows completion of pathnames, which is its most common use, completion will also work on variables (if the beginning of the word is a `$`), usernames (if the word begins with `~`), commands (if the word is the first word on the line), and hostnames (if the beginning of the word is `@`). Hostname completion works only for hostnames listed in */etc/hosts*.

A number of control and meta key sequences are associated with completion, as listed in Table 8-4.

Table 8-4: Completion Commands

Key	Action
ALT-?	Display a list of possible completions. On most systems, you can also do this by pressing the TAB key a second time, which is much easier.
ALT-*	Insert all possible completions. This is useful when you want to use more than one possible match.

There are quite a few more that are rather obscure. A list appears in the bash man page under “READLINE.”

PROGRAMMABLE COMPLETION

Recent versions of bash have a facility called *programmable completion*. Programmable completion allows you (or more likely, your distribution provider) to add more completion rules. Usually this is done to add support for specific applications. For example, it is possible to add completions for the option list of a command or match particular file types that an application supports. Ubuntu has a fairly large set defined by default. Programmable completion is implemented by shell functions, a kind of mini shell script that we will cover in later chapters. If you are curious, try the following:

```
set | less
```

Now see whether you can find them. Not all distributions include them by default.

Using History

As we discovered in Chapter 1, bash maintains a history of commands that have been entered. This list of commands is kept in your home directory in a file called *bash_history*. The history facility is a useful resource for reducing the amount of typing you have to do, especially when combined with command line editing.

Searching History

At any time, we can view the contents of the command history list by doing the following:

```
[me@linuxbox ~]$ history | less
```

By default, bash stores the last 500 commands we have entered, though most modern distributions set this value to 1,000. We will see how to adjust this value in Chapter 11. Let's say we want to find the commands we used to list */usr/bin*. This is one way we could do this:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

And let's say that among our results we got a line containing an interesting command like this:

```
 88  ls -l /usr/bin > ls-output.txt
```

The 88 is the line number of the command in the history list. We could use this immediately using another type of expansion called *history expansion*. To use our discovered line, we could do this:

```
[me@linuxbox ~]$ !88
```

bash will expand !88 into the contents of the 88th line in the history list. There are other forms of history expansion that we will cover in the next section.

bash also provides the ability to search the history list incrementally. This means we can tell bash to search the history list as we enter characters, with each additional character further refining our search. To start incremental search, press CTRL-R followed by the text you are looking for. When you find it, you can either press ENTER to execute the command or press CTRL-J to copy the line from the history list to the current command line. To find the next occurrence of the text (moving "up" the history list), press CTRL-R again. To quit searching, press either CTRL-G or CTRL-C. Here we see it in action:

```
[me@linuxbox ~]$
```

First press CTRL-R.

```
(reverse-i-search)`':
```

The prompt changes to indicate that we are performing a reverse incremental search. It is "reverse" because we are searching from "now" to sometime in the past. Next, we start typing our search text. In this example, we're searching for */usr/bin*:

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

Immediately, the search returns our result. With our result, we can execute the command by pressing ENTER, or we can copy the command to our current command line for further editing by pressing CTRL-J. Let's copy it. Press CTRL-J.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Our shell prompt returns, and our command line is loaded and ready for action!

Table 8-5 lists some of the keystrokes used to manipulate the history list.

Table 8-5: History Commands

Key	Action
CTRL-P	Move to the previous history entry. This is the same action as the up arrow.
CTRL-N	Move to the next history entry. This is the same action as the down arrow.
ALT-<	Move to the beginning (top) of the history list.
ALT->	Move to the end (bottom) of the history list, i.e., the current command line.

Key	Action
CTRL-R	Reverse incremental search. This searches incrementally from the current command line up the history list.
ALT-P	Reverse search, nonincremental. With this key, type in the search string and press ENTER before the search is performed.
ALT-N	Forward search, nonincremental.
CTRL-O	Execute the current item in the history list and advance to the next one. This is handy if you are trying to re-execute a sequence of commands in the history list.

History Expansion

The shell offers a specialized type of expansion for items in the history list by using the `!` character. We have already seen how the exclamation point can be followed by a number to insert an entry from the history list. There are a number of other expansion features, as described in Table 8-6.

Table 8-6: History Expansion Commands

Sequence	Action
`!!`	Repeat the last command. It is probably easier to press the up arrow and ENTER.
`!`*`number`*	Repeat history list item *`number`*.
`!`*`string`*	Repeat last history list item starting with *`string`*.

Sequence	Action
`!?string`	Repeat last history list item containing *string*.

I caution against using the `!string` and `!?string` forms unless you are absolutely sure of the contents of the history list items.

Many more elements are available in the history expansion mechanism, but this subject is already too arcane, and our heads may explode if we continue. The "HISTORY EXPANSION" section of the `bash` man page goes into all the gory details. Feel free to explore!

SCRIPT

In addition to the command history feature in `bash`, most Linux distributions include a program called `script` that can be used to record an entire shell session and store it in a file. The basic syntax of the command is as follows:

```
script file
```

where *file* is the name of the file used for storing the recording. If no file is specified, the file `typescript` is used. See the `script` man page for a complete list of the program's options and features.

Summing Up

In this chapter, we covered *some* of the keyboard tricks that the shell provides to help hardcore typists reduce their workloads. As time goes by and we become more involved with the command line, we can refer back to this chapter to pick up more of these tricks. For now, consider them optional and potentially helpful.