4

Text Blocks and Multiple Files

THIS CHAPTER SHOWS YOU HOW TO DEAL with larger text blocks. This includes the commands that enable you to define a large text block as well as perform cut, paste, and copy operations.

With most editors, you can just cut and paste. However, the *Vim* editor has the concept of a register. This enables you to hold data for multiple cut, copy, or paste operations. Most other editors are limited to a single cut/paste clipboard. With the *Vim* registers you get more than 26 clipboards.

One of the strengths of UNIX is the number of text manipulation commands it provides. This chapter shows you how to use the filter command to take advantage of this power to use UNIX filters to edit text from within *Vim*.

Up until now, you have worked with single files in this book. You will now start using multiple files. This will enable you to perform the same edits on a series of files, and to cut and paste between files.

This chapter discusses the following topics:

- Simple cut-and-paste operations (in *Vim* terms, delete and put)
- Marking locations within the text
- Copying text into a register using the yank commands
- Filtering text
- Editing multiple files

Cut, Paste, and Copy

When you delete something with the **d**, **x,** or another command, the text is saved. You can paste it back by using the **p** command. (The technical name for this is a *put*).

Take a look at how this works. First you will delete an entire line with the **dd** command, by putting the cursor on the line you want to delete and pressing **dd**. Now you move the cursor to where you want to place the line and use the **p** (put) command. The line is inserted on the line following the cursor. Figure 4.1 shows the operation of these commands.

Because you deleted an entire line, the **p** command placed the text on the line after the cursor.

If you delete part of a line (a word with the **dw** command, for instance), the **p** command puts it just after the character under the cursor (see Figure 4.2).

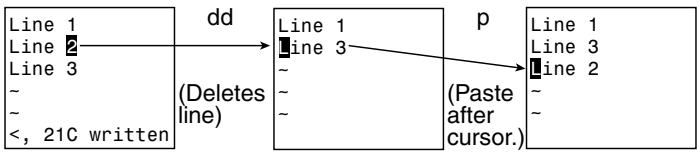

Figure 4.1 Deleting (cutting) and putting (pasting).

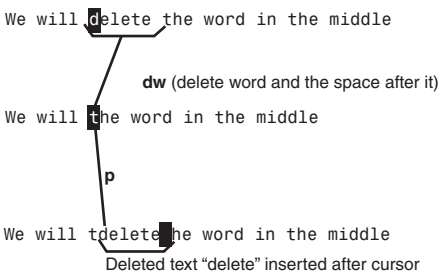

Figure 4.2 Deleting a word and putting back again.

Character Twiddling

Frequently when you are typing, your fingers get ahead of your brain. The result is a typo such as *teh* for *the*. The *Vim* editor makes it easy to correct such problems. Just put the cursor on the *e* of *teh* and execute the command **xp**. Figure 4.3 illustrates this command. This works as follows:

x Deletes the character 'e' and places it in a register.

p Puts the text after the cursor, which is on the 'h'.

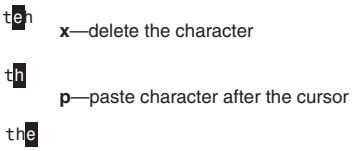

Figure 4.3 **Character twiddling with** `xp`**.**

More on "Putting"

You can execute the **p** command multiple times. Each time, it inserts another copy of the text into the file.

The **p** command places the text after the cursor. The **P** command places the text *before* the cursor. A count can be used with both commands and, if specified, the text will be inserted *count* times.

Marks

The *Vim* editor enables you to place marks in your text. The command **ma** marks the place under the cursor as mark *a*. You can place 26 marks (*a* through *z*) in your text. (You can use a number of other marks as well.)

To go to a mark, use the command `` `*mark* ``, where *mark* is the mark letter (and ` is the backtick or open single-quote character).

The command `'mark` (single quotation mark, or apostrophe) moves you to the beginning of the line containing the mark. This differs from the `` `*mark* `` command, which moves you to the marked line and column.

The `'mark` command can be very useful when deleting a long series of lines. To delete a long series of lines, follow these steps:

1. Move the cursor to the beginning of the text you want to delete.
2. Mark it using the command **ma**. (This marks it with mark *a*.)
3. Go to the end of the text to be removed. Delete to mark *a* using the command **d'a**.

   Note: There is nothing special about using the *a* mark. Any mark from *a* to *z* may be used.

There is nothing special about doing the beginning first followed by the end. You could just as easily have marked the end, moved the cursor to the beginning, and deleted to the mark.

One nice thing about marks is that they stay with the text even if the text moves (because you inserted or deleted text above the mark. Of course, if you delete the text containing the mark, the mark disappears.

Where Are the Marks?

To list all the marks, use the following command:

```
:marks
```

Figure 4.4 shows the typical results of such a command.

The display shows the location of the marks *a* through *d* as well as the special marks: *'*, *"*, *[*, and *]*.

Marks *a* through *d* are located at lines 1, 8, 14, and 25 in the file.

The special marks are as follows:

'	The last place the cursor was at line 67 of the current file
"	Line 1 (we were at the top of the file when last closed it)
[	The start of the last insert (line 128)
]	The end of the insert (line 129)

To view specific marks, use this command:

```
:marks args
```

Replace *args* with the characters representing the marks you want to view.

```
 *          the data from an input
 * (.c) file.
 */
struct in_file_struct {
:marks
mark line  col  file/text
 '      67    0  *^I^I^I   into the "bad" list^I^I*
 a       1    0  #undef USE_CC^I/* Use Sun's CC com
 b       8    1  * Usage:^I^I^I^I^I^I*
 c      14    1  *^I^I^I   (default = proto_db)^I^I
 d      25    1  *^I--quote^I^I^I^I^I^I*
 "       1    0  #undef USE_CC^I/* Use Sun's CC com
 [     128   42  * in_file_struct -- structure that
 ]     129   12  *          the data from an input
Press RETURN or enter command to continue
```

Figure 4.4 **:marks**.

Yanking

For years, I used a simple method for copying a block of text from one place to another. I deleted it using the **d** command, restored the deleted text with the **p** command, and then went to where I wanted the copy and used the **p** to put it into the text.

There is a better way. The **y** command "yanks" text into a register (without removing it from the file). The general form of the **y** command is **y***motion*. It works just like the delete (**d**) command except the text is not deleted. And the shorthand **yy** yanks the current line into the buffer.

(Note: Most other editors call this a "copy" operation.)

Take a look at how you can use this command to duplicate a block of text. First go to the top of the text to be copied and mark it with **ma**. Then go to the bottom and do a **y'a** (yank to mark *a*).

Now go to where the copied text is to be inserted and put it there using the **p** command.

Figure 4.5 shows these commands in action.

Yanking Lines

The **Y** command yanks a single line. If preceded by a count, it yanks that number of lines into the register. You might have expected **Y** to yank until the end of the line, like **D** and **C**, but it really yanks the whole line.

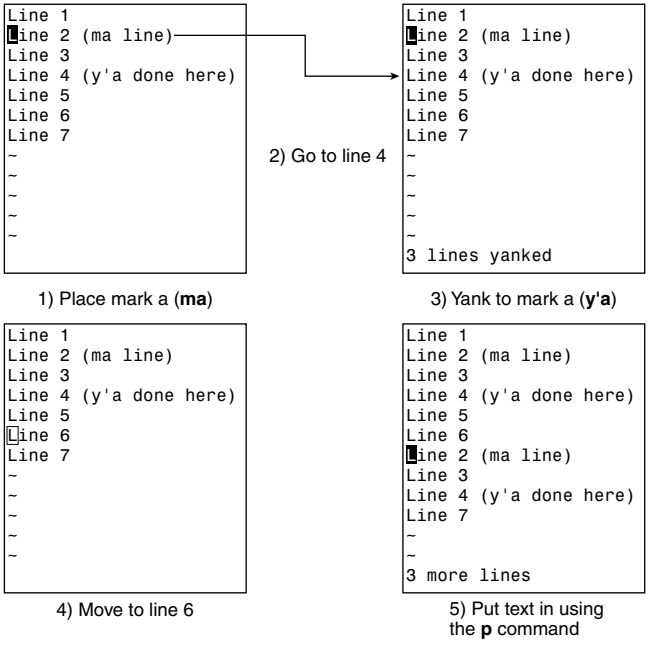

Figure 4.5 Yank (copy) and put (paste).

Filtering

The **!***motion command* takes a block of text and filters it through another program. In other words, it runs the system command represented by *command*, giving it the block of text represented by *motion* as input. The output of this command then replaces the selected block.

Because this summarizes badly if you are unfamiliar with UNIX filters, take a look at an example. The **sort** command sorts a file. If you execute the following command, the unsorted file `input.txt` will be sorted and written to `output.txt`. (This works on both UNIX and Microsoft Windows.)

```
$ sort <input.txt >output.txt
```

Now do the same thing in *Vim*. You want to sort lines 1 through 10 of a file. You start by putting the cursor on line 1. Next you execute the following command:

```
!10G
```

The **!** tells *Vim* that you are performing a filter operation. The *Vim* editor expects a motion command to follow indicating which part of the file to filter. The **10G** command tells *Vim* to go to line 10, so it now knows that it is to filter lines 1 (the current line) through 10 (**10G**).

In anticipation of the filtering, the cursor drops to the bottom of the screen and a **!** prompt displays. You can now type in the name of the filter program, in this case **sort**.

Therefore, your full command is as follows:

```
!10Gsort<Enter>
```

The result is that the **sort** program is run on the first 10 lines. The output of the program replaces these lines.

The **!!** command runs the current line through a filter. (I have found this a good way to get the output of system commands into a file.)

I'm editing a `readme.txt` file, for example, and want to include in it a list of the files in the current directory. I position the cursor on a blank line and type the following:

```
!!ls
```

This puts the output of the **ls** command into my file. (Microsoft Windows users would use **dir**.)

Another trick is to time stamp a change. To get the current date time (on UNIX), I use the following command:

```
!!date
```

This proves extremely useful for change histories and such.

> **Note**
> Using *!!* like this is technically not filtering because commands like *ls* and *date* don't read standard input.

Editing Another File

Suppose that you have finished editing one file and want to edit another file. The simple way to switch to the other file is to exit *Vim* and start it up again on the other file.

Another way to do so is to execute the following command:

```
:vi file
```

This command automatically closes the current file and opens the new one. If the current file has unsaved changes, however, *Vim* displays a warning message and aborts the command:

```
No write since last change (use ! to override)
```

At this point, you have a number of options. You can write the file using this command:

```
:write
```

Or you can force *Vim* to discard your changes and edit the new file using the force (**!**) option, as follows:

```
:vi! file.txt
```

> **Note**
>
> The **:e** command can be used in place of **:vi**. The fact that these commands are equivalent has led to a flame war between Steve Oualline, who prefers **:vi** and Bram Moolenaar, who prefers **:e**. (Okay, it was limited to three slightly discordant emails, but it's hard to introduce real drama in a book like this.)

The *:view* Command

The following command works just like the **:vi** command, except the new file is opened in read-only mode:

```
:view file
```

If you attempt to change a read-only file, you receive a warning. You can still make the changes; you just can't save them. When you attempt to save a changed read-only file, *Vim* issues an error message and refuses to save the file. (You can force the write with the **:write!** command, as described later in this chapter.)

Dealing with Multiple Files

So far the examples in this book have dealt with commands that edit a single file. This section introduces you to some commands that can edit multiple files.

Consider the initial **Vim** command, for example. You can specify multiple files on the command line, as follows:

```
$ gvim one.c two.c three.c
```

This command starts *Vim* and tells it that you will be editing three files. By default, *Vim* displays just the first file (see Figure 4.6).

To edit the next file, you need to change files using the **:next** command. Figure 4.7 shows the results. Note that if you have unsaved changes in the current file and you try to do a **:next**, you will get a warning message and the **:next** will not work.

You can solve this problem in many different ways. The first is to save the file using the following command:

```
:write
```

In other words, you can perform a **:write** followed by a **:next**.

The *Vim* editor has a shorthand command for this. The following command performs both operations:

```
:wnext
```

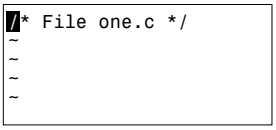

Figure 4.6 Editing the first of multiple files.

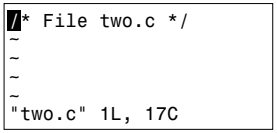

Figure 4.7 **:next**.

Or, you can force *Vim* to go the next file using the force (**!**) option. If you use the following command and your current file has changes, you will lose those changes:

```
:next!
```

Finally, there is the **'autowrite'** option. If this option is set, *Vim* will not issue any `No write...` messages. Instead, it just writes the file for you and goes on. To turn this option on, use the following command:

```
:set autowrite
```

To turn it off, use this command:

```
:set noautowrite
```

You can continue to go through the file list using the following command until you reach the last file:

```
:next
```

Also, the **:next** command can take a repeat count. For example, if you execute the command

```
:2 next
```

(or **:2next**), *Vim* acts like you issued a **:next** twice.

Which File Am I On?

Suppose you are editing a number of files and want to see which one you are on. The following command displays the list of the files currently being edited:

```
:args
```

The one that you are working on now is enclosed in square brackets. Figure 4.8 shows the output of the command.

```
/* File two.c */
~
~
~
~
one.c [two.c] three.c
```

Figure 4.8 Output of **:args**.

This figure shows three files being edited: `one.c`, `two.c`, and `three.c`. The file currently being editing is `two.c`.

Going Back a File

To go back a file, you can execute either of the following commands:

```
:previous
```

or

```
:Next
```

These commands act just like the **:next** command, except that they go backward rather than forward.

If you want to write the current file and go to the previous one, use either of the following commands:

```
:wprevious
:wNext
```

Editing the First or Last File

You might think that **:first** would put you at the first file and **:last** would edit the last file. This makes too much sense for use with computers. To start editing from the first file, no matter which file you are on, execute the following command:

```
:rewind
```

To edit the last file, use this command:

```
:last
```

Note: Bram has promised to add a **:first** command in the next release of *Vim*.

Editing Two Files

Suppose that you edit two files by starting *Vim* with the following:

```
$ gvim one.c two.c
```

You edit a little on the first file, and then go to the next file with the following:

```
:wnext
```

At this point, the previous file, `one.c`, is considered the *alternate file*. This has special significance in *Vim*. For example, a special text register (#) contains the name of this file.

By pressing **CTRL-^**, you can switch editing from the current file to the alternate file. Therefore, if you are editing `two.c` and press **CTRL-^**, you will switch to `one.c` (`two.c` becoming the alternate file). Pressing **CTRL-^** one more time switches you back.

Suppose that you are editing a bunch of files, as follows:

```
$ gvim one.c two.c three.c
```

The command *count***CTRL-^** goes to the count file on the command line. The following list shows the results of several **CTRL-^** commands:

1 CTRL-^	one.c	
2 CTRL-^	two.c	
3 CTRL-^	three.c	
CTRL-^	two.c	(previous file)

> **Note**
>
> When you first start editing (file `one.c`) and press `CTRL-^`, you will get an error message: `No alternate file`. Remember the alternate file is the last file you just edited before this one (in this editing session). Because `one.c` is the only file edited, there is no previous file and therefore the error message.