From Bash to Z Shell: Conquering the Command Line

OLIVER KIDDLE, JERRY PEEK, AND PETER STEPHENSON

Apress®

From Bash to Z Shell: Conquering the Command Line

ISBN (pbk): 1-59059-376-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore

Technical Reviewers: Bart Schaefer and Ed Schaefer

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Liz Welch

Production Manager: Kari Brooks-Copony

Production Editor: Laura Cheu

Compositor: Susan Glinert

Proofreader: Linda Seifert

Indexer: Kevin Broccoli

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

CHAPTER 4

Entering and Editing the Command Line

In this chapter, we'll introduce you to a wide array of nuances concerning command-line manipulation. We'll start with some basic concepts, showing you how to choose a shell, and how to copy and paste commands. We'll then introduce line-editing basics using both `vi` and Emacs bindings, followed by a discussion of command history and navigation. We'll conclude this chapter with a series of discussions about advanced line manipulation and editing. By its conclusion, you'll be quite familiar with command-line navigation and best practices, knowledge that will undoubtedly save you considerable time and inconvenience as you interact with the shell. To begin, let's review the options and procedures involved when choosing a shell.

Terminals and Shells

If you're interacting with a shell, it can be running in one of two places:

- Occupying a full screen. If you used MS-DOS in the past, this will be what you remember. However, this way of using a computer is now rather old-fashioned.
- Sitting in a window in part of a screen, which may have lots of windows. This is much more common now. The window running the shell is a *terminal emulator*.

We'll now spend a little while explaining some of the features of a terminal emulator—how to start one, how to copy text from one to another, and so on.

Opening a Terminal and Choosing the Shell

Normally when you open a terminal emulator it starts your shell. By default, it uses the shell defined by you or your system administrator when your account was set up. (Cygwin is a little different. See the section "Getting Started with Cygwin" in Chapter 5.) This happens no matter which terminal emulator you use.

The other way of starting your shell is by going through a login procedure. That happens either from a `login` prompt at a console before the windowing system was started, or by logging in remotely to a system using one of the programs `telnet`, `rlogin` (you may be more familiar with the similar `rsh` or `remsh`), or `ssh`. We'll use the first approach. It's probably the more common, and it gives you more choices to make.

In the "old days," there was only one terminal emulator for the X Window System, `xterm`. It's still there, it still works, and it's still a good choice. In fact, at least one of the authors regularly uses `xterm`. Nowadays, there are plenty of other terminal emulators. The good news is that most others work pretty much like `xterm` as far as the user is concerned and, in fact, many of them are just enhancements to `xterm`.

To start a new `xterm`, execute the following:

```
% xterm &
```

Remember the `&` at the end. This lets your current terminal carry straight on, ignoring the new `xterm`. (This is further explained in the section "Starting and Stopping Processes: Signals, Job Control" in Chapter 3.) If the shell says it can't find `xterm`, try typing `/usr/X11/bin/xterm` or `/usr/X11R6/bin/xterm` or something similar; on a SunOS/Solaris system, there's probably an `xterm` in the directory `/usr/openwin/bin`. If it works, you'll get a new window looking more or less like the window you were in before.

You can run a different command (which will usually be a shell) using the `-e` option. For example, to start an `xterm` that is running `bash`, type the following:

```
% xterm -e bash &
```

Without the `-e` option, `xterm` consults the environment variable `SHELL`. This variable is typically set when you log in to the shell set up by the system administrator (which is specified in your system's password file). You can also set `SHELL` yourself. For example, if you want new `xterm`s to start up a `zsh` that lives in `/usr/local/bin/zsh`, you can set

```
% export SHELL=/usr/local/bin/zsh
```

You need to include that in a startup file if you want it to be remembered. (See the section "Startup Files, Login and Interactive Shells" in Chapter 5 for more on this topic.) Otherwise it only works for the current shell and other programs you start from it. Unfortunately, these days it's not so easy to find out what startup files are run when you log in graphically to a computer that starts up a desktop for you. Finding what the desktop will start can be hard work; it's a lot more complicated than simply starting a shell. So the easiest way to be sure terminal emulators use the shell you want is to change your login shell. Usually this is done with the `chsh` command. You type the command name and follow the prompts to change your login shell. You will need to enter the full path to the shell, for example `/usr/local/bin/zsh`.

Here's how to start a new `xterm` so that it remembers the shell but doesn't affect any other `xterm`s you start:

```
bash$ SHELL=/usr/local/bin/zsh xterm &
```

This sets the value of `SHELL` for this single `xterm`. Because it's immediately in front of that program, it's only set for that program. So if you `echo $SHELL` you'll find it hasn't changed locally, but the new `xterm` is using `zsh` instead of `bash`. (This is pointless if the value already was `/usr/local/bin/zsh`.)

Copy and Paste

Most terminal emulators have a copy-and-paste system based on `xterm`. They involve moving the mouse. The mouse position is completely separate from the cursor's current position. When you copy a chunk of text, it is done entirely with the mouse. Only when you paste it is the cursor position important—the text you paste begins at that point.

■ **Note** Mouse-1, Mouse-2, and Mouse-2 typically refer to the left, middle, and right mouse buttons, respectively.

Let's highlight some of this feature's basic behavior:

- To select a region, hold and move the left mouse button. Or click the left mouse button once at the start of the region and click once with the right mouse button at the end.
- Rapidly clicking the left mouse button twice selects a word; clicking three times selects a line.
- To extend a region, click the right mouse button at the point to which you want the region to be extended. The shell remembers whether you originally selected characters, words, or lines and continues to use the appropriate type. In some terminal emulators, this doesn't work; instead it brings up a menu. You may still be able to extend a region by holding down Shift and clicking the left mouse button.
- There is no special cut or copy operation; the selected region is immediately ready to be pasted somewhere else, and there's no way of deleting it. To paste it, click the middle mouse button (or wheel if there is one). Clicking both of the outside mouse buttons at the same time has the same effect on many systems.

Not all terminal emulators' cut and paste works in quite the same way. If the instructions we've just given don't seem to work, read the documentation for your terminal emulator.

Choosing a Terminal Emulator

Do you use either KDE or Gnome (two of the most popular desktops for Unix and GNU/Linux systems)? If you do, opening a terminal window will probably retrieve either Konsole or `gnome-terminal`, respectively. These behave pretty much like `xterm` with a menu bar. Note that there's nothing to stop you from using `gnome-terminal` with Gnome or Konsole with KDE, except that they probably aren't in the menus. You can start the one you want in the same fashion as opening an `xterm` window, described earlier.

■ **Tip** Old `xterm`s don't handle colors. Newer versions, and most alternatives, do. You will find colors are very useful with `zsh`'s completion listings to make different types of files stand out. The GNU `ls` command also supports colored file lists (try `ls --color=auto`).

We've summarized several key points concerning terminal emulators we've used in the past in the hopes that it will clear up some of the confusion as you begin your own experimentation. Keep in mind that the terminal emulator you choose is largely a matter of preference.

- Plain old `xterm` doesn't have menu buttons visible, but actually there are three menus that appear if you hold down the Ctrl key and then one of the mouse buttons. Two are quite useful. Holding Ctrl and clicking the middle mouse button gives you some options to change the look of the terminal, such as adding a scrollbar or turning on reverse video. Holding Ctrl and clicking the right mouse button lets you change the font to make it more (or less) readable. These also have command-line options, given in the `xterm` manual page.

  Saving options for future `xterm` sessions is trickier than setting shell options, though. It's done by the "X resources" mechanism, which is a bit out of the scope of this book. As an example, you can put the following lines in the file `~/.Xdefaults`. (Some systems use `~/.Xresources` instead, and some systems allow either file.)

  ```
  ! Change the font
  xterm*VT100*font: 10x20
  ! Turn on reverse video
  xterm*VT100*reverseVideo: true
  ```

  On most systems, those lines may automatically set the font and reverse video option for `xterm` when your desktop starts up. You also may need to load the values in the file by hand, which is done by the following command:

  ```
  % xrdb -merge ~/.Xdefaults
  ```

  (You will certainly need to do this to install the settings without logging in again.) Many traditional X Window programs like `xterm` will look for settings in this format in a file in your home directory. The name of the file is based on the program, for example `xterm`. (You can configure which directories are searched for such files by using the environment variable `XUSERFILESEARCHPATH`. Type `man X` and search for this variable for more information.) This is on the computer where the program is running, not necessarily the computer where the display is. The command `xrdb` is known as the "X server resource database utility." The `-merge` option tells `xrdb` not to remove existing resources. To find other things you can set in `~/.Xdefaults` in the same way, see the list of "RESOURCES" in the `xterm` manual page.

- In the case of KDE's Konsole, settings are done much more simply via obvious menus. Konsole has the additional feature that you can have multiple sessions (in other words, different shells, or even other programs) handled by the same Konsole. You create a new one from the New menu at the bottom, and can then switch between them by clicking on the appropriate name, or by using Shift-Left Arrow and Shift-Right Arrow. I tend to get confused and prefer to start separate windows. Needless to say, there are many other features. I have not yet successfully persuaded Konsole to accept a Meta key; see the sidebar "The Meta Key" (later in this chapter) for what that means.

- Gnome's `gnome-terminal` behavior is fairly similar to KDE's Konsole. Its version of the multiple-session feature is called "tabs," because of the way you select them, as you do the tabs in newer versions of the Netscape and Mozilla web browsers; you create a new tab from the File menu. It also has "profiles," which means you can store different groups of settings, for example different combinations of foreground and background colors, and switch between them by selecting a completely new profile instead of changing the options separately. You may find that the Alt or Meta key on your keyboard sometimes gets hijacked by `gnome-terminal` when you expect it to perform some editing task; you can fix this by going to the Edit menu, selecting Keybindings, and checking the option "Disable all menu mnemonics" (this is for `gnome-terminal` 2.0.1; older versions may be different).

- The third of the three most common Unix desktops—there are plenty of others—is the Common Desktop Environment (CDE). It's found on many versions of Solaris, HP/UX, and AIX. If you ask for a terminal here, you're likely to get `dtterm`.

- There's a lightweight `xterm` look-alike called `rxvt`. Its main claim to fame is it has fewer features than `xterm`. This may seem more impressive if you know some of the really strange things `xterm` can do. It's most likely to interest you if you have an old PC or workstation that strains to start terminal windows. Otherwise, it looks pretty similar.

- The `aterm` terminal emulator is based on `rxvt`; it was designed for AfterStep, the desktop used by one of the authors, but like all the others it can be used with any desktop or window manager. One nice feature of `aterm` is the option `-tr` that makes it appear transparent, which actually means it copies the piece of the root window (wallpaper in Windows-speak) that is behind it into its own background.

  Neither `rxvt` nor `aterm` provides menus. Note that both will pick up many of the values in `~/.Xdefaults` for `xterm`, but in that case you should remove the string `VT100*` from wherever it occurs in the example I showed earlier. It's there because `xterm` has another, now very seldom used terminal emulator built into it that handles graphics, and I didn't want to affect that, only the normal VT100 mode. VT100 is the name of an extremely widespread terminal of the 1980s made by DEC; `xterm` and its friends have borrowed many of the VT100's features.

- The Enlightened terminal, `Eterm`, is another enhanced `xterm`. It has a readable manual page and flashy backgrounds, and can be configured via configuration files, a fact that should make it appealing to those who are real shell programmers at heart.

- Old SunOS/Solaris systems came with OpenWindows, which had `shelltool` and `commandtool` as the terminal emulators. If you are stuck with these you should go for `shelltool`, since `commandtool` swallows up the escape sequences that the editors in `bash` and `zsh` use for advanced editing. There's an option for disabling `commandtool`'s special features, which essentially turns it into a shelltool. OpenWindows also provides genuine `xterm`, though on some really old versions it's hidden in a demo directory. It's unlikely to support color in any version of OpenWindows.

There are various ways to connect to a Unix-style system from Windows. The most powerful is to get hold of one of the full X Window servers that essentially turn your PC into an X server. In addition to the commercial packages such as XWin-32 and eXceed, you can get an X Window

server with Cygwin. We'll discuss Cygwin in the section "Getting Started with Cygwin" in Chapter 5.

If you don't want all of that complexity, there is a useful program called `PuTTY` that connects via the `telnet` or `ssh` protocol, among others. The administrator of the machine you are logging into will be able to tell you which one you should use. When `PuTTY` is connected, it behaves exactly like a color `xterm`, and even indicates to the Unix system that it is an `xterm` by setting the environment variable `TERM` to `xterm`. The emulation is very good, and we recommend this over other solutions.

The Command Line

Entering a key on the command line can have one of two effects:

1. Something appears on the screen. This is normal text.
2. The key has some special editing effect. Sometimes you can see what it is straight away, because some text was deleted or the cursor moved.

In the next few sections we'll introduce you to that second behavior. We'll also introduce the notion of command history and show you ways of referring to lines you've already typed, maybe at a previous session. There will be more on command history—including how to make sure it is saved and restored—in Chapter 6. (Some of the basics were already covered in the section "History" in Chapter 1.)

Line Editing Basics

Even if you're new to shells, you might well have found some of the simpler features of line editing already:

- We saw in the section "Editing Data, Continuing Command Lines" in Chapter 2 how you can break long lines. You can insert a backslash and a newline, and the shell will allow you to continue typing at the start of the next line. We also saw that Bourne-type shells (which include both `bash` and `zsh`) know that if there is something you need to finish typing, `zsh` will prompt you with a message saying what that is (in Chapter 2, you can see it produced the message `pipe pipe pipe>` to show that three pipelines are in progress). In that second case, you don't need a backslash.
- You can delete the previous character you typed with the Backspace or Erase key. (It's not always that simple, unfortunately; see the section "The Backspace and Erase Confusion" in this chapter.)
- You can move the cursor around the command line using the Left and Right Arrow keys. (This works on most terminals, but on sufficiently weird ones the cursor keys may result in strange effects; see the section "Finding the Key Sequences for Function Keys" in this chapter for what to do if you have problems.)
- The Up Arrow and Down Arrow keys allow you to go back and forth in the list of commands you've already entered.

These basic operations are similar in bash and zsh. In fact, some particularly simple operations can be carried out by the terminal itself, as we explain in the sidebar "The Terminal Driver," later in this chapter. However, the more complicated the operation, the more different the shells look.

Emacs and vi Modes

Both bash and zsh offer two editing modes based around the Emacs and vi editors. The emacs mode presents the more obvious operation for people who have no particular preference, whereas vi behavior can seem a bit strange at first. In vi mode, the keyboard can be in two different states, where the same keys are used either for entering text or for performing editing operations. There is no visual indication of whether you are in the insert or editing state. (These are usually known as "insert mode" and "command mode" by vi users.) Because Emacs is a little easier to pick up, we will spend most of this section talking about this editing mode. If you know Emacs's reputation for complexity, don't worry: the shell's editor is much simpler, and most editing is done just with a few keystrokes.

Setting the editing style is done differently in bash and zsh. In fact, virtually all the shell commands used to control editing are completely different in the two shells. In bash, to enter emacs or vi mode you can set options as follows:

```
bash$ set -o emacs
bash$ set -o vi
```

In zsh, almost all the commands that change the behavior of keys use the bindkey command. As we'll see in the section "Configuration and Key Binding: readline and zle" in this chapter, this command is normally used for associating a keystroke with an editor command. However, it's also used for putting the line editor into emacs or vi mode. The following shows the command to enter emacs mode, then the command to enter vi mode:

```
zsh% bindkey -e
zsh% bindkey -v
```

(From version 4.2, zsh understands `set -o vi` and `set -o emacs`. We suggest you learn about bindkey, however, since it is so useful in other ways.)

If you usually use vi —or your system administrator thinks you do—one or both of the environment variables EDITOR and VISUAL may be set to vi. Many commands use this variable to determine your preference when they need to start up an external editor. zsh will also examine these variables when it starts. If either contains vi (including names like elvis, which is an editor like vi), it will set the editing mode to vi. So if you are sure you want to use emacs mode, it is a good idea to include an explicit line `bindkey -e` in `~/.zshrc`.

If you use other editors, a lot of things probably won't be set up quite how you want them. We'll talk about how to configure the shells later in the section "Configuration and Key Binding: readline and zle."

Conventions for Key Names

The shells inherit some conventions from Emacs. They are also used in vi mode, so you should know them. Here, we'll explain how the syntax used in describing keys is related to the keys you type.

The Ctrl Key

Where we write `\C-x`, you should hold down the Ctrl key and then press x. Do it in that order, or you will get an x on the screen. There are many other possibilities instead of x, although typically you will use a lowercase letter. When you use a `zsh bindkey` or `bash bind` command, that's the form you need to use. Elsewhere, when we're talking about what you type, we'll just say Ctrl-x. Don't type the hyphen; it just shows which keys are held down at once.

The Meta and Esc Keys

In `\M-x` the Meta key is used instead of the Ctrl key. We talk about the Meta key in the sidebar "The Meta Key." However, you can get the same behavior as `\M-x` by typing the Esc key, nearly always at the extreme top left of the keyboard, then pressing x (don't hold Esc down). (The Meta and Ctrl keys, and any others used similarly, are usually known as *modifiers*. The Shift key is a modifier, too, but you don't need a special symbol since Shifted characters are different from the un-Shifted ones.)

In `bash`, preceding a key with Esc and holding down Meta while you press a key do exactly the same thing. The shell uses `\M-x` for both purposes. You can combine the effects of modifiers, for example `\M-\C-x`. Using the Esc key, you get this by pressing Esc, then Ctrl-x. (You don't need Ctrl with the Esc key.)

In `zsh`, Meta and Esc aren't always tied together and it's often easier to use the Esc key alone. The Esc key is represented within `zsh` by `\e`, so we'll use that when we're referring to shell code. Note that `zsh` rather unhelpfully shows Esc in lists of keys as `^[`. This means the same but isn't as obvious.

Key Sequences

Some more complicated operations use a *key sequence*, a set of keys pressed one after the other. We show these with a space in between when there is a special key like one of the two above. For example `\M-xfoo` means press Meta x or Esc x, then type the ordinary characters `foo`.

THE META KEY

Few keyboards now have a key marked Meta. It is intended to be a modifier, just like Shift or Ctrl. On some PC keyboards the Alt key functions identically to the Meta key. You can combine modifiers; for example, `\M-\C-x` is given by holding down both the Meta and Ctrl keys, followed by the x key.

`bash` and `zsh` have different approaches to key bindings with the `\M-` prefix. In `bash`, they are tied to the corresponding key bindings with Esc automatically. In `zsh`, however, the keys are actually different. There is no automatic link in `zsh` between what `\M-x` does and what Esc x does. You can force `zsh` to use equivalent bindings for both sets, so that, for example, pressing Meta x has the same effect as pressing Esc x. The following command has this effect on all the bindings that use the Esc key in emacs mode:

```
zsh% bindkey -me
```

For vi mode the equivalent command is the following:

```
zsh% bindkey -mv
```

The extra `-m` has the effect of copying bindings for Esc x to those for Meta x. Nothing happens if the key has already been bound. This means that you need to remember to add separate commands to bind `\e` and `\M-`, if you use both.

Actually, Esc is often a good deal more useful than any Meta key. This is not just because that may be hard to set up, but more importantly because of how Meta works. It makes the terminal produce the same character but with the eighth bit set. This is not what you want if those characters are actually part of the character set you are using. In particular, it is no use at all if you are using the increasingly popular UTF-8 representation of characters, since the eighth bit has a special effect. So from now on we'll assume you're using the Esc key. However, we'll still refer to bindings in the form `\M-x` for `bash`, since that's how the `bash` documentation describes them. When we're not showing the code you use for key binding we'll say Esc x.

Tour of emacs Mode

If you use Emacs, XEmacs, or some other editor with similar functions such as MicroEmacs, many of the keys you use will be very familiar. If you don't, many of them might seem a little obscure at first, but there is a theme you can get used to. Most of the keys in this section are common to `zsh` and `bash`. We won't give examples, but you can try everything very easily; just type a few characters, without pressing Return, and then the special keys we tell you.

A few terminal emulators won't let you use the Esc key in the shell editor because it's taken for the terminal's own purposes. If you find this annoying, you either have to look at the terminal's documentation to see how to change this, or use another type of terminal.

Basic Moving and Deleting

Simple move and delete keys for single characters are nearly all either special keys or keys with Ctrl held down. The most obvious are in Table 4-1.

Table 4-1. *Simple Move and Delete Keys*

Key	Purpose
Backspace or Erase	Delete previous character
Ctrl-d	Delete character under cursor
Left Arrow or Ctrl-b	Move to previous character
Right Arrow or Ctrl-f	Move to next character
Ctrl-t	Transpose the character with the previous one (see the sidebar "Transposing Characters")
Ctrl-v	Insert the next character literally (explained later)

Note that the Delete key on a full-size PC keyboard is not the same as Erase or Backspace. You might expect it to delete the next character, as it often does under Windows. See the section "Configuration and Key Binding: readline and zle" in this chapter for help with this. (We're describing the most common case on a PC-style keyboard. Some older Unix keyboards may behave differently.)

It's sometimes useful to remember Ctrl-b and Ctrl-f if you find yourself using an odd keyboard where the shell is confused by the cursor keys.

The key Ctrl-d is often rather overworked:

- If there are characters after the cursor, Ctrl-d deletes the first one, as we showed in Table 4-1.
- In `zsh`, if you press Ctrl-d at the end of a line, it will probably show you a list of files. This is part of the shell's completion facility, described in Chapter 10. (This is a feature taken from the C shell.)
- In both `bash` and `zsh`, typing Ctrl-d on an empty line may even cause the shell to exit! That's because traditionally Unix treats Ctrl-d as "end-of-file" (EOF), a message from the user that they have finished sending input. Both shells have mechanisms for suppressing this. Here, `bash`'s is more powerful. If you set the following, then `bash` will only exit if you press Ctrl-d ten times in a row:

  ```
  bash$ IGNOREEOF=10
  ```

  In `zsh`, the equivalent effect is achieved by setting an option, as follows:

  ```
  zsh% setopt ignore_eof
  ```

  However, in `zsh` the value 10 is fixed. This needs to be a finite number so that the shell can exit if the terminal has completely gone away. In that case the shell will read EOF characters continuously, and it would be unfortunate if it kept on doing so forever.

TRANSPOSING CHARACTERS

It may not be obvious quite what Ctrl-t, bound to the command `transpose-chars`, is doing. If you type it at the end of the line, it transposes the previous two characters, so repeating it swaps them back and forth. However, if you do it early in the line, it swaps the character under the cursor with the one before it, then moves the cursor on. The point of this rather complicated behavior is that pressing Ctrl-t repeatedly has the effect of marching a character further and further up the line. This is useful if more than one character were out of place. Of course, in that case deleting it and inserting it somewhere else may be easier.

You use Ctrl-v to put onto the command line a character that would otherwise be an editing key. One common use for this is to insert a Tab character. On its own, Tab performs completion. If you want to insert a Tab character, you need to press Ctrl-v-Tab.

Moving Further: Words and Lines

Some more powerful motion keys that move the cursor further than a single character are shown in Table 4-2. The backward and forward keys are easy enough to remember, and Ctrl-e for end is also easy enough, but you'll just have to remember Ctrl-a, the start of the alphabet, meaning the start of the line.

Table 4-2. *More Powerful Motion Keys*

Key	Purpose
Esc-b	Move backward one word
Esc-f	Move forward one word
Ctrl-a	Move to the beginning of the line
Ctrl-e	Move to the end of the line

The keys Esc-b and Esc-f move the cursor backward and forward over *words*. There's some disagreement over what a "word" means. In particular, it doesn't mean a complete command-line argument. In `bash`, it's simple: only letters and digits (alphanumerics) are part of a word.

In `zsh` alphanumerics are part of a word, too. However, there is an additional set of characters that will be considered part of a word. These are given by the shell variable `WORDCHARS`, whose initial contents are the following:

```
*?_-.[]~=/&;!#$%^(){}<>
```

You can set `WORDCHARS` to any set of characters. In particular, if you always want the `bash` behavior, you can make it empty:

```
zsh% WORDCHARS=
```

There's no equivalent to `WORDCHARS` in `bash`; the notion of a word is fixed. In both shells, there are only two kinds of characters: word characters and the rest. That may sound rather stupid, but remembering it helps you to understand what Esc-b and Esc-f actually do:

- Esc-b skips backward over any number of nonword characters, and then any number of word characters before that.
- Esc-f skips forward over any number of word characters, and then over any number of nonword characters after that.

In each case "any number" may be zero if there aren't any characters of that type at that point. The reason for this behavior is so that repeated Esc-b or Esc-f keystrokes take you over as many words as necessary, without you having to move the cursor in between.

The Command History and Searching

The shells remember previous lines you've entered; you can scroll back or search in that list. How many previous lines it remembers, and whether they are saved between sessions, is discussed in Chapter 6. For now, just think of the history as a series of command lines in the order you typed them.

Two very useful keys you may already have found are given in Table 4-3.

Table 4-3. *The Simplest Keys for the Command History*

Key	Purpose
Up Arrow	Go back in the command history
Down Arrow	Go forward in the command history

The last few commands are waiting for you to scroll back through them with the Up Arrow key. Usually the command you're just typing is at the end of the command history, so Down Arrow is only useful after one or more presses of Up Arrow. However, both shells allow you to edit any line in the history, and all those edits are remembered until you press Return to execute the command. At that point the history returns to the actual history of executed commands.

Here's an example. Type the following:

```
% echo this is the first line
this is the first line
% echo and this is the second
and this is the second
```

Then press Up Arrow twice to see the first line again. Change it in some way, but don't press Return. Then press Down Arrow, and change that line. You'll find you can move up and down between the two changed lines. Let's suppose you change the first line and press Return:

```
% echo but this is no longer the first line
but this is no longer the first line
```

Now you'll find the history contains the three last lines I've shown you; the edits to the second line you made were lost, and the edited version of the first was added as a new line at the end.

Searching the Command History

One very common task involves searching backwards through the command history to find something you entered before. There are various shortcuts that let you avoid having to scroll up through the history until you see it.

First, there are easy ways to get to the very top and very bottom of the history, although they don't perform searches. These are shown in Table 4-4.

Table 4-4. *Moving to the Start and End of the History*

Key	Purpose
Esc<	Go to the start of the command history
Esc>	Go to the end of the command history

Next, there are real searches. You have various ways to tell the shell what to search for. The easiest to use are the "incremental" searches, shown in Table 4-5. These search for the nearest string in the history that matches everything you've typed. The whole line is searched.

Table 4-5. *Incremental Searches*

Key	Purpose
Ctrl-r	Search backward incrementally
Ctrl-s	Search forward incrementally (also Ctrl-x-s in `zsh`)

Normally, Ctrl-r is more useful since you start at the end of the history and want to search back through it.

THE NO_FLOW_CONTROL OPTION

We've offered Ctrl-x-s as an alternative version of Ctrl-s, which is made available in `zsh`. (So is Ctrl-x-r, but you don't need that so often.) The problem is that very often the terminal is set up so that Ctrl-s and Ctrl-q perform flow control: another feature provided by the terminal itself, not the shell. We talk about other such features in the sidebar "The Terminal Driver." When you press Ctrl-s, any output to the terminal, including anything you type, is stopped. When you press Ctrl-q, it starts up again and you can see what you typed. In `zsh` you can fix this by setting the shell option `no_flow_control`. This doesn't stop the use of Ctrl-s and Ctrl-q in external programs, to let you see something which is scrolling past quickly, so it's fairly safe. (You may find if your terminal output is very fast they don't have an instant effect even then.) There's another, clumsier way of getting control using the `stty` command, which we'll explain near the end of this chapter.

Let's consider an example depicting the effects of Ctrl-r. Suppose you've typed the three lines I used as an example earlier:

```
% echo this is the first line
this is the first line
% echo and this is the second
and this is the second
% echo but this is no longer the first line
but this is no longer the first line
```

Now you type the following:

```
<ctrl-r>this is
```

Both shells prompt you during searches, showing you what you've typed to find the line you have reached. In `bash`, you'll see the message (`reverse-i-search`) appear. If you've entered an incorrect character for the search, you can press Backspace to delete it.

You'll return to where that string occurs on the third line. Now continue so that the complete line looks like the following:

```
<ctrl-r>this is the
```

Now you'll find you're looking at the line before. Finally, continue with the following text:

```
<ctrl-r>this is the first
```

You've returned to the first line. If you type something other than a key that inserts itself or deletes backwards, the search ends and you stay on the line found. Be careful with Return, however, since it immediately executes the line. If you want a chance to edit the line first, press another key such as Left Arrow.

This is probably as good a point as any to tell you about Ctrl-g, which you can press to abort the current operation. If you were in the middle of a search, you will return to the line at the end of the history, quite likely still blank. If you were just doing normal editing, Ctrl-g would abort the current line and display an empty one.

You can resume searching from where you left off by pressing Ctrl-r twice, once to start the search and again to restore the string you were searching for before. If you abort with Ctrl-g, however, the shell doesn't remember what you are searching for on that occasion and keeps whatever was there before. In other words, it's best to get into the habit of exiting with a different key.

There are two other types of searching. First, there are nonincremental versions of the search commands, available in `bash` as Esc-p and Esc-n. (The p and n stand for previous and next. In fact, Ctrl-p and Ctrl-n are actually alternatives to the Up and Down Arrow keys, respectively.) If you press Esc-p, you will see a colon as a prompt. Type a string after it, then press Return. The shell searches back for that string.

Often there is no advantage to using the nonincremental instead of the incremental versions. Still, if you feel called to use the `zsh` versions, they can be borrowed from vi mode. The names of the commands are `vi-history-search-backward` and `vi-history-search-forward`; we'll see shortly how you can use this information to associate the command with a key.

A more useful alternative is to have the shell search backward for a line that begins with the same command word, or with the same string up to the cursor position, as the current line. Both forms are present in `zsh`, but only the second form in `bash`.

The first (`zsh` only) form is shown in Table 4-6.

Table 4-6. *Searching Lines for the First Word in `zsh`*

Key	Purpose
Esc-p	Go to the previous line starting with the same word
Esc-n	Go to the next line starting with the same word

This means that wherever the cursor is on the command line Esc-p will take you back to the previous line beginning with the same command word, ignoring any other text. There's an exception: If you are still typing the command word, Esc-p will take you back to any line beginning with the same characters. Note that in `bash` the same keys do nonincremental searching as described above.

The other form is called `history-beginning-search-backward` (or `-forward`) in `zsh` and `history-search-backward` (or `-forward`) in `bash`. In this case, the shell looks back for a line in which every character between the start of the line and the cursor position is the same, so the more you have already typed, the more precise the match. It's essentially a form of nonincremental search where you type the characters before you search for them. For those of us with limited foreknowledge of our own actions, this can be quite a blessing.

(As an extra piece of confusion, the `bash` names `history-search-backward` and `-forward` are what `zsh` calls the commands behind Esc-p and Esc-n.)

Deleting and Moving Chunks of Text

For deleting chunks of text larger than a single character, the shells use the word *killing*. This is more slang from Emacs. It means the shell has deleted the text, but has remembered it on something called the *kill ring*. Some keys for killing text are given in Table 4-7. (Editor commands exist for simply deleting text, but since remembering the deleted text is usually harmless, they're not often used.)

Table 4-7. *Keys for Killing Text*

Key	Purpose
Esc-Backspace or Esc-Erase	Kill the previous word
Esc-d	Kill the next word
Ctrl-k	Kill to the end of line
Ctrl-u	Kill to the beginning of the line (`bash`); kill the entire line (`zsh`)

Retrieving the text you killed is called *yanking*. You can do this after moving the cursor or even on a new command line. There are two basic commands, shown in Table 4-8.

Table 4-8. *Keys for Yanking Text*

Key	Purpose
Ctrl-y	Yank the last killed text
Esc-y	Remove the last yanked text and replace it with the previously killed text

Most of the time, you just want to pull back the last text using Ctrl-y. You can get back earlier text, though, as we explain in the sidebar "The Kill Ring."

THE KILL RING

The shell doesn't just remember the last text you killed; it remembers several of the previous strings as well. The place it stores them is called the "kill ring." After pressing Ctrl-y to bring back the last piece of text killed, you can press Esc-y one or more times. (Pressing Ctrl-y again has a different effect; it yanks back the same piece of text repeatedly.) Each time you press Esc-y, the text that was just yanked is taken away again, and the previous piece of killed text is inserted instead.

It's a "ring" because only a finite number of chunks of text are remembered. When you have reached the last one stored, pressing Esc-y cycles back to the start, which is the text that was last killed. The ring contains up to ten elements. Just keep typing Esc-y until you see what you want or you get back to the original.

The Region

The *region* is yet another concept originating from Emacs. You set a *mark* in one place and move the cursor somewhere else. The "region" is the set of characters between the cursor and the mark. To be more precise, the cursor defines a *point* just to its left. When you make a mark it remembers that point, and when you move the cursor, the region is between the new "point" and the "mark." Some keys for using the region are given in Table 4-9.

Table 4-9. *Keys for Using the Region*

Key	Purpose
Ctrl-Space	Set the mark at the current point. This is written `\C-@` in both shells; if Ctrl-Space doesn't work, try Ctrl-2.
Ctrl-x Ctrl-x	In `zsh`, swap the mark and the point. Present in `bash` without a key defined.
Ctrl-w	In `bash`, kill the region for later yanking. Present in `zsh` without a key defined.
Esc-w	In `zsh`, copy the region for yanking, but don't actually delete the text. Present in `bash` without a key defined.

As you see, we're getting to the point where the behavior of bash and zsh start to diverge. So as to keep the information together, here are the commands to set up Ctrl-x Ctrl-x and Esc-w in bash:

```
bind '"\C-x\C-x": exchange-point-and-mark'
bind '"\M-w": copy-region-as-kill'
```

Here is how to set up Ctrl-w in zsh:

```
bindkey '\C-w' kill-region
```

Those are the first key bindings we've shown; there will be more. You need to type them at a prompt, or save them in your .bashrc for later use. For now, note that the long names with hyphens are *editor commands*, also known as *editor functions*, or known in zsh as *widgets*. There are many of these, and you can bind them to any set of keys the shell understands.

If you try Ctrl-w in zsh without this, you'll find it kills the previous word. For example, suppose you type the words `echo this is a line`. With the original binding of Ctrl-w, typing those keys would delete the word line. If you have rebound it to kill-region, however, it will probably delete the whole line. That's because by default the region starts at the beginning of the line.

A Few Miscellaneous Commands

Before we move on to more complicated matters, Table 4-10 gives a few commonly used commands available in both shells.

Table 4-10. *Miscellaneous Common Keys*

Key	Purpose
Esc-c	Capitalize the next word and move over it
Esc-u	Uppercase the next word and move over it
Esc-l	Lowercase the next word and move over it
Ctrl-l	Clear the screen and redraw it. In zsh this may preserve the list of possible completions.
Esc-. (period)	Insert the last argument of the previous command line. Repeat to retrieve arguments from earlier lines.

Beyond Keystrokes: Commands and Bindings

Every editor command has a name, whether or not there is a key associated with it. The names mostly have the form several-hyphenated-words. A few editor commands are just a single word with no hyphens. The name is useful when you come to attach a command to a key or set of keys. For a complete set of commands, consult the documentation for the shell.

Names and Extended Commands

In zsh you can execute a command simply by knowing its name. To do this, you press Esc x, then the name of the editing command. Completion is available—type as much as you like, then press Tab. If you haven't typed much, you may be overwhelmed with information when you press Tab and the shell will ask if you really want to see it all before showing it.

Configuration and Key Binding: readline and zle

It's time to talk a bit more about how to set up the line editor the way you want it. First, we'll offer some background about the way the shells' line editors work, which is a bit different between bash and zsh.

bash uses the GNU project's standard library, *readline*, which does exactly what its name suggests. Because it's a library, it is used as part of a lot of different programs. For example, if you use the GNU debugger, gdb, you will find it uses readline to handle command-line editing. readline has its own configuration file, from which you can configure a lot of different tools; the sidebar "readline Configuration" has more information on this. You can still configure readline settings from your ~/.bashrc. To keep things simple, we'll show you how to do everything from there.

READLINE CONFIGURATION

Configuration for the readline library goes in a file named ~/.inputrc. (A system-wide /etc/inputrc file also exists but you shouldn't need to be concerned with that.) Although the syntax of this file is different from the normal shell syntax, it is fairly simple. Each line takes the same form as an argument to the bind built in. So to bind a key you might, for example, use the following command:

```
"\C-u": kill-whole-line
```

A number of *readline variables* allow other aspects of readline's behavior to be configured. One example is mark-modified-lines. It makes readline warn you about lines in the history that have been modified since readline added them to the history. When you recall a modified line, an initial star ("*") will be displayed at the beginning of the line. To enable this feature, you can put the following line in your .inputrc file:

```
set mark-modified-lines on
```

You can see what the current settings for all the variables are by typing bind -v. You can also set readline variables direct from bash using the bind command:

```
bind 'set mark-modified-lines on'
```

We mention specific readline variables later in the book, particularly in Chapter 10. Be aware that many are new. They may not work if you don't have the latest version of bash.

zsh has its own "zsh line editor," or "zle" for short. Because it's always used within zsh, the only way to set up zle is with standard shell commands.

The most common action to configure the line editor is to attach a series of keystrokes to an editing command. This is described as "binding" the command. Both shells use this language, and it shows up in the name of the respective commands. The `bash` command is `bind` and the `zsh` command is `bindkey`; they have rather different syntaxes.

You can find out what commands are bound to keys already. Here's how to do that in `bash`:

```
bash$ bind -p
```

This produces a complete list of editor commands. Each line is in the form you can use for a later `bind` command to re-create the binding (although if you're just looking at the defaults, that's not so useful). Unbound commands are shown with a comment character (`#`) in front. Here is an example of a bound and an unbound command:

```
# vi-yank-to (not bound)
"\C-y": yank
```

The equivalent in `zsh` is the following:

```
zsh% bindkey -L
```

which is better than `bind` in one way—it generates a complete command that you simply paste into your `~/.zshrc`—and worse in another—it doesn't show you unbound commands. Omitting `-L` produces a simpler list with the keys and bindings but not in a form you can cut and paste.

There is another command to generate a complete list of all editor commands that can be bound:

```
zsh% zle -la
```

It's just a simple list of names, one per line. The `zle` command is used for creating extensions to the editor; we'll meet it in Chapter 14. The `-l` option means list, and `-la` means list all. Without the `a` you just see user-defined commands. These are features you've added to the editor yourself; we'll see how to do this in Chapter 14.

In `zsh`, you are allowed to query individual bindings to see if they already do something. Here is an example:

```
zsh% bindkey '\C-y'
"^Y" yank
```

That's the binding for Ctrl-y. Don't put spaces in the argument to `bindkey`, since they count as characters. In `bash` you can use the `grep` command:

```
bash$ bind -p | grep '"\\C-y"'
"\C-y": yank
```

This produces comfortingly similar results. Note that extra backslash inside the quotes, since backslash is special to `grep` as well as to the shell.

Finally, you can always find out from `zsh` what a command is bound to on the fly by using the editor command `where-is`. This isn't itself bound by default, so you need to use Esc x `where-is` unless you have bound it. It then takes an editor command, just like Esc x itself, and will tell you what keys you need to type to get that command.

For example, I entered the `where-is` command, typed `backward-char`, and pressed Return, and `zsh` reported the following:

```
backward-char is on "^B" "^[OD" "^[[D"
```

The last two sets of characters are the two most common strings of characters sent by the Left Arrow key.

Note there is nothing stopping you from binding a command to a key sequence that is already in use; the shell will silently replace the old one.

Once you've picked the editor command and a key sequence, the commands to bind the two together are the following. For `bash`, let's bind `history-search-backward` to Esc-p:

```
bash$ bind '"\M-p": history-search-backward'
```

You do need to remember all those quotes. The rationale is that the single quotes surround a string that `bash` will pass to the readline library. That in turn expects the name of complex characters such as Esc-p to be quoted, hence the double quotes around the `\M-p` inside the single quotes.

For `zsh`, let's bind `history-beginning-search-backward` to Esc-p:

```
zsh% bindkey '\ep' history-beginning-search-backward
```

Here, the key and the command are separate arguments to `bindkey`. You still need the single quotes around `\ep`, though. Suppose you issue the following command:

```
zsh% bindkey \ep history-beginning-search-backward  # Wrong!
```

Because of the shell's quoting roles, the command-line processor uses the backslash to quote the `e` that follows, and `bindkey` sees only `ep`, which just means exactly those two characters—not what you meant. You need to be on your guard in this way any time you see "funny" characters being passed to a command. For `bind` and `bindkey`, "funny" characters are so common it's just as well to get into the habit of quoting any reference to keys.

The sequence `\e` for Esc is one of a set of special keys. Other useful sequences include `\t` for Tab (strictly a horizontal tab, since there's a rarely used vertical tab), `\b` for Backspace, and `\\` for backslash itself. All need the extra quotes when used with `bindkey`.

■ **Warning** You can't just use the characters `Escape` in a `bind` or `bindkey` command because they just mean exactly those characters: `E`, `s`, `c`, `a`, `p`, `e`. This is one of many places where the backslash is necessary to tell the shell that some special behavior is required.

You can bind any 8-bit character in this fashion if you know the character's number in the character set. The easiest way (in both `bash` and `zsh`) is to convert the number to two hexadecimal digits *HH*. The character is then represented as `\x`*HH*. For example, character `\x7f` (127 decimal) is the Delete key. This might look a little more memorable than the standard but obscure string `\C-?`.

Finding the Key Sequences for Function Keys

Your keyboard probably has a row of function keys at the top. If you are using a full-size PC keyboard there is also a group of six named keys above the cursor keys. The keys have names like F1 and Home. It's very useful to bind these to commands.

Unfortunately, you can't simply use the name on the key. Shells only have the ability to read strings of characters. The terminal emulator generates a string of characters (often called an *escape sequence*) for each special key. These bear no relation to the names on the keys. Luckily, it's easy to bind the escape sequences. The hard part is finding out what sequence of characters the function keys send. The simplest, most general way we know of is to type `read`, then press Return, then the key combination you want to investigate. You'll see the characters the terminal sends. Let's try the function key F1. On my system this has the following effect:

```
zsh% read
^[[11~
```

(If you press Return again, you've actually assigned that value to the variable `REPLY`. We'll meet `read` in the section "Reading Input" in Chapter 13.) The characters appear literally because neither shell currently uses the line editor for the `read` command.

The key sequence `^[[11~` shown above is the one most commonly produced by F1. It's not guaranteed to be the same in other terminal programs, if you use more than one, but there's a pretty good chance. However, it doesn't matter as long as you know what the string is.

The `^[` represents an escape character. The shells know that as `\e`, which is a little more obvious. So here's how to make F1 perform the command `forward-word` in `bash`:

```
bash$ bind '"\e[11~": forward-word'
```

Here's the corresponding example in `zsh`:

```
zsh% bindkey '\e[11~' forward-word
```

Very often the key sequences sent by Shift-F1, Ctrl-F1, and Alt-F1 are different from the key sequence sent by F1 alone. I get `\e[23~`, `\e[11^`, and `\e\e[11~`, respectively; you can of course investigate and bind these in just the same way. You may also be able to combine the modifiers, for example Shift-Alt-F1, to get yet another key sequence.

■ **Warning** There is a good chance that one or more of the modified forms of the function keys are intercepted by the window manager for some special task. Usually it is possible to get round this by changing key bindings for the window manager, either in a startup file or interactively using the Gnome or KDE control panel or equivalent. Also, the terminal emulator itself may intercept keys; for example, often Shift-PageUp scrolls the terminal window.

The set of six keys—Insert, Delete, Home, End, PageUp, and PageDown—found on modern PC keyboards can be treated in the same way we just described for function keys. You may decide to make these do what their names suggest. (In modern graphical editors, they already will:

Home often moves the cursor to the beginning of the line. However, the shell doesn't have a binding until you give it one.)

For example, both shells have an `overwrite-mode` editor command that toggles the line editor between two states. In the normal state (insert), typing a character in the middle of a command line inserts the new character and pushes the existing text to the right. In the other state (overwrite), typing a character replaces the character that was there before and leaves the rest of the line where it was. It's quite useful to bind this to the Insert key. (If you like overwrite mode, in `zsh` you can use the command `setopt overstrike` to make it the default mode when the line editor starts.)

Also, you might want to bind Home and End to `beginning-of-history` and `end-of-history`, which take you to the first and last line in the history.

On my terminal, I can bind the keys as follows:

```
zsh% bindkey '\e[2~' overwrite-mode       # Insert
zsh% bindkey '\e[1~' beginning-of-history # Home
zsh% bindkey '\e[4~' end-of-history       # End
```

As we just noted, `\e[2~` has the same effect as `^[[2~`, and so on. The `\e` for escape is just a little more readable. Note that the cursor keys, too, send escape sequences of this sort. The shells try to find out what the cursor keys send and bind them for you, but in any case you can use the `read` command to find out. What's more, the cursor keys with modifiers, such as Ctrl-Up Arrow, may send different key sequences from the Up and Down Arrow keys alone. So you can bind those, too.

If you use several terminal emulators where the function keys send different escapes, you can use the shell's `case` statement to choose the right binding for the terminal. We show an example of that in the section "Case Statement" in Chapter 13.

Binding Strings

Both shells have ways of binding a string instead of a command to a key sequence. This means that when you type the key sequence the string appears on the command line. What's more, the string can itself consist of special keys that have a meaning to the editor.

In `bash`, this is done by putting a quoted string in the place of the command when you use `bind`. Here's an example:

```
bash$ bind '"\C-xb": "bind"'
```

The quoted `"bind"` is treated as a string, so when you press Ctrl-x-b the shell acts as if you'd typed `b`, `i`, `n`, `d`. From now on, you can use this instead of typing the full name of the `bind` command.

The same thing in `zsh` is done by the `-s` (for string) option to `bindkey`:

```
zsh% bindkey -s '\C-xb' bindkey
```

If you include special characters, they have exactly the same form as they do in the part of the command where the keystrokes are defined:

```
zsh% bindkey -s '\C-xt' 'March 2004\eb'
```

This inserts the string `March 2004` onto the command line, then emits the sequence `\eb`. If you are using normal Emacs bindings the cursor goes back a word to the `2` so you can insert the day of the month before the year. You can make these strings as complex as you like.

This only works with the normal key bindings for emacs mode. If you rebind `\eb`, that other command is called when the `\eb` is processed instead of the `backward-word` command. In Chapter 14 we show you how to write your own editor commands to avoid this problem.

In addition to `\C-x`, `zsh` accepts `^x` (a caret followed by a letter) as a string for the key Ctrl-x, and similarly for other control keys. We recommended that you put it in quotes, since in some circumstances `^` can have a special meaning to `zsh`:

```
zsh% bindkey -s '^xf' foo
```

Executing Commands

Now you know about binding a named command to a keystroke, it's worth pointing out that the action of executing a command line uses an editor command. In both shells it's called `accept-line` and, as you will guess, it's bound to the Return key. There's usually no reason for changing this, except for special effects. In `zsh`, we'll see such effects in Chapter 14.

There are related and more sophisticated commands, however. Both shells have a handy function bound to Ctrl-o in emacs mode, though it has different names—`accept-line-and-down-history` in `zsh` and `operate-and-get-next` in `bash`.

An illustration shows this best. Let's suppose you're using the `vi` editor to edit a program, then the `make` command to compile it. (The `make` command reads instructions from a file, usually named `Makefile` or `makefile`.) Often, you need to look at the output from the compiler, then edit the program again to fix a problem. Suppose you've already issued the following commands:

```
% vi myprogram.c
% make
... error message from make ...
```

(If you want to try that straight away, just put "`echo`" at the start of both lines. Then you can see what the shell would execute if you were really writing a program.)

Now press the Up Arrow key twice to get back to the first of the two lines, then press Ctrl-o. The line is executed again, but this time when you get back to the line editor it puts up the second line again. This makes it very easy to recycle a complete set of history lines.

Press Ctrl-o again, and you see the first line again. (It was saved to the history the first time you pressed Ctrl-o.) If you keep pressing Ctrl-o, you keep re-executing those two lines, as many times as you like.

When you've finished editing and want to run the program, just delete the line the shell shows you and start a new one. When you press Return at the end it will be added at the end of the history as normal.

Multiple Key Sequences

We've referred to several keystrokes that consist of several keypresses chained together, such as Ctrl-x t for Ctrl-x followed by t. You may wonder what the rules are for constructing them. The answer is that you can do pretty much what you like. It's a convention that Esc and Ctrl-x

are *prefixes*, in other words keystrokes that don't do anything on their own but wait for you to type something else. However, any keystroke can be a prefix, and you are not limited to two characters in a row. Suppose, for example, you decide to keep all the bindings to do deletion in sequences beginning with Ctrl-x-d. Here are `bash` and `zsh` bindings to make Ctrl-x-dd remove the next word:

```
zsh% bindkey '\C-xdd' kill-word
bash$ bind '"\C-xdd": kill-word'
```

Conflicts between Prefixes and Editor Commands

You can actually make a prefix, such as the sequence Ctrl-x-d in the previous example, into a keybinding in its own right. In `bash`, the required command is as follows:

```
bash$ bind '"\C-xd": backward-kill-word'
```

This binding conflicts with the example using `kill-word` at the end of the previous section: After Ctrl-x-d, the shell doesn't know whether you're going to type a *d* next, to get `kill-word`. `bash` resolves this simply by waiting to see what you type next, so if you don't type anything, nothing happens; if you then type a *d*, you get `kill-word`, and if you type anything else, the shell executes `backward-kill-word`, followed by the editor commands corresponding to whatever else you typed.

`zsh` works like `bash` if you type something immediately after the prefix. However, it has a bit of extra magic to avoid the shell waiting forever to see what character comes after the Ctrl-x-d. The shell variable `KEYTIMEOUT` specifies a value in hundredths of a second for which the shell will wait for the next key when the keys typed so far form a prefix. If none turns up in that time, and the keystrokes so far are themselves bound to a command, then that command is executed. So let's consider the following:

```
zsh% bindkey '\C-xdd' kill-word
zsh% bindkey '\C-xd' backward-kill-word
```

When you've entered those commands, go up to the previous line, position the cursor at the start of `forward-word`, press Ctrl-x-d, and wait. You'll find that after a short time the previous word is deleted. That's because the default value of `$KEYTIMEOUT` is 40 in units of 100ths of a second. Why hundredths of a second? When `KEYTIMEOUT` was introduced, the shell didn't handle numbers with a decimal point. The unit had to be small enough to be useful, and a second was too large, so hundredths of a second was chosen.

Choosing a value for `KEYTIMEOUT` is a bit of an art: It needs to be small enough that you're not annoyed by the wait, but large enough so you have time to type a complete key sequence. There is an extra factor when you are using a shell over the network; the various keys can be sent out at any old time the network feels like, so `$KEYTIMEOUT` has to be large enough to cope with the sort of delays you get. There's no real way of predicting this; you just have increase the value of `KEYTIMEOUT` until everything seems to work. You set it by the following assignment:

```
zsh% KEYTIMEOUT=60
```

Of course, you can set it to whatever value you please. To make this setting permanent put it in a startup file. Remember, there are no spaces around the `=`.

Cursor Keys and vi Insert Mode

In case you think the key timeout feature that we introduced in the previous section is a bit pointless and you never want to use that feature, there is one case that is very common; indeed, if you use vi mode you come across it all the time. As we said, the keys can do two completely different sets of things in vi mode. To begin with, they simply enter text ("insert mode"), but you can switch to a mode in which (for example) the cursor keys Up, Down, Left, and Right are replaced by *k*, *j*, *h*, *l*, respectively ("command mode").

The usual way of switching to command mode is by pressing the Esc key. (`vi` users are trained to do this without thinking.) Unfortunately, on most modern terminals, the real cursor keys are turned into a key sequence, which starts with Esc. Most commonly, for example, The Up Arrow key sends `\e[A` or possibly `\eOA`. Therefore, when you press Esc to switch from insert mode to command mode, the shell will wait for `$KEYTIMEOUT` hundredths of a second. You can enter an editing command, though, and the shell will respond immediately.

Partly for this reason, `zsh` used not to make the cursor keys available in vi insert mode at all. However, the developers decided that beginning users were more likely to need the cursor keys than to be worried about the delay pressing Esc.

Keymaps

The vi insert mode and command mode are examples of *keymaps*. A keymap is a complete set of key bindings in use at the same time. Looked at this way, emacs mode is simply another keymap. These are the three you will meet most often, but in special operations, particularly in `zsh`'s completion system, you will come across others. Keymaps are completely independent; when you bind a key in a certain keymap, nothing happens to any other keymap.

There are two things you most often want to do with a keymap: switch to it, or bind keys in it. You've already seen how to switch to vi or emacs keymaps, but let's summarize the easiest ways in each case. These are shown in Table 4-11. (In the case of vi, you always start in the insert keymap.)

Table 4-11. *Command to Change between emacs and vi Keymaps*

	bash	zsh
emacs	`set -o emacs`	`bindkey -e`
vi	`set -o vi`	`bindkey -v`

To bind keys in a particular keymap, the clearest way is to use the keymap's name. In the case of emacs mode the name is just `emacs`. The two vi keymaps are called in `bash`, `vi-insert` and `vi-command` and in `zsh`, `viins` and `vicmd`. To bind a key in a keymap, use `bind -m` or `bindkey -M` followed by the keymap name, then the remaining arguments as normal.

For example, the commands below bind `B` to `backward-word` in the vi command keymap in each shell:

```
bash$ bind -m vi-command '"B": backward-word'
zsh% bindkey -M vicmd B backward-word
```

(Actually, *B* is already bound to something similar (a vi version of `backward-word`) in both shells already.) All keymaps in both shells are case-sensitive, so this is not the same as binding *b*. If you don't give a keymap, `bind` and `bindkey` always operate on the current one, which is usually `emacs` or `vi-insert`/`viins`. The shells never start up with `vi-command`/`vicmd` active. So to bind a key in the vi command keymap you need to specify the name of the keymap. There's a shortcut in `zsh`: `bindkey -a` works like `bindkey -M vicmd`. The `vicmd` keymap is also known as the *alternate* keymap. We've avoided this since it's less clear.

You can use the `-m` and `-M` options with other `bind` and `bindkey` commands. For example, you can list the vi command keymap with the following commands for `bash` and `zsh`, respectively:

```
bash$ bind -m vi-command -p
zsh% bindkey -M vicmd
```

You never lose information simply by switching keymaps. As you switch between `bindkey -e` and `bindkey -v` in `zsh` the bindings in each keymap are preserved, including any you added yourself. The commands simply switch between the sets—they don't perform any rebinding:

```
zsh% bindkey -e
zsh% bindkey '\C-xf' forward-word
zsh% bindkey -v
zsh% bindkey '\C-xf'
"^Xf" undefined-key
zsh% bindkey -e
zsh% bindkey '\C-xf'
"^Xf" forward-word
```

We bound the key sequence `'\C-xf'` in the emacs keymap. We switched to vi mode, and found it wasn't bound. Then we switched back to emacs mode, and found it was still there.

Creating Your Own Keymaps

In `bash` you are restricted to the existing keymaps. In `zsh`, however, you can create and use your own keymaps. You might use this for a special editing task where you intend to return to one of the standard keymaps such as `emacs` when you have finished. It's not completely obvious, but all you need to know can be summarized in a few sets of commands.

- You can create a new keymap by copying an existing one. This is usually more useful than creating one from scratch. The following example copies the keymap `emacs` into the new keymap `my-keymap`:

  ```
  zsh% bindkey -N my-keymap emacs
  ```

  Now you can treat `my-keymap` as your own private copy of the `emacs` map. Note it's a copy of the keymap in its current state, not with the default bindings.

- To create a new keymap from scratch, omit the final argument from the previous example:

  ```
  zsh% bindkey -N my-keymap
  ```

  You can use pretty much any name you are likely to want. However, it overwrites any existing keymap of that name, which may be a bad thing, so be careful. The keymap you create has no bindings at all, not even standard alphanumerics; you have to fill it from scratch. You can bind ranges of characters. For example, the following command forces all the lowercase characters to insert themselves as they do in an ordinary keymap such as Emacs:

  ```
  zsh% bindkey -R -M my-keymap 'a-z' self-insert
  ```

  The order, with `-R` first, is because `-M` is a bit tricky. In versions of `zsh` up to 4.0, the `my-keymap` is not as you'd expect an argument to the `-M` option, but the first non-option argument to the command. In other words, `-M` doesn't take an argument; it swallows up the first argument left after the option processing. This changed in version 4.1, but the order we've shown here will always work.

- Now you need to switch to your new keymap. This is done in a slightly obscure way by giving the new keymap the name `main` as an alias:

  ```
  zsh% bindkey -A my-keymap main
  ```

  (Remember the order; `main` comes last.) This is basically what the `bindkey -v` and `bindkey -e` commands do for you. When I talked about "the current keymap" earlier, I could have talked about `main` instead. If you want to switch back to the original keymap, you issue another `bindkey` command. For example, you can return to the `emacs` keymap with the command `bindkey -e`.

  Don't alias the `vicmd` keymap to `main`, as I just did. If you do that, both of the keymaps in use for `vi`-style editing are bound to `vicmd`. In that case you can't ever enter text! Instead, use `viins`. The effect of `bindkey -v` is to alias `viins` to `main`.

- To get rid of a keymap, you can delete it as follows:

  ```
  zsh% bindkey -D my-keymap
  ```

  but since the current keymap is lost when you leave the shell, you're unlikely to care enough to want to delete one.

■ **Tip** There's one keymap in `zsh` you probably don't want to come across. If you delete the `main` keymap, `zsh` uses a keymap called `.safe`. This keymap is specially restrictive; most of the commands we showed earlier for use with keymaps won't work. The only bindings in the `.safe` keymap are the characters that usually insert themselves and the Return and Linefeed keys (Ctrl-j if there isn't one so marked). That's enough to let you type a command and execute it to put things back to normal. You should immediately type something like `bindkey -e` to return to a better keymap.

Options for Editing

The shells have various other options for choosing the behavior of the editor. There's not much in common between many of these, so this section will be something of a ragbag of possibilities.

Customization of readline from within `bash` needs the shell to pass a complete string down to the readline library (the readline variables we mentioned in the sidebar "readline Configuration"). Let's show an example to make this clear. You may have noticed that the shell makes a beeping noise when you make a mistake or try something the shell won't permit. Many users find this effect infuriating and want to turn it off. In `bash`, this is done by setting the readline variable `bell-style` to `none`. Options to readline are set using `bind`; in this case the command is as follows:

```
bash$ bind 'set bell-style none'
```

In contrast, options in `zle` are just ordinary shell options. We'll introduce options properly in the section "Setting zsh Options with setopt" in Chapter 5. You set options using the `setopt` command. To turn off the beeping noise you need to set the option `nobeep` as follows:

```
zsh% setopt nobeep
```

An alternative to turning off the audible warning is to tell the shell or the terminal emulator to use a visible bell. This makes the screen flash at you. It's a little less annoying than the noise. In `bash`, you can set the readline variable `bell-style`:

```
bash$ bind 'set bell-style visible'
```

There's no such variable in `zsh`, but you can make a terminal emulator derived from `xterm` use a visible indication instead of beeping. This is done with a line in the `.Xdefaults` file mentioned in the section "Choosing a Terminal Emulator" earlier in this chapter:

```
*visualBell:    true
```

Strictly, this turns on the visual bell for any application that has it, but that might be what you want. Put `xterm` in front of the * to restrict it to `xterm`-like programs, including `rxvt` and `aterm`. Alternatively, simply start `xterm` using

```
xterm -vb
```

Add & to the end of the line if you try this from the shell command line; otherwise your current shell will wait for the `xterm` to exit.

`zsh` has a separate option, `hist_beep`, to control whether it beeps (or otherwise signals) when you try to go past the end of the history.

Finally, there are two `zsh` options to start up the line editor in a different way. The option `overstrike` starts the editor so that each printable character overwrites the character that was there before. (Normally, it pushes the rest of the line to the right.) You can swap in and out of this mode by using the `overwrite-mode` editor command (the one we bound to Insert in the section "Finding the Key Sequences for Function Keys" earlier in this chapter).

How the shell retrieves its history is controlled by many options. These will be explained in Chapter 6, since there are several other ways of getting at the same information and all are affected. Also, options for completion will be discussed in Chapter 10 although they have a bearing on the line editor.

Multiline Editing and the zsh Editor Stack

In Chapter 2 we saw various examples of entering commands on multiple lines. The shell presents a special prompt to show that it is expecting more input. When the input is complete, the shell executes the command. In `bash`, if you scroll back in the command history to reedit such a line, you will find that the whole command has been put on a single line. When you execute the line, it still has the same effect as the original because `bash` inserts special characters to make sure the line you are editing has the same effect as the original one. There's no way of editing it in the original form, though.

However, `zsh` has powerful handling for editing commands that span more than one line. We already explained how the shell would prompt you if a command wasn't finished. In that case, you are effectively editing a new line. For example,

```
zsh% cp file1 file2 file3 file4 \
> /disk1/storage/pws/projects/utopia/data/run32
```

If you use the cursor keys on the second line, the editor behaves pretty much as if you were editing a separate command, with the previous line already in the history.

However, it's possible to edit two lines at once. In fact, that's how continuation lines are put into the history. If you press Return and then Up Arrow, you will find both lines appear, without the continuation prompt >. Go up once more, and you go from the `bar` to the `foo` line. (We're assuming you use the normal binding for the cursor keys, `up-line-or-history`. There's another possible binding, `up-history`, which always takes you back in the history, not through the set of lines displayed for editing.) Go up again, and you are taken to the previous command.

There are two ways of editing multiline commands without the continuation prompt:

1. Use Esc-Return where you would normally press Return on its own. It puts the cursor onto a new line with no prompt. You can add as many lines as you want this way, and they will go into the command history together. Be careful to note that unless you use a \ at the end of the previous line, the next line will be treated as a new command, even though the previous line hasn't yet been executed. For example,

   ```
   zsh% mv file1 old_file1<escape><return>
   mv file2 old_file2<return>
   ```

   There are two complete commands there. We executed both at once so that the two files were renamed at (almost) the same time. We could have put a semicolon between them (remember the semicolon connects a list of different commands on the same line), but this is a little easier to read.

2. The other way uses `push-line-or-edit`. We'll bind that to a key sequence, to avoid typing a very long command after Esc x:

   ```
   zsh% bindkey '\eq' push-line-or-edit
   ```

   Now go back and again type the following:

   ```
   zsh% cp file1 file2 file3 file4 \
   > /disk1/storage/pws/projects/utopia/data/run32
   ```

   But this time press Esc q at the end instead of Return. You'll see the continuation prompt magically disappear and you will be editing the complete command.

The Buffer Stack

The `push-line-or-edit` command has two functions. In the previous section we showed the `-or-edit` part. Let's move on to the other one.

Suppose you have forgotten to do something you need to, and are halfway through typing the next command when you remember. For example, you are in the middle of a long `ls` command and forget what options you need:

```
ls -L here/there/everywhere
```

Or should that be `-l` instead of `-L`? Press Esc-q; this time the line completely disappears. Now execute the following command:

```
man ls
```

This confirms that you meant `-l` to list in the long format, not `-L`, which follows links. The point of the Esc-q becomes apparent when you quit reading the manual: The complete previous command magically reappears.

You are using `zsh`'s *buffer stack*. Every time you press Esc-q on a complete buffer, the buffer is pushed onto the end of the stack. Every time you press Return, the last piece of text pushed onto the end of the stack is popped off and loaded back into the line editor. (You can also summon the last value explicitly with Esc-g.) By the way, Esc-q has this `push-line` effect by default in emacs mode; however, it doesn't have the extra `-or-edit` function unless you bind that as described earlier.

A Quick Way of Getting Help on a Command

The use of `push-line` we explained is so common that there is a special command that does it all for you. You can look up the documentation for a command you are entering by simply pressing Esc-h, without clearing the command line. This pushes the line for you, then runs the command `run-help`, which by default is an alias for `man`. Afterwards, the command line appears from the buffer stack.

You can customize `run-help` to be even more helpful, as we describe in the sidebar "Customizing zsh's run-help Command."

CUSTOMIZING ZSH'S RUN-HELP COMMAND

You aren't stuck with the default behavior of `run-help`. The shell comes with a function you can use instead, which should be installed into your function search path. In that case you can use it by entering the following:

```
zsh% unalias run-help
zsh% autoload -U run-help
```

Now when you press Esc-h, you get more help. If you do this a lot with the commands that are part of the shell, you can get the `run-help` function to show you the documentation for the individual command, but you need to do some setting up first. Somewhere create a directory called `Help`, change into it, and run the Perl script `helpfiles` provided with the `zsh` distribution in the `Util` subdirectory. The script isn't installed with the shell, but you only need to run it once. There's some documentation at the top of the script (run `more` on the file to see it). To make it work, you need to pass the `zshbuiltins` manual into it. Here's an example that creates the help files in the directory `~/zsh-help`:

```
zsh% mkdir ~/zsh-help
zsh% cd ~/zsh-help
zsh% man zshbuiltins | col -bx | perl ~/src/zsh-4.2.0/Util/helpfiles
```

If you see error messages, make sure your `zsh` manual pages are installed properly. Sometimes you can replace `col -bx` with `colcrt` and get slightly better effects, depending on the system. Both commands strip the terminal control characters from the output of man to turn it into plain text. That's the bit you only need to do once. Then you need to tell the `run-help` function where to find those files; this goes in `~/.zshrc` (without the prompt, of course):

```
zsh% HELPDIR=~/zsh-help
```

along with the two-line setup for `run-help` shown previously. Now pressing Esc-h on a line starting with a shell built-in will show you the manual entry for the built-in.

Keyboard Macros

Many editors provide keyboard macros, where you record a set of keystrokes by typing them in, and they can be played back when you need them. `bash` offers this, but `zsh` doesn't; the nearest `zsh` has is the `bindkey -s` feature we described earlier. That will do everything a recorded macro can; you just have to remember the names for the keys instead of typing them. This is harder at first, but easier to edit if you make a mistake.

The readline keyboard macros available in `bash` work (surprise, surprise) much like those in Emacs. You start recording by pressing Ctrl-x (and after typing all the necessary keys end it by pressing Ctrl-x). You then play back the macro by pressing Ctrl-x-e. As a trivial example, find a command line and type the following keys:

```
<ctrl-x><(><escape><b><escape><b><ctrl-x><)>
```

Now every time you press Ctrl-x-e the cursor moves left two words.

Other Tips on Terminals

We'll finish the chapter with a few remarks regarding interaction between editing command lines and the terminal. Unix terminal drivers—the part of the system that handles input and output for a terminal or terminal emulator—are slightly weird things. We say a little about why in the sidebar "The Terminal Driver." We've already met some occasions where certain special keys are swallowed up. They were Ctrl-d at the start of the line, which meant end-of-file (EOF), Ctrl-s to stop output, and Ctrl-q to start it again.

There's a program called `stty` that controls these settings, as well as a lot of others, many of which, we can assure you, you will not want to know about. However, it is useful to know the basics of `stty` if you are interested in customizing your own environment. The following lists the settings:

```
% stty -a
speed 38400 baud; rows 24; columns 80; line = 4;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

This is from a Linux system using the GNU tools. Your output could be completely different, since other versions of Unix can have very different setups. However, `stty` is a program provided by the system, so it doesn't matter if you're using `bash` or `zsh` or any other shell.

The settings we're most interested in here are the three lines starting with `intr = ^C`. This means that the interrupt character is set to Ctrl-c. We met this feature and used it to interrupt a running program in the section "Starting and Stopping Processes: Signals, Job Control" in Chapter 3, but we didn't say you could change the key. You can do so like this:

```
% stty intr '^t'
```

(The quotes are recommended, since `zsh`'s `extended_glob` option makes `^` special. By the way, control characters are case insensitive; `^t` and `^T` are the same.) This makes Ctrl-t the interrupt character. You can try it by typing the following command:

```
% sleep 10
<ctrl-t>
```

The `sleep 10` makes nothing happen for 10 seconds, unless you interrupt it early with Ctrl-t. Note you can only have one key doing each special task, so we freed Ctrl-c at the same time as we started using Ctrl-t. This is different from the way shells work: the shells remember what single function is assigned to each key, for every key, while the terminal driver remembers for each special function what key is assigned to it.

It's also possible to turn off special keys—that's what the <undef> after eol meant. The traditional way of doing this is with the sequence ^-:

```
% stty intr '^-'
```

Yet some versions of stty understand the more logical undef, too. Obviously, doing this means you can't send an interrupt from the keyboard until you redefine intr.

As you'll see in the sidebar "The Terminal Driver," not all keys known to stty are special inside zsh. On my system, at least, you only need to worry about intr, eof, start, and stop inside the shell. This is probably just as well since some of the others are rather obscure. Hence you can bind an editing command, for example, to Ctrl-z, despite the fact that it is assigned to susp by stty. (This is one of the more useful special keys used for job control; see the section "Starting and Stopping Processes: Signals, Job Control" in Chapter 3.)

THE TERMINAL DRIVER

Why are there all these strange effects associated with terminals, and why do we need the stty command to control them?

In the early days of Unix, shells had no editing capabilities of their own, not even the basic ability to delete the previous character. However, a program existed that read the characters typed by the user, and sent them to the shell: the *terminal driver*. It gradually developed a few simple editing features of its own, until it grew to include all the features you can see from the output of stty -a in the main text.

By default, the terminal driver accepts input a line at a time; this is sometimes known as "canonical" input mode. For commands that don't know about terminals, such as simple shells without their own editors, the terminal driver usually runs in a mode sometimes known as "cooked." (This is Unix humor; when not in "cooked" mode, the terminal is in "raw" mode.) Here, all the special keys are used. This allows you some very primitive editing on a line of input. For these simple editing features, the command stty shown in the main text acts as a sort of bind or bindkey command.

For example, let's try cat, which simply copies input to output. Type the following:

```
% cat
this is a line<ctrl-u>
```

The line disappears; that's because of the kill stty setting. If it didn't work, try setting the following first:

```
% stty kill '^u'
```

This is handled entirely by the terminal driver; neither the program (cat) nor the shell knows anything about it. In "cooked" mode, the terminal passes a complete line to the program when you press Return. So cat never saw what you typed before the Ctrl-u.

Since not all programs want the terminal driver to handle their input, the terminal has other modes. The shell itself uses "cbreak" mode (not quite equivalent to "raw" mode), which means many of the characters which are special in "cooked" mode are passed straight through to the shell. Hence when you press Backspace in either bash or zsh, it's the shell, not the terminal driver, that deals with it.

Working Around Programs with Bad Terminal Handling

Every now and then, you may find a program that reacts badly to the terminal settings and needs you to issue an `stty` to fix things up. This is usually a bug in the program, but `zsh` provides you with a way to work around the problem: assigning a command line for `stty` to the environment variable `STTY`. In practice you would use it in a way similar to the following example:

```
zsh% STTY="intr '^-'" sleep 10
```

That turns off the keyboard interrupt character for the life of that one command. It's not worth remembering the details of this feature; just come back here to look if you suspect terminal problems with a program.

`zsh` has another way of dealing with programs that mess about with `stty` settings: the built-in command `ttyctl`. You use this to "freeze" or "unfreeze" the terminal settings:

```
zsh% ttyctl -f
```

After this command, the terminal is frozen, which means if an external program changes one of the settings that would show up in `stty -a`, the shell will immediately put it back so that any future programs won't see that change. The following command has the opposite effect:

```
zsh% ttyctl -u
```

Now the terminal is unfrozen, and changes from an external program will show up later. Note that "an external program" includes `stty` itself. Suppose you enter the following commands:

```
zsh% ttyctl -f
zsh% stty intr '^-'
```

Because the terminal settings are in their frozen state, the `stty` setting won't take effect. You need to unfreeze the terminal, then run `stty`, then freeze it again.

Variables That Affect the Terminal

Both shells handle resizes to the terminal window—if you extend your terminal emulator to the right, the shell will know it has extra columns on the display, and so on. For future reference, you might like to know that the width and height of the terminal are stored in the variables `COLUMNS` and `LINES`, respectively.

There's one other variable in `zsh` that's worth mentioning here. `BAUD` is set to the speed of the connection between your terminal and the actual computer in bits per second. Usually you don't need to set this, but there are two reasons why you might. First, if you have a really slow line, set `BAUD` to the proper speed; for example if your connection speed is 2400 bits per second, the setting

```
zsh% BAUD=2400
```

tells `zsh` that your terminal will be slow to respond. Conversely, if the terminal appears to be doing something very slowly, you may need to increase the value. The reason is that when `BAUD` has a low value, the shell tries to change the screen a little bit at a time, to keep it consistent.

When the value is larger, it will try to be more sophisticated, making more changes at once. Setting BAUD to 38400 as follows is usually good enough:

```
zsh% BAUD=38400
```

The number doesn't have to be set to the actual speed of the link between your keyboard and the shell.

When Unix Gets a Bit Confused about Keys

There are two occasions when the keys you type may look to the system like some other keys. If you're lucky enough never to have these problems, you can safely skip this section: the first of the two issues below isn't noticed by a lot of users, and the second depends on your system's settings as well as the way the system handles your keyboard.

The Carriage Return and Linefeed Confusion

There is rather a lot of confusion on Unix systems about the difference between Return (more properly, carriage return, sent by Ctrl-m or the Return key) and Linefeed (Ctrl-j, or possibly a special key on some Unix keyboards).

When the terminal is in "cooked" mode—in other words the shell or some intelligent interface is not running—the terminal actually turns an input Return into Linefeed. This is controlled by the setting `stty icrnl`, which is usually on (you can turn it off with `stty -icrnl`). The shell protects you from this, but it does make it difficult for *typeahead input* to distinguish the two characters. ("Typeahead input" is where you don't wait for a program to finish before typing a new command line.) If you have Return and Linefeed bound to different commands you might notice this. Luckily, they normally do the same thing.

The Backspace and Erase Confusion

You might have thought deleting the previous character using the Erase key was simple enough. However, this is not always the case. On traditional Unix keyboards, the key at the top right of the largest group of keys, marked as "delete" or "erase" or something similar, generates a certain character usually written as Ctrl-? (`^?` to `stty`) and described as "delete". However, on a PC keyboard the key at that position, often marked with a backward arrow or something similar, sends a backspace, which is the same as Ctrl-h. To make it worse, some systems where the key would normally send Ctrl-h intercept it and send Ctrl-? instead.

Luckily, both `zsh` and `bash` will delete the previous character when they receive either of those keys, so much of the time you don't need to know. The time you do need to know which is which is when you are entering input that is not under the control of the shell. As explained earlier, `stty` only handles one special character for any function, in this case the function named `erase`, so the other won't work in cooked mode—for example, when you are entering text to `cat`. The usual indication something has gone wrong is a series of `^H` or `^?` characters appearing when you try to delete. It is a strange fact about the universe that the erase value is almost always set incorrectly by default. However, putting it right is simple. One of the following will set the value you want:

```
% stty erase '^?'
```

or

```
% stty erase '^h'
```

As the final part of this confusion, the key on a PC keyboard marked Delete or Del is different again. On a full-sized keyboard you usually find it in the group of six over the cursor keys. It probably does nothing by default and you have to bind it yourself using the tricks in the section "Finding the Key Sequences for Function Keys" in this chapter.

Summary

In this chapter, you learned about

- Starting a terminal window with a shell running in it
- How continuation of command lines works
- The line editor, and its various modes of operation using different keystrokes
- Moving around the command line
- Finding previous command lines
- Deleting and restoring chunks of text
- Defining your own keystrokes for the editor's commands and setting options for the editor
- Using function keys, and other special keys, to execute commands
- How the various sets of keystrokes are arranged in keymaps
- The tricks you can use for handling multiline input
- Defining keyboard macros
- Avoiding problems caused by the part of the operating system that controls the terminal, known as the "terminal driver"

In the next chapter we will take a look at what happens when the shell starts. Knowing that will help you to customize the shell. We will also talk about shell options, the simplest way of changing the behavior. Also, we introduce Cygwin, a way of letting the shell work under Windows with a good deal of the Unix look and feel.