CHAPTER 9

Kill and Delete; Move and Copy; Correct Mistakes; Spelling; Fill

Section 9.1: Kill and Delete: Two Ways to Erase Text

There are a variety of Emacs commands you can use to erase text from the buffer. As a convenience, Emacs will remember the erased text so you can insert it back into the buffer if you want. For example, to move a paragraph from one place to another, you can erase it and then insert it somewhere else.

Once you start using these commands, it won't be long before you see there is no need for you to worry about preserving everything. For example, when you erase a single character or an empty line, there is really no need to save it. However, when you erase several lines, a whole paragraph, or an entire region, it does make sense to keep a copy of the text. In fact, this is how Emacs works.

Emacs is designed so that some commands save the erased text while others do not. In general, Emacs saves the text from commands that erase more than a single character. Such commands are called KILL COMMANDS. Emacs does not save the text from commands that erase only a single character or whitespace. These commands are called DELETE COMMANDS. (As we discussed in Section 8.3, "whitespace" refers to the **SPC**, **TAB** or **RET** characters, which are described in Section 4.4.)

The custom within Emacs is to use these two terms — kill and delete — as verbs. When we say we KILL some text, we imply that the text is being saved. When we say that we DELETE some text, we imply that the text is not saved. In other words, whatever you delete is gone for good. Whatever you kill can be resurrected.

In the next two sections, we will talk about the delete commands and then the kill commands. I will then explain how Emacs saves killed text, and what commands you can use to insert such text back into the buffer. You will find that these commands are especially useful, as you can use them to copy or move text from one part of the buffer to another, or even between two different buffers.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_9

Section 9.2: Commands to Delete Text

As I explained in Section 9.1, delete commands erase only a single character or whitespace. These commands are shown in Figure 9-1. Remember, unlike the kill commands, delete characters are not saved for later recall. Still, you can't lose more than one character at a time.

Command	Description
BS	Delete one character to the left of cursor
DEL	Delete one character at the position of cursor
C-d	Same as **DEL**
M-	Delete spaces & tabs around point
M-SPC	Delete spaces & tabs around point; leave one space
C-x C-o	Delete blank lines around current line
M-^	Join two lines (delete **RET** + surrounding spaces)

FIGURE 9-1. Commands to delete text.

We have already covered the first three commands. **BS** <Backspace> erases the character to the left of the cursor. This is the key to use when you are typing and you need to back up, delete, and fix a mistake. **DEL** <Delete> or **C-d** erases the character that is at the cursor position. More precisely, **BS** erases to the left of point, and **DEL** and **C-d** erase to the right of point.

As with other Emacs commands, you can use a prefix argument (explained in Section 8.4) with these two commands to operate on more than one character at a time. For example, to erase 5 characters to the left, you can use **ESC 5 BS**; to erase 18 characters to the right, you can use **ESC 18 DEL** or **ESC 18 C-d**.

When you use such commands, you are erasing more than one character and, as such, you are killing and not deleting. For this reason, Emacs does save the erased text when you use **DEL** or **C-d** to erase more than one character at a time.

Secret Note

Here is a way to make a bit of money for yourself.

Take this book and go to a bar where Emacs people hang out. Look for some people who are learning Emacs and practicing on their laptops, and sit down next to them. Open the book so that Figure 9-2 in Section 9.3 is clearly visible and casually leave it where the people next to you can see it. This figure contains a summary of all the Emacs kill commands. Pretend you are not looking at the book.

Next, strike up a casual conversation with the people next to you and carefully work the topic around to the Emacs kill commands. Offer to bet them a small sum of money that they can't think of a way to kill text without pressing any upper- or lowercase letters.

When they sneak a look at the book, pretend you don't notice. When they look at Figure 9-2, they will see that **M-BS** (<Meta-Backspace> = <Alt-Backspace>) is a kill command and, thinking that you are an easy mark, they will accept the bet. When they press **M-BS**, pretend to be annoyed with yourself, pay off the bet, and tell them you would like a chance to win back your money. Offer them a much larger bet that you can kill text without using **M-BS** and without pressing any alphabetic keys.

Now, look the other way for a second, which will give them another chance to check out Figure 9-2. Aside from **M-BS**, they will not see any other kill commands that do not use a letter of the alphabet.

As soon as they take your bet, press **ESC 5 DEL** and clean up.

To continue, the **M-** (<Meta-Backslash>) command erases any **space** or **tab** characters that happen to be on either side of point. This command provides a quick way to clean up a section of whitespace. For example, say that you have typed the following text and the cursor is under one of the spaces between **tea** and **ch**:

```
Everything we tea      ch you is true.
```

If you press **M-**, Emacs will erase all the surrounding spaces. The line now looks like:

```
Everything we teach you is true.
```

The **M-SPC** (<Meta-Space>) command is similar, except that it leaves exactly one space. Here is an example. You have just typed the line:

```
The sentence above is   only partially correct.
```

You would like to erase the extra spaces. Move the cursor to one of the spaces between **is** and **only** and press **M-SPC**. The line is changed to the following:

```
The sentence above is only partially correct.
```

■ **Note**

If you are using a virtual terminal, you won't have a problem with **M-SPC**. However, with a terminal window (see Section 2.6), you may find that **M-SPC** doesn't work. This is because, in some GUIs, the sequence <Alt-Space> opens the terminal window's main context menu.

If this is a problem for you, simply use **ESC SPC** (<Esc> <Space>) instead of **M-SPC** (<Alt-Space>).

The **C-x C-o** command performs the analogous operation for blank lines. This command will erase all the blank lines surrounding the current line. For example, say that the buffer contains the following text:

```
Everything we teach you is true.


The sentence above is only partially correct.

Don't believe everything you read.
```

You would like only a single blank line between each line of text. Move to one of the blank lines following the first line, and press **C-x C-o**. The extra blank lines will be erased and you will be left with the following:

```
Everything we teach you is true.

The sentence above is only partially correct.

Don't believe everything you read.
```

Finally, the **M-^** command joins two lines into one long one. This command joins the current line to the one immediately above it, while leaving a single space between the two groups of text. Any extra spaces (at the end of the first line or at the beginning of the second line) are removed. For example, say that you have the following lines of text:

```
This is the first sentence.
      This is the second sentence.
```

You want to join these two lines. Move the cursor to the second line and press **M-^**. You will now have one long line:

```
This is the first sentence. This is the second sentence.
```

The cursor will be at the place where the lines were joined, in this case, at the space between the two sentences.

Section 9.3: Commands to Kill Text

Most of the commands that erase text are kill commands. When you use a kill command, Emacs saves the text that is erased in case you want to insert it back into the buffer. Figure 9-2 summarizes the various kill commands.

Command	Description
C-k	Kill from cursor to end of line
M-d	Kill a word
M-BS	Kill a word backward
M-k	Kill from cursor to end of sentence
C-x BS	Kill backward to beginning of sentence
C-w	Kill the region
M-z *char*	Kill through next occurrence of specified character

FIGURE 9-2. Commands to kill text.

The **C-k** command erases all the characters from the cursor to the end of the line. More precisely, **C-k** erases from point to the end of the line. (Remember, point is between the cursor and the character to its left. See Section 8.1.) If you are at the beginning of a line when you press **C-k**, it will erase all the characters on the line. If you are on a blank line, **C-k** will erase the line itself. Thus, you can erase a line completely by (1) moving to the beginning, (2) erasing all the characters, and then (3) erasing the line itself. The sequence to do this is **C-a C-k C-k**.

Note To erase an entire line: if you are at the beginning of the line, press **C-k C-k**. If you are not at the beginning of the line, press **C-a C-k C-k**.

There are two kill commands that erase a word. The **M-d** command erases from point to the end of the word. The **M-BS** command erases from point to the beginning of the word. Remember that point lies between the cursor and the character to its left. Thus, **M-BS** does not erase the character above the cursor. (Remember that **BS** is <Backspace>, so **M-BS** = <Meta-Backspace> = <Alt-Backspace> .)

Here are a few examples. Say that you have just typed the following text:

```
This book is not the best Emacs book ever written.
```

You decide you want to erase the word **not**. There are two ways to do it. First, you can move the cursor to the **space** between **is** and **not** and press **M-d** to erase the next word. Or you can move to the **t** at the beginning of **the** and press **M-BS** to erase the previous word.

Until you practice these commands, the exact positioning may seem a little odd. However, when you look carefully at the location of point, it all starts to make sense. In Section 8.2, I explained that **M-f** moves the cursor forward by one word, and **M-b** moves the cursor backward by one word. When you use **M-f**, it leaves you on the **space** between two words. When you use **M-b**, it leaves you on the first character of a word. Take a few moments and experiment. You will see that **M-f** and **M-d** work well together when you are moving forward, and **M-b** and **M-BS** work well together when you are moving backward.

The next two kill commands erase text within a sentence. The **M-k** command erases forward, from point to the end of the sentence. The **C-x BS** command erases backward, from point to the beginning of the sentence. When you use these commands, it is helpful to remember the commands that move the cursor one sentence at a time. **M-e** moves forward by one sentence; **M-a** moves backward by one sentence. These relationships are summarized in Figure 9-3.

	WORDS			SENTENCES	
	Backward	Forward		Backward	Forward
Move:	**M-b**	**M-f**	Move:	**M-a**	**M-e**
Kill:	**M-BS**	**M-d**	Kill:	**C-x BS**	**M-k**

FIGURE 9-3. Commands to move and kill by word or sentence.

The next kill command, **C-w**, is one you will use frequently, as it allows you to kill the entire region. (As I described in Section 8.7, the region consists of all the characters between mark and point.)

■ **Note** To erase the entire buffer, use **C-x h C-w. C-x h** creates a region consisting of the entire buffer (see Section 8.8). **C-w** kills the region.

C-w is particularly handy when you want to move a section of text. All you need to do is set mark and point to enclose the text, use **C-w** to kill the region, and then insert the text back into the buffer at a different location (or even into another buffer). I will discuss this idea — which is called "yanking" — in Section 9.4.

The final kill command is **M-z**. When you type this command, Emacs will prompt you to specify a single character. Emacs will kill all the text from point (your current position) to the next occurrence of that character. When you use this command, we say that you ZAP the characters, thus the name **M-z**.

Here is an example. You are editing an important document that was typed by someone who is not as smart as you. You come across a line that reads:

```
I can't imagine how anyone could prefer Emacs to vi.
```

Having read this, it is the work of a moment to move to the **space** following the **I** (at the beginning of the sentence) and type **M-z d**. This zaps all the characters from the **space** up to (and including) the **d** in **could**. The line now reads:

```
I prefer Emacs to vi.
```

Section 9.4: The Kill Ring and Yanking; Moving and Copying

As we have discussed, there are two types of commands that erase text. Delete commands erase single characters and whitespace (Section 9.2). Kill commands erase more than one character (Section 9.3). Whenever you use a kill command, Emacs saves the text in a KILL RING. (The name will make sense in a moment.) Because killed text is saved, you can copy it to wherever you want. Emacs has a number of commands that let you work with the kill ring and its contents. These commands are summarized in Figure 9-4.

Command	Description
C-y	Yank most recently killed text
C-u C-y	Same as **C-y**, cursor at beginning of new text
M-y	Replace yanked text with previously killed text
M-w	Copy region to kill ring, without erasing
M-C-w	Append next kill to newest kill ring entry
C-h v kill-ring	Display the actual values in the kill ring

FIGURE 9-4. Commands to yank text.

The kill ring is really a set of storage areas, each of which holds text that has been killed. Each storage area is called a KILL RING ENTRY. When you kill some text, it is stored as the most recent kill ring entry. To insert this text back into the buffer, you move the cursor to where you want to insert the text, and press **C-y**. When you copy such text into a buffer, we say that you YANK the text (hence the name **C-y**).

■ **What's in a Name?**

Kill, Yank

In the Emacs culture, "killing" refers to deleting text that is saved to the kill ring; "yanking" is copying this text back into the buffer. In other programs, these operations are generally referred to by the more refined names of cutting and pasting.

In other words, if you understand how to cut and paste, you know how to kill and yank.

When you use the **C-y** command to yank text into the buffer, Emacs sets mark at the beginning of the text and point at the end of the text. This means that the yanked text is now defined to be the region, in case you want to operate on it in some way. It also means that the cursor is just past the end of the text.

If you use the **C-u C-y** command instead, Emacs will yank the exact same text, but the locations of mark and point will be reversed. So, although the region will still enclose the newly inserted text, the cursor will be at the beginning. This is useful when you want to yank some text and then type something in front of it.

■ **Note** Before you yank text, think about where you want the cursor. If you want it at the beginning of the text, use **C-u C-y**. If you want the cursor at the end of the text, use **C-y**.

Let's talk for a moment about what happens as you kill more than one section of text. The first time you kill text, Emacs stores it in a kill ring entry. Later, when you kill more text, Emacs stores it in a different entry. The kill ring has a lot of storage space, so there is no need to throw away old text until you fill up all the key ring entries. And even then, you need discard only the very oldest material in order to make room for the new text.

By default, the kill ring has 60 entries. Thus, Emacs can store the last 60 sequences of killed text. (You can change this number if you want, but it is rarely necessary.) So what happens when all the kill ring entries are filled and you kill some more text? Emacs discards the oldest entry and uses it to hold the new text.

Some people like to visualize the kill ring as 60 entries organized into a circle. As Emacs needs to store text, it works its way around the circle, using one entry after another. Thus, we have the idea of a ring.

At all times, one of the kill ring entries in the circle is the current entry. This means it is possible to yank, not only the most recent kill ring entry, but the entry before that, and the entry before that, and so on (up to 60 entries). To go through the ring, one entry at a time, you use the **M-y** command. Here is how it works.

Just after you have used a **C-y** or **C-u C-y** command, take a look and see if the newly inserted text was what you wanted. If what you wanted is stored in a previous kill ring entry, press **M-y**. Emacs will erase the text and replace it with the previous entry. If this is what you wanted, fine. If not, press **M-y** again. You can continue pressing **M-y** until you run out of kill ring entries. In conceptual terms, you can think of the **M-y** command as working its way around the kill ring, showing you one entry after another.

■ **Note** If you are using **M-y** repeatedly to search for an old kill ring entry and you can't find what you want, you can always use **C-x u** to undo the last insertion and forget the whole thing.

Emacs maintains only one kill ring for all your buffers. This means you can kill some text in one buffer, and yank it into another buffer. Indeed, this is exactly how you move text from one buffer to another: kill some text, move to another buffer, then yank the last kill ring entry.

This procedure works well, but it does have one drawback: you have to erase something before you can move it. What if you merely want to copy something without destroying the original?

The solution is to define the region (using mark and point) so as to contain the text you want to copy, and then use the **M-w** command. This tells Emacs to copy the text to the kill ring without erasing anything. You can then yank the text wherever you want.

To copy something from one buffer to another, you set mark and point to enclose the text, press **M-w**, change to the other buffer, move to where you want to insert the text, and then yank it with **C-y**. Of course, this will also work with two locations in the same buffer.

When you use more than one kill command in a row, Emacs will automatically collect all the killed text into the same kill ring entry. This allows you to move from place to place, killing text and accumulating it into a single large block. You can then yank all of it with a single **C-y** or **C-u C-y** command.

As soon as you use a non-kill command, Emacs stops the accumulation. The next time you kill something, it will be put into a different kill ring entry. However, there is a way to tell Emacs to place killed text into the previous entry. All you need to do is press **M-C-w** before you kill the text. This tells Emacs to *append* the next thing you kill to the current kill ring entry.

Here is an example of how you might use **M-C-w**. Let's say you want to copy three paragraphs from different places in the buffer into a new file named **summary**. Go to each paragraph in turn and copy/append it into the current kill ring entry. Then open a new buffer and yank the three paragraphs into that buffer. You can now save the buffer as a new file. Here is what the whole procedure looks like. Read it closely and slowly to make sure it makes sense to you. Then take a moment to try it for yourself.

Copy the first paragraph to the current kill ring entry:

1. Move the cursor to the beginning of the first paragraph.
2. **C-SPC**: Set mark (Section 8.7).
3. **M-}**: Move the cursor to end of the paragraph (Section 8.2).
4. **M-w**: Copy region to the current kill ring entry.
5. Move the cursor to the beginning of the second paragraph.

Append the second paragraph:

6. **C-SPC**: Set mark.
7. **M-}**: Move the cursor to end of the paragraph.
8. **M-C-w M-w**: Append region to the current kill ring entry.

Append the third paragraph:

9. Move the cursor to the beginning of the third paragraph.
10. **C-SPC**: Set mark.
11. **M-}**: Move the cursor to the end of the paragraph.
12. **M-C-w M-w**: Append region to the current kill ring entry.

Copy the current kill ring entry to a new buffer.

13. **C-b summary** <Enter>: Create a new buffer named **summary**.
14. **C-y**: Yank contents of current ring entry into buffer.
15. **C-x C-w summary** <Enter>: Save to file named **summary**.

The final kill-oriented command is **C-h v kill-ring**. This tells Emacs to display the actual contents of the kill ring. You will rarely have to use this command but, from time to time, it is interesting to look at the entire kill ring. This command makes use of the built-in Help facility (described in Section 12.3). In technical terms, Emacs stores the kill ring as a "variable" called **kill-ring** (we can ignore the details). The **C-h v** command simply tells Emacs to display the value of that variable.

Section 9.5: Correcting Common Typing Mistakes

Emacs has a number of commands that have been specifically designed to make it easy to correct typing mistakes. These commands are shown in Figure 9-5. As you can see, there are three types of commands: erasing, case changing and transposing. Notice that most of the commands act upon characters you have just typed (that is, characters to the left of the cursor). In Section 9.6, I will discuss how to correct spelling mistakes.

Command	Description
BS	Delete one character to the left of cursor
DEL	Delete one character at the position of cursor
C-d	Same as **DEL**
M-BS	Kill the previous word
C-x BS	Kill backward to beginning of sentence
M-- M-l	Change previous word to lowercase
M-- M-u	Change previous word to uppercase
M-- M-c	Change previous word to lowercase, initial cap
M-l	Change following word to lowercase
M-u	Change following word to uppercase
M-c	Change following word to lowercase, initial cap
C-t	Transpose two adjacent characters
M-t	Transpose two adjacent words
C-x C-t	Transpose two consecutive lines

FIGURE 9-5. Commands for correcting common typing mistakes.

■ **Note** **BS** is <Backspace>; M-- is <Meta-hyphen>.

We have already discussed the delete and kill commands. **BS** erases the character you have just typed; **M-BS** erases the word you have just typed; and **C-x BS** erases to the beginning of the sentence.

■ **Note**

Your typing habits will influence which commands you prefer to use to make corrections when you notice a mistake.

If you are a slow typist who looks at the keys as you type, you will probably find it easier to move the cursor to the left and right, and make your corrections one character at a time. However, if you can type quickly without looking at the keys, it is a lot easier to press **M-BS** several times in a row and simply retype the last few words.

One of the most common typing mistakes is to mix up your lower- and uppercase letters. Emacs has three commands to help you. To change all the letters in the previous word to lowercase, use **M-- M-l**. To change the letters to uppercase, use **M-- M-u**. To change the letters to lowercase with the first letter capitalized, use **M-- M-c**. (**M--** means <Meta-hyphen>.)

Although these three commands act on a single word, you can use a prefix argument (explained in Section 8.4) to change more than one word at a time. For example, say that you have just typed 10 words in lowercase, and you decide to make each word lowercase with an initial capital letter (as in a title). All you have to do is type **ESC 10 M-- M-c**.

The remaining commands transpose (switch around) characters, words or lines. **C-t** transposes two adjacent characters. In general, **C-t** switches the character at the cursor with the character immediately to its left. (In other words, it switches the two characters on either side of point.) Thus, if you have typed the word **Halrey** you can correct it by moving to the **r** and pressing **C-t**.

However, there is one special case. If the cursor is at the end of a line (just past the final character in the line), pressing **C-t** will transpose the last two characters. This means that if you are typing on a new line, and you press two keys in the wrong order, you can make an immediate correction by typing **C-t**. (Try it and it will make sense.)

To be more technical, the **C-t** command normally transposes the two characters on either side of point. However, at the end of a line, **C-t** switches the two characters before point. (Remember, point is between the character at the cursor, and the character on its left. For example, if the cursor is on the **r** in **Harley**, point is between the **a** and the **r**.)

The **M-t** command is similar except that it transposes two words: the word before point and the word after point. Here is an example. Say that you have just typed the line:

```
I should buy a copy of "Harley Hahn's Guide to Unix Linux and".
```

You decide to switch the last two words.

To do so, move the cursor so that point is between the words **Linux** and **and**. (The cursor should be on the space after "Linux" or on the following "a".) Now, press **M-t**. When Emacs transposes words, it does not change the punctuation, so in our example the quotation mark and the period will not be moved. You will see:

```
I should buy a copy of "Harley Hahn's Guide to Unix and Linux".
```

(Which, of course, is good advice.)

The final command is **C-x C-t**. This command switches two consecutive lines: the current line and the line above it. For example, say that the buffer contains the lines:

```
11111
22222
33333
44444
55555
```

You move the cursor to the fourth line (**44444**) and press **C-x C-t**. The buffer will change as follows:

```
11111
22222
44444
33333
55555
```

The only exception is when you are on the first line of the buffer. In this position, there is no line above the current line so, as you might expect, **C-t** switches the top two lines. In our example, if you were to move to the top line (**11111**) and press **C-x C-t**, you would see:

```
22222
11111
44444
33333
55555
```

Note If you place the cursor below a specific line and then press **C-x C-t** repeatedly, you will move that line down the buffer, one line at a time. Take a moment to try it.

Section 9.6: Correcting Spelling Mistakes

Emacs has a variety of commands to help you correct spelling. These commands are shown in Figure 9-6. Before you read this section, take a moment to create a new buffer and type some text, so you can test the various commands as I explain them.

Command	Description
M-$	Spell check: word at point, or region
M-x ispell-buffer	Spell check: buffer (only)
M-x ispell-region	Spell check: region (only)
M-x ispell	Spell check: buffer, or region
M-TAB	Complete word before point, based on dictionary
ESC TAB	Same as **M-TAB**
M-x flyspell-mode	Highlight spelling mistakes in regular text
M-x flyspell-prog-mode	Highlight spelling mistakes in programs

FIGURE 9-6. Commands for correcting spelling mistakes.

The simplest and most important command is **M-$** (on most keyboards, you would press <Alt-Shift-4>). This command has two possible actions, depending on the situation.

If the region is not set — which will be the case most of the time — **M-$** checks the spelling of the word at point. That is, at the position of the cursor. If the region is set, **M-$** will check the spelling of all the words in the region.

Think about that for a moment. If you want to check the spelling of a single word, just move to it and press **M-$**. If you want to check the spelling of all the words in a section of text, or a paragraph, or the entire buffer, all you need to do is set region appropriately and press **M-$**.

For example, to spell check a paragraph, move point (the cursor) to that paragraph and use:

M-h M-$

To spell check the entire buffer, use:

C-x h M-$

(To learn how to set the region, see Section 8.8.)

When Emacs finds a word that is misspelled, it will create a new buffer named ***Choices*** to show you possible words from the dictionary. Each word will have a single number or other character in front of it. Simply press the number (or character) of the word you want to use. (It's easier than it sounds. Try it once, and you will see how it works.) If you don't see what you want, you can stop the spell check by pressing **C-g**.

For completeness, I will mention that there are three more specific spell check commands:

- **M-x ispell-buffer**: Spell checks the entire buffer (only).
- **M-x ispell-region**: Spell checks the region (only).
- **M-x ispell**: If the region is set, spell checks the region. Otherwise, spell checks the entire buffer.

As you are typing, Emacs can help you complete a partially typed word. Just type **M-TAB**. Emacs will then check the word to the left of point and show you a list of possible choices. Just pick the one you want and keep typing.

Note

To use **M-TAB**, you need to press <Alt-Tab>. However, if you are working in a GUI, it is likely that <Alt-Tab> will have a special meaning because, in many GUIs, this key combination is used for task switching.

If this is the case for you, you can use **ESC TAB** instead. (Using **ESC** as a substitute for the Meta key is discussed in Section 4.4.)

The final spell check tool I want to discuss is called Flyspell mode. Flyspell mode is what we call a "minor mode", which means that it is a setting you can turn on or off to provide a service. (We will discuss minor modes in Section 11.4.)

When Flyspell mode is on, Emacs will highlight all the misspelled words. This is a big help, not only as you are typing, but to let you check for spelling mistakes within the entire window, just by glancing at your screen. If you see a mistake, all you need to do is move to it and press **M-$**.

To turn on Flyspell mode, use:

M-x flyspell-mode

To turn it off, use the same command again: it acts like a toggle.

When you are editing computer programs, you can use Flyspell-Prog mode. This tells Emacs to highlight spelling mistakes, but only within comments and character strings.

The command to turn Flyspell-Prog mode on or off is:

M-x flyspell-prog-mode

Section 9.7: Filling and Formatting Text

As you type regular text (compared to say, typing a computer program), you have two special needs. First, when you reach the end of the line, you will want Emacs to start a new line for you automatically. Otherwise, you will have to keep track of the cursor position and press <Enter> each time you near the right margin.

Second, as you edit the text, you will shorten some lines and lengthen others, and you will want to be able to reformat the text to maintain as smooth a right margin as possible.

Emacs has commands to perform both of these tasks. These commands are summarized in Figure 9-7.

Command	Description
M-x auto-fill-mode	Turn on/off Auto Fill mode
M-q	Fill a paragraph
ESC 1 M-q	Fill+justify a paragraph
M-x fill-region	Fill each paragraph in the region
ESC 1 M-x fill-region	Fill+justify each paragraph in region
M-x fill-region-as-paragraph	Fill region as one long paragraph
ESC 1 M-x fill-region-as-paragraph	Fill+justify region as one paragraph
C-x f	Set the fill column value
C-h v fill-column	Display current fill column value

FIGURE 9-7. Commands to fill text.

To tell Emacs to break long lines automatically as you type, you turn on "Auto Fill mode". (We will discuss modes in Sections 11.1 through 11.7.) When Auto Fill mode is turned on, Emacs will break lines for you as you type. When Auto Fill mode is off, Emacs will not break lines.

How do you know if Auto Fill mode is on or off? When it is on, Emacs will display the word **Fill** on the mode line near the bottom of your window (see Section 6.4). You can see this in Figure 9-8.

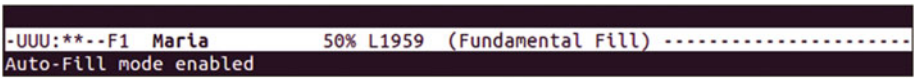

FIGURE 9-8. Mode line showing that Auto Fill mode is enabled.

When you have turned on Auto Fill mode using **M-x auto-fill-mode** *you will see the word* **Fill** *on the mode line*

When Auto Fill mode is off, you can turn it on by using the command **M-x auto-fill-mode**. When Auto Fill mode is on, you can turn it off by using the same command. Thus, **M-x auto-fill-mode** acts as an on/off toggle switch. If you are not sure of the current status of Auto Fill mode, just look at the mode line for the word **Fill**.

If you use Emacs only for typing regular text, you will probably want Auto Fill mode to be on all the time by default. I explain how to do this in Section 11.9, when we talk about customizing Emacs.

■ **Note** You do not have to type the full command **M-x auto-fill-mode**. As with all such commands, you can use completion (discussed in Section 6.7) to ask Emacs to do some of the typing for you. In this case, all you need to type is:

M-x au SPC -f RET

One of the limitations of Auto Fill mode is that it will not automatically reformat a paragraph when you make changes. For example, when you delete and insert words, you change the length of the lines and the right margin can become ragged.

To reformat a paragraph, move the cursor to be within that paragraph (or on a blank line before the paragraph), and use the command **M-q**. Emacs will reformat by removing all the line breaks and inserting new ones as necessary. When this happens, we say that Emacs FILLS the paragraph.

There are two other fill commands you will find useful. When you want to fill part of a paragraph or more than one paragraph, you can define the region (by setting mark and point) to contain the text you want to fill. You can then fill the entire region by using **M-x fill-region.** (With completion, you only need to type **M-x fil SPC r RET**.)

A variation of this command is **M-x fill-region-as-paragraph**. This command fills an entire region as a single large paragraph. Again, with completion, you can save keystrokes by typing:

M-x fill SPC r SPC SPC RET

The fill commands can also right justify at the same time as they format. That is, they can insert spaces within the text in order to make the right margin line up. To fill and right justify at the same time, just use a prefix argument before a fill command (see Section 8.4). Any numeric value will do, so you might as well use **1**.

For example, to fill and right justify a paragraph, use the command **ESC 1 M-q**. You can also use a similar prefix argument with the other fill commands to right justify and fill the region.

When Emacs fills a paragraph, it ensures that no line is longer than a specific maximum width. By default, this width is 70 characters, but you can change it if necessary.

The value of the fill column width is kept in a variable (storage location) named **fill-column**. You can display the current value of this variable by using the command **C-h v fill-column**. (The **C-h v** command is part of the built-in Help facility. This particular command displays the value of a variable along with a short description.)

There are two ways to change the value of **`fill-column`**, both of which use the **`C-x f`** command. First, you can use this command with a prefix argument showing how many characters wide you want your lines to be. For example, to set **`fill-column`** to a value of 55, you can use **`ESC 55 C-x f`**. From now on, all lines you fill will be formatted to be no longer than 55 characters long.

The second way to change the value of **`fill-column`** is to move the cursor to a line that is exactly the width that you want, and then press **`C-x f`**. Emacs will use the width of that particular line as the new value for **`fill-column`**.

CHAPTER 10

Searching

Section 10.1: Introducing the Emacs Search Commands

Emacs gives you several ways to search for a pattern within the buffer. Figure 10-1 shows a summary of the different search commands. At first, it looks overwhelming, but don't worry. In almost all cases, it boils down to four simple ideas.

1. To search forward, press **C-s** and type what you want to find.
2. To search backward, press **C-r** and type what you want to find.

To remember the names, think of **s** for "search" and **r** for "reverse". As you are searching, the characters you type are called the SEARCH STRING.

3. If Emacs finds what you specified, you can search for another occurrence by pressing **C-s** (or **C-r**) as many times as you need.
4. When you find exactly what you want, press **RET** (the Enter key) to stop the search.

Most of the time, that's all you need to remember. However, this being Emacs, there are lots and lots of optional details (which we will get to in a moment).

Command	Description
C-s	Forward: Incremental search
C-s RET	Forward: Non-incremental search
M-s w	Forward: Incremental *word* search
M-C-s	Forward: Incremental *regexp* search
M-C-s RET	Forward: Non-incremental *regexp* search

Command	Description
C-r	Backward: Incremental search
C-r RET	Backward: Non-incremental search
M-s w C-r	Backward: Incremental *word* search
M-C-r	Backward: Incremental *regexp* search
M-C-r RET	Backward: Non-incremental *regexp* search

FIGURE 10-1. Search commands.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_10

■ **Note** The **C-s** key sequence is bound to the command **isearch-forward. C-r** is bound to **isearch-backward. isearch** stands for "incremental search".

Section 10.2: Incremental Searching

The basic type of Emacs search is called an INCREMENTAL SEARCH. That means Emacs starts searching as soon as you type a single character. With each character you type, Emacs refines its search. Here is an example.

Let's say that you want to search for the pattern **harley**. You start by pressing **C-s**. Emacs is now waiting for you to type something. You type **h**. The search string is **h**, so Emacs jumps to the first place in the buffer that has the letter "h".

Next, you type **a**. The search string is now **ha**, so Emacs jumps to the first place in the buffer that has the letters "ha".

You then type **r**. The search string is now **har**, so Emacs jumps to the first place that has "har".

In other words, Emacs starts searching the moment you begin to type. Each time you type another character, the search string becomes more and more specific. The advantage is that, most of the time, you do not have to type the full pattern to find what you want. For instance, it is possible that you would find the pattern **harley** after typing only the first few letters.

The best way to see how this all works is to try it for yourself.

1. Start Emacs.

When you are practicing, you can start Emacs by using the **-Q** option to suppress the splash screen (see Section 5.1). Either of the following commands (see Section 5.2) will do:

```
emacs -Q
emacs -Q -nw
```

Emacs will start by creating one large window containing a buffer named ***scratch***. There will be a few lines at the top with some useful information. If you want to erase these lines, use **C-x h C-w** (see Section 9.3), although you don't have to.

2. Display something interesting to search.

Start the Help facility by pressing **C-h b**. This will display a large list of all the key bindings. Press **C-x 0** to delete the window with the ***scratch*** buffer. You will be left in the ***Help*** buffer.

3. Practice using **C-s** and **C-r**.

You can now use **C-s** (search forward) and **C-r** (search backward) as much as you want. If you are not sure what to search for, try searching for the word "command".

I suggest that you set this up right now, so as you read the following sections, you can try out the various commands.

Section 10.3: Keys to Use While Searching

While you are in the middle of a search, there are a number of keys that have special meanings. These are shown in Figure 10-2.

Key	Description
BS	Erase last character you typed
RET	Terminate the search
C-s	Search forward for same pattern
C-r	Search backward for same pattern
C-g	While search is in progress: Stop current search
C-g	While waiting for input: Abort entire command
C-w	Copy the word after point to search string
C-y	Copy current kill ring entry to search string
M-y	Copy previous kill ring entry to search string

FIGURE 10-2. Keys to use during a search.

While you are typing the search pattern, you can make a correction by pressing **BS** (<Backspace>) to erase the last character you typed. When you do, Emacs will back up to the place that matches those characters that remain.

Here is an example. Say that you are searching for the pattern **harley**. After you have typed the first three letters, **har**, Emacs jumps to the word **share** (because this happens to be the first word that contains these three characters). To continue, you want to type the letter **l**, but, by accident, you press the **p** key. Thus, Emacs thinks you are searching for **harp** and it jumps to the word **harpoon**. To make the correction, you press **BS**. This erases the letter **p**, whereupon Emacs jumps back to the word **share**. You now type **ley** and Emacs jumps to **harley**. (Don't worry if this is a bit difficult to follow. It will all make perfect sense when you see it.)

At any time, you can tell Emacs to find the next occurrence of the search string by pressing **C-s** once again. To go backward, press **C-r**. For example, say that you have typed **harley** and Emacs has dutifully found the first occurrence of that word. However, that is not the one you want. Simply press **C-s** and Emacs will find the next one.

If you press **C-s** and Emacs cannot find any occurrences of the search string from your current position to the end of the buffer, you will see the message:

```
Failing I-search
```

This means the incremental search has failed. That is, you are looking at the last place in the buffer that matches the search string. However, if you press **C-s** one more time, Emacs will wrap around and start searching at the beginning of the buffer.

In other words, you can press **C-s** repeatedly to search for a pattern throughout the buffer. When you get to the end (and you see the message), you can press **C-s** and start again from the beginning. Similarly, when you are searching backward with **C-r**, you can press **C-r** twice when you get to the top and wrap around to the bottom.

To terminate your search, you have two options. If Emacs has found what you want, simply press **RET** (<Enter>). The search command will stop and you will be left at the current position in the buffer.

If, however, you can't find what you want and you decide the whole thing was a bad idea, press **C-g**. This will stop the search command, but will leave you where you started, just as if nothing had happened.

There are three other keys that are handy to use during a search. If you press **C-w**, Emacs appends the word immediately after point (the position of the cursor) to the search string. Emacs then advances point to the end of that word. You can press **C-w** more than once, to pick up one word after another.

What does this mean? It means that you can search for a word in the buffer without having to actually type the word for yourself. Here is an example. You are reading an essay that contains the following quote from the writer Isaac Asimov's final autobiography:

```
Perhaps writers are so self-absorbed as a
necessary part of their profession.
```

The cursor is on the space before the word **writers**. (That is, point is between the space and the **w**.) You decide to search for further occurrences of the word **writers**.

You press **C-s**. Emacs is now waiting for you to type a search string. Instead of typing the actual word, you press **C-w** and Emacs copies the word **writers** for you into the minibuffer, just as if you had typed the letters yourself. Emacs then moves the cursor to the next word (to the space before **are**). You can either press **C-w** again to pick up the next word, or press **C-s** to begin the search.

You can copy as many words as you want by pressing **C-w**. And you can do so not only when you start the search, but at any point along the way. Thus, whenever Emacs is paused waiting for you to type something to add to the search pattern, you can press **C-w** to copy the current word. If you make a mistake, you can correct it by pressing **BS** (<Backspace>). When you do, Emacs will erase an entire word, not just a single character. (In other words, when you are searching and you press **BS**, Emacs knows whether it should erase a whole word or simply one character. Pretty cool, eh?)

Aside from **C-w**, there are two other commands that will append text to the search string while you are searching. **C-y** ("yank") appends the current kill ring entry. If after using **C-y**, you type **M-y**, Emacs replaces the yanked text with the previous kill ring entry. Here is an example:

In the course of your work, you erase the word **Ignormus** (it's actually a name), and then you erase the word **heffalump**. Thus, the current key ring entry is **heffalump**, and the previous key ring entry is **Ignormus**.

You then switch to another buffer, where you are reading another quote from Isaac Asimov's autobiography:

```
I have always thought of myself as a
remarkable fellow, even from childhood,
and I have never wavered in that opinion.
```

(This, of course, is true of all good writers.)

The cursor is at the space before **always** in the first line, so point is between the space and the **a**. You press **C-s** to start a search, and then press **C-w** to copy the next word to the search string. The search string is:

```
always
```

You press <Space> to put in a single space. You then press **C-y** to append the current key ring entry to the end of the search sting. Since the current key ring entry is **heffalump**, the search string is now:

```
always heffalump
```

You now press **M-y**. This replaces **heffalump** with the previous key ring entry **Ignormus**. The search string is now:

```
always ignormus
```

Note

If you are not familiar with a heffalump or the Ignormus, take a moment to look them up online (separately).

Be sure to spell Ignormus correctly (it is not "Ignoramus").

Section 10.4: Upper- and Lowercase Searching

One issue we have not yet discussed is what Emacs does about upper- and lowercase letters while it is searching. The general rule is that Emacs ignores all distinctions between upper- and lowercase as long as you type only lowercase letters in the search pattern. However, once you type even a single uppercase letter, Emacs will search for an exact match.

Here is an example. If you tell Emacs to search for **harley**, it will find any occurrence of these letters, whether they are upper- or lowercase or mixed. For instance, Emacs would find **harley**, **Harley**, **HARLEY**, **harLEY**, and so on. However, if you tell Emacs to search for **Harley**, it will find only this exact word.

When a program does not distinguish between upper- and lowercase, we say that it is CASE INSENSITIVE. When the distinction is made, we say the program is CASE SENSITIVE.

So let's say that you are searching for the phrase **send money to Harley**. You press **C-s** and then start to type the pattern. As you do, Emacs begins the search. So far, you have typed **send money to**. Up to this point, you have typed only lowercase letters, so Emacs is performing a case insensitive search. However, as soon as you type the **H**, Emacs switches to a case sensitive search.

At this point, you might wonder what would happen if you were to press **BS** <Backspace> and erase the only uppercase letter? Emacs would recognize what you did and switch from being case sensitive back to case insensitive.

Please realize that changing from a case insensitive search to case sensitive search requires you to actually type an uppercase letter. The change won't happen if you copy an uppercase letter to the search string from the key ring.

To see what I mean, take a look at the very last example in Section 10.2. The search string was the word **always** followed by a space:

```
always
```

We pressed **M-y** to copy an entry from the key ring to the search string. The key ring entry was the word **Ignormus**. However, the new search string was:

```
always ignormus
```

Why was the word **ignormus** in lower case? Because we had not typed an uppercase letter into the search string manually, Emacs was using a case insensitive search, so it changed **Ignormus** to **ignormus**.

If you want to test this for yourself, type **Ignormus** (with an uppercase **I**), and then press **M-BS** to kill the entire word (see Section 9.3). The current kill ring entry is now **Ignormus**.

Now press **C-s** to start a new search and type **Always**, followed by a space. The search string is:

```
Always
```

Because you typed an uppercase letter (**A**), Emacs is doing a case sensitive search. Press **C-y** to append the current kill ring entry to the search string. The new search string is:

```
Always Ignormus
```

Because you are using a case sensitive search, Emacs preserved the uppercase **I** when it copied the kill ring entry to the search string.

Section 10.5: Non-Incremental Searching

If you look back at Figure 10-1, you will see several types of search commands, and a forward and backward variation for each type. So far, we have discussed the incremental search commands (**C-s** and **C-r**). These commands begin searching as soon as you type a single character. As you type more characters, the search becomes more specific. This is called an incremental search (Section 10.2).

It may be that you want Emacs to wait until you have typed the entire pattern before it starts to search. To do so, you can use the NON-INCREMENTAL SEARCH commands listed it Figure 10-3. This is convenient if you have trouble typing and you make lots of mistakes. If so, it is a bother to have Emacs jump all over the place like a one-legged tap dancer as you fix your mistakes, one character at a time.

Command	Description
C-s RET	Forward: non-incremental search
C-r RET	Backward: non-incremental search

FIGURE 10-3. Non-incremental search commands.

To start a non-incremental search, simply press **RET** (<Enter>) before you start typing the search pattern. Thus, to perform a forward non-incremental search, type **C-s RET** and then type the pattern. When you are finished typing, press **RET** once again to start the search. To abort the search, press **C-g**. Similarly, you can request a non-incremental backward search by typing **C-r RET**. For reference, these commands are shown in Figure 10-3.

Section 10.6: Word Searching

A WORD SEARCH tells Emacs to search only for complete words. To use a forward word search, type **M-s w**. To use a backward word search, type **M-s w C-r**. You can now type the words you want to search for. When you are ready, press **RET** to begin searching. (The word search is actually a variation of the incremental search.)

Command	Description
M-s w	Forward: Incremental word search
M-s w C-r	Backward: Incremental word search

FIGURE 10-4. Search commands.

The great thing about a word search is that it ignores all punctuation, spaces, tabs and end of lines. Thus, you can look for a series of words that, for example, span more than one line. Here is an example.

You are using Emacs to read the text of Isaac Asimov's final autobiography. Within the text, you find the following passage:

```
My turn will come too, eventually, but I have
had a good life and I have accomplished all I
wanted to, and more than I had a right to expect.

So I am ready.

But not too ready.
```

Some time later, you want to find this passage again, but all you can remember are the last two lines. So you press **M-s w** to start a word search. You then type **ready but not** and press **RET** again. Emacs finds the text, even though these particular words are separated by punctuation (a period) and are broken over two lines.

Section 10.7: Searching for Regular Expressions

A REGULAR EXPRESSION or REGEXP is a compact way of specifying a general pattern of characters. Emacs has special commands that allow you to search for a regular expression. These commands are shown in Figure 10-5. As you can see, the term "regular expression" is sometimes abbreviated to "regexp". This abbreviation is worth memorizing, as you will see it a lot.

Command	Description
M-C-s	Forward: Incremental search for regexp
M-C-s RET	Forward: Non-incremental search for regexp

Command	Description
M-C-r	Backward: Incremental search for regexp
M-C-r RET	Backward: Non-incremental search for regexp

FIGURE 10-5. Search commands for regular expressions.

■ **Note** If you have a problem using the key sequence **M-C-s**, you can use **ESC C-s** instead.

Notice that these commands use an **M-C** combination. This means that you must hold down both the Meta and Ctrl keys while you press the other key (either **s** or **r**). For example, to search forward for a regular expression you use **M-C-s**. To do so, you press <Meta-Ctrl-S>, which is actually <Alt-Ctrl-S>. Alternatively, instead of holding down the Meta key, you can press **ESC** (see Section 4.3). So, to search forward for a regular expression, you use **M-C-s** by typing either of the following:

<Alt-Ctrl-S>
<Esc> <Ctrl-S>

The same goes for **M-C-r**. To search backward for a regular expression, you can type either of the following:

<Alt-Ctrl-R>
<Esc> <Ctrl-R>

■ **Note**

Stop for a moment and think about what you have just read. Notice that, when you read **M-C-s**, you just know, without thinking that you must press <Alt-Ctrl-S> or type <Esc> <Ctrl-R>.

If you are an experienced Unix user and you've gotten this far, it's too late to go back to **vi**. Your brain has already changed permanently. So stick it out and read the rest of the book.

If you are using Emacs within a terminal window, you may run into trouble with **M-C-s**, because the key combination <Alt-Ctrl-S> has a special meaning with certain GUIs. If you have a problem using **M-C-s**, you can avoid it by using **ESC C-s** instead. (Isn't it nice that Emacs is so flexible?) Alternatively, if you want to fix the problem permanently, see the instructions in Section 10.9.

Section 10.8: Regular Expressions

Regular expressions allow you to expand your search capabilities enormously. For example, using the ordinary search commands, you can tell Emacs to look for an occurrence of the pattern **`Harley`**. But with regular expressions you could, for example, tell Emacs to search for the pattern **`Harley`** at the beginning of a line, or a line that consists only of the word **`Harley`**, or any word that starts with **`Har`** and ends with **`y`**.

Regular expressions are widely used — far beyond the world of Emacs — and understanding how they work is a *very* important skill, especially if you are a programmer. In this section, I will give you a summary of the special characters that Emacs uses for regular expressions, along with a few examples. However, if this topic is new to you, I encourage you to take time, away from Emacs, to learn all about regular expressions and how to use them well. (I cover this topic in detail in my book *Harley Hahn's Guide to Unix and Linux*.[1])

Figure 10-6 summarizes the basic characters you can use to create a regular expression with Emacs. If you already understand regular expressions, these characters will make sense to you, although you may see a few new ones.

Character	Description
Char	Any regular character matches itself
`.`	Match any single character except **`RET`**
`*`	Match zero or more of the preceding characters
`+`	Match one or more of the preceding characters
`?`	Match exactly zero or one of the preceding characters
`^`	Match the beginning of a line
`$`	Match the end of a line
`\<`	Match the beginning of a word
`\>`	Match the end of a word
`\b`	Match the beginning or end of a word
`\B`	Match anywhere not at the beginning or end of a word
`\``	Match the beginning of the buffer
`\'`	Match the end of the buffer
`\`*char*	Quotes a special character
`[ ]`	Match one of the enclosed characters
`[^ ]`	Match any character that is not enclosed

FIGURE 10-6. Characters to use with regular expressions.

[1] *Harley Hahn's Guide to Unix and Linux*, McGraw-Hill Higher Education, 2008. The ISBN is 0073133612.

Here is an example of a search involving a regular expression. To search for the characters **Harley** at the beginning of a line, press **M-C-s**, then type **^Harley**. If you prefer, you can make the search non-incremental by pressing **RET** (the Enter key) before you type the regular expression. To search backward, you can use either **M-C-r** or **M-C-r RET**.

Here are a few more examples. To search for a line that consists entirely of the word **Harley**, use:

```
^Harley$
```

To search for a word that begins with **Har**, use:

```
\<Har
```

To search for a pattern that starts with **h** or **H**, and is followed by at least one **a**, use:

```
[hH]a+
```

You can use a range of characters within square brackets, for example:

[0-9] all numerals
[a-z] lowercase letters
[A-Z] uppercase letters
[a-zA-Z] all letters (uppercase and lowercase)

For instance, to search for a complete word that consists of a single letter, either upper- or lowercase, possibly followed by a single digit, use:

```
\<[a-zA-Z][0-9]?\>
```

This regular expression will match words like:

```
a0  a  A0  A  B5  z8  Z
```

If you want to search for a character that has a special meaning, you must put a **** (backslash) in front of it. For example, to search for a dollar sign followed by one or more digits, use:

```
\$[0-9]+
```

Note

When you specify all lowercase letters, Emacs does a case insensitive search. However, if you use at least one uppercase letter, the search becomes case sensitive. Examples:

The regular expression **[har]** searches for **h**, **H**, **a**, **A**, **r** or **R**.

The regular expression **[Har]** searches only for **H**, **a** or **r**.

Section 10.9: Fixing Emacs Key Conflicts

When you run Emacs within a terminal window under your GUI, you may, from time to time, run into trouble with an Emacs key combination if it has a special meaning to that particular GUI.

We saw such an example in Section 10.7. Within Unity, the default GUI used with Ubuntu Linux (see Section 2.5), <Alt-Ctrl-S> has a special meaning. However, this is the combination you need to type **M-C-s**, the Emacs command to search for a regular expression. One alternative is to use **ESC** instead of the Meta key. For example, instead of using **M-C-s**, you can type **ESC C-s**. However, you may want to fix the problem permanently.

In this section, I will show you how to do so with Unity**.** If you encounter a similar problem with a different Emacs key sequence, the instructions you are about to read may give you an idea how to solve it.

When you run Emacs within a terminal window under Unity, you may find that, when you press <Alt-Ctrl-S>, instead of starting a regex search, it causes your terminal window to "roll up" and disappear. This is called a "shaded state". (Think of an actual window shade that, when you pull a cord, rolls up out of the way.) To fix the problem, all you have to do is tell Unity to change the meaning of this particular key combination. Here is how to do it.

1. Open the Dash (see Section 2.6) and search for "System Settings".
2. Open "System Settings".
3. Under Hardware, open Keyboard.
4. Click the Shortcuts tab.
5. In the left-hand column, look for the category that is likely to contain the key you want to change. In our example, click Windows, because <Alt-Ctrl-S> is changing your terminal window.
6. In the right-hand column, click the offending key function. In our example, it is "Toggle Shaded State", because that is what is matched to <Alt-Ctrl-S>.

7. When you click on the key function, the key name will change to:

New accelerator...

The program is asking you to specify which key combination you want to use for an accelerator key (shortcut key) for the function "Toggle shaded state".

8. You have two choices. To disable the shortcut key completely, press <Backspace>. To specify a different key combination, simply press it.
9. Close the Keyboard window.

Now, when you return to Emacs, **M-C-s** will work properly and start a regular expression search.

Section 10.10: Searching and Replacing

There are several commands you can use to search for a particular pattern and then replace it once it is found. For example, say you have written a long memo to your boss in which you have used the word "jerk" several times. As you read the document a second time, it occurs to you that an overly sensitive person might take offense, so you decide to change every occurrence of **jerk** to **goofball**.

To do so, you can use one of the search and replace commands. You tell Emacs what to search for and what to use as a replacement. Emacs will start from point (the current position within the buffer) and search forward for occurrences of the search pattern. Each time it finds your search string, Emacs can make the replacement.

Note A search and replacement operation starts from point and continues to the end of the buffer. Thus, if you want to process the entire buffer, you must jump to the beginning before you start (use the **M-<** command).

With some of the search and replace commands, Emacs will ask you for your approval, each time, before it makes a change. With other commands, Emacs makes all the changes automatically.

Figure 10-7 summarizes the Emacs search and replace commands. Notice that, of the four commands, two have a key sequence (**M-%** and **M-C-%**). The other two commands must be executed explicitly using **M-x**. However, as you will see in a moment, you can use completion (see Section 6.7), so typing these long names is actually pretty easy.

Command	Description
M-%	Query: Search and replace
M-C-%	Query: Search and replace (regexp)
M-x replace-string	No query: Search and replace
M-x replace-regexp	No query: Search and replace (regexp)

FIGURE 10-7. Search and replace commands.

The basic search and replace command is **M-%**. On most keyboards, the **%** (percent) character is <Shift-5>, so to use **M-%** you type <Alt-Shift-5>.

After you type **M-%**, Emacs will prompt you to type the search string, that is, the pattern for which you are searching. You will see the following message in the minibuffer:

```
Query replace:
```

Don't be confused. Emacs is not asking for the replacement string. It is simply reminding you that you are using the "Query replace" command. Type your search string and press **RET**. In our example, you would type **jerk** and press <Enter>. Emacs will then prompt you for the replacement characters. You will see:

```
Query replace jerk with:
```

Type the replacement and press **RET**. In our example, you would type **goofball** and press <Enter>.

■ **Note**

Emacs is designed to save you typing effort whenever possible. Here is a good example.

The first time in a work session that you use search and replace, Emacs asks you to type your search string and your replace string. The next time you use search and replace, Emacs will suggest the same two strings. For example, if you have already used **jerk** and **goofball**, the next time you type **M-%**, Emacs will display:

```
Query replace (default jerk -> goofball):
```

If you want to repeat the same operation, press **RET**. If you want to make changes, simply type a new search string and press **RET**.

Once you specify your search and replace strings, Emacs jumps to the first occurrence of the search string and asks you what to do. In our example, you would see:

```
Query replacing jerk with goofball: (? for help)
```

At this point, you have several choices, which are described in Figure 10-8. Notice that you can press **?** to display a help summary.

Command	Description
?	Display help summary
y	(yes) Replace
n	(no) Do not replace
q	Quit immediately
SPC	Same as **y**
BS	Same as **n**
RET	Same as **q**
!	Replace all remaining matches, no questions
.	(period) Replace current match and then quit
,	(comma) Replace but stay at current position
^	(circumflex) Move back to previous match
C-l	Clear screen, redisplay, and ask again
C-r	Start recursive editing (use **M-C-c** to return)
C-w	Delete matching pattern, start recursive edit

FIGURE 10-8. Responses during a search and replace command.

Most of the time, you will need only four of these responses:

- **SPC** (<Space>) to make a replacement and continue.
- **BS** (<Backspace>) to skip a replacement and continue.
- **!** (exclamation mark) to make all the rest of the replacements automatically with no more questions.
- **RET** (<Enter>) to quit immediately

The rest of the commands (except for **C-r** and **C-w**) are straightforward and, with a little practice, you should have no trouble. The **C-r** and **C-w** commands are used for "recursive editing", which we will discuss in Section 10.11.

Please take another look at Figure 10-7. You will notice there are two other search and replace commands. These are the "no-query" commands:

M-x replace-string makes all the replacements automatically without asking you any questions. This is similar to using **M-%** and then pressing the **!** (exclamation mark) character at the first match. The **M-x replace-string** is handy when you know you want to make all the replacements, and there is no point in stopping at each one to confirm your intentions.

M-x replace-regexp makes all the replacements automatically, while allowing you to use a regular expression for the search pattern.

You will have noticed that the **M-%** and **M-C-%** key sequences are short and easy to type. The no-query commands are not used as much, so they don't have their own key sequences: you need to use **M-x** and specify the command name explicitly. However, you don't actually have to type the entire command name. You can use completion, which we discussed in Section 6.7. (If you have not read about completion, take a moment and do so now: it is an especially handy tool. Briefly, you type part of what you want, and Emacs helps you complete the rest.)

As a reference, Figure 10-9 shows the command names for the four search and replace commands, along with the minimum number of keystrokes you need to type the command with completion. Notice, that I have described two possible key combinations for **M-x query-replace-regexp**. I did this so you could learn a bit more about completion. Take a moment to try both combinations, and see which type you prefer.

Key Sequence	Command	Keystrokes to Use
M-%	**M-x query-replace**	**M-x que RET**
M-C-%	**M-x query-replace-regexp**	**M-x que TAB SPC RET**
M-C-%	**M-x query-replace-regexp**	**M-x que SPC SPC SPC RET**
-none-	**M-x replace-string**	**M-x repl SPC s RET**
-none-	**M-x replace-regexp**	**M-x repl SPC reg RET**

FIGURE 10-9. Minimum keystrokes to invoke search and replace commands.

Section 10.11: Recursive Editing

The last search-related concept I want to discuss is RECURSIVE EDITING. This allows you to put a search and replace operation on hold temporarily, while you perform some editing. When you are finished, you can return to the search and replace that is already in progress.

Here is how you might use this facility. Let's say you are in the middle of a long search and replace operation, and you happen to notice a different change you want to make. At such times, it can be inconvenient to stop what you are doing just to make a single change. However, if you wait until your search and replace operation is finished, you may forget what it was you wanted to change.

Instead, you can press **C-r**. This pauses the search and replace, and puts you back into a recursive editing environment. You can now make any changes you want. When you are finished, press **M-C-c** <Alt-Ctrl-C>. This will stop recursive editing and return you to the search and replace operation, exactly where you left off.

Whenever you press **C-r**, Emacs will put square brackets, **[** and **]**, around the name of the mode on your mode line (see Section 6.4 for a discussion of the mode line). The square brackets are a reminder that you are working in a recursive editing environment.

For example, say if you are editing a buffer named **document**. Your mode line looks like the top line in Figure 10-10. You then type **C-r** to start recursive editing, your mode line now looks like the bottom line in Figure 10-10. When you press **M-C-c** to quit recursive editing, the square brackets will disappear.

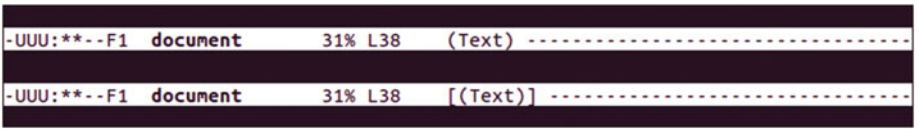

FIGURE 10-10. Mode lines showing a change to recursive editing.

The top mode line shows a buffer named **document** *that is being edited in Text mode (see Section 8.3). The bottom line has square brackets around the name of the mode indicating that you are now in a recursive editing environment.*

Another way to start recursive editing during a search and replace operation (aside from **C-r**) is by pressing **C-w**. This will delete the current matching pattern and then start recursive editing.

Using **C-w** is handy when you want to replace the matching pattern with something that is not your specified replacement. You can delete the match, enter recursive editing, insert the replacement by hand, and then go back to where you were.

For example, say that you are in the middle of changing all the occurrences of **jerk** to **goofball**. You happen upon a particular occurrence of **jerk** that you think should really be **nice guy**. Press **C-w**. This will erase the word **jerk** and place you in recursive editing. Type **nice guy**. Then press **M-C-c** to return to the search and replace operation.

Note

While you are in recursive editing, it is possible to start another search and replace operation. While it is active, if you type **C-r** or **C-w** you will enter a second level of recursive editing. When this happens, you will see double square brackets around the mode name.

If you do the whole thing again, you will be in yet another level of recursive editing, and you will see triple square brackets around the mode name. You can see this in Figure 10-11.

In fact, you can go as deep as you want. Eventually, of course, you will have to press **M-C-c** enough times to work your way back out to the top level.

FIGURE 10-11. Mode line showing triple-level recursive editing.

While you are in recursive editing, it is possible to start another search and replace, and put that operation into recursive editing. You can do this as many times as you want. The mode line will show you one set of square brackets for each level of recursive editing. In this case, there are three levels. To work your way back out to the top, press **M-C-c** *once for each level.*

CHAPTER 11

Modes; Customizing Using Your .emacs File

Section 11.1: Introducing Modes

Emacs was designed to be flexible, especially when you become an advanced user or if you are a programmer. The designers of Emacs realized that your needs will vary depending on what you are doing. For example, if you are writing English prose, you will be typing and editing in a different manner than if you are, say, debugging a computer program. One of the ways in which Emacs helps you is by acting slightly differently depending on what you are trying to do. For example, when you are typing a computer program, you will want to indent your lines differently than when you are composing an essay.

For this reason, Emacs uses what are called "major modes" and "minor modes" to enable you to modify your working environment. Major modes are used to edit particular types of text. For example, there are separate major modes for editing plain text (such as English, HTML, TeX, LaTeX, and outlines), and for writing programs (with Lisp, SQL, Java, JavaScript, C, C++, Ruby, Python, and so on). You use only one major mode at a time, whichever is the most appropriate for the work at hand.

Minor modes provide a variety of different features that you can turn on and off to suit your needs. You can use as many minor modes as you need.

All modes, major and minor, have names that end with **`-mode`**, for example, **`text-mode`** and **`emacs-lisp-mode`**. Each mode is defined by an Emacs Lisp function that has the name of the mode with the file extension **`.el`**, for example, **`text-mode.el`** and **`emacs-lisp-mode.el`**. (The file extension **`.el`** stands for "Emacs Lisp".) Once you learn how to understand modes and how to write Lisp programs, you will be able to read some such files to see exactly what they are doing. You may even choose to copy and modify some of them to create your own customized modes.

Emacs modes can be complicated, but they are important, so we are going to take some time to talk about the details. In fact, it won't be until Section 11.5 that I will actually show you how to use the commands you need to use individual modes. Until then, all I want you to do is read, think and learn. The foundation you are about to build is important.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_11

Section 11.2: Major Modes

Each buffer uses one major mode, which you can change as the need arises. The MAJOR MODE affects the behavior of Emacs as you work within that buffer. For example, certain key sequences may be redefined in a way that is appropriate for the type of work you are doing. The name of the major mode is displayed in parentheses on the mode line (see Section 6.4) near the bottom of the window. You can see an example in Figure 11-1. In this example, the name of the buffer is **`starting-with-emacs`**, and its major mode is Fundamental mode.

FIGURE 11-1. Typical mode line showing the name of major mode.
In this example, the buffer is named **`starting-with-emacs`***, and the buffer is in Fundamental mode.*

If you find that there is a particular major mode you use a lot of the time, you can customize your working environment so that, by default, Emacs automatically starts in that major mode (see Section 11.9). For example, if you are writing a lot of notes, you might set your default major mode to **`text-mode`**. If you spend most of your time working with Lisp programs, you might make **`emacs-lisp-mode`** your default mode.

Emacs has a great many major modes. However, almost all of them are based, directly or indirectly, on one of the four basic major modes that you see in Figure 11-2. A PARENT MODE is a mode from which other modes are created. We say that such modes are DERIVED from the parent mode. In this way, the four basic Emacs major modes are the parents (or grandparents) of almost all the other major modes.

	Name	Type of Data to Edit
Fundamental Mode	**`fundamental-mode`**	Anything Emacs doesn't know about
Prog Mode	**`prog-mode`**	Programming source code
Text Mode	**`text-mode`**	Human-readable text
Special Mode	**`special-mode`**	Special text created by Emacs

FIGURE 11-2. The Four Basic Major Modes.
Almost all the major modes are derived, directly or indirectly, from one of these four basic modes.

FUNDAMENTAL MODE is the least specialized major mode. In Fundamental mode, every Emacs command behaves in its most general manner: there are no mode-specific definitions or variable settings, and all user options are in their default state. As such, Fundamental mode is the best one to use when you are learning, or if you aren't sure what mode to use.

When Emacs loads a file into a buffer, if the file name has an extension, Emacs will set the appropriate major mode for that type of data, whenever possible. For example, if you load the file **`notes.txt`**, Emacs will set the major mode for that buffer to **`text-mode`**. If you load the file **`calculate.el`**, Emacs will set the major mode to **`emacs-lisp-mode`**. If Emacs can't figure out what type of file you are loading, it will use **`fundamental-mode`** as a default. This is really the main use for Fundamental mode. It is not intended to be used as a parent mode, although in a few cases, it has been customized to create other modes.

PROG MODE is for working with programming language source code. (As we discussed in Section 1.3, source code refers to instructions written in a human-readable programming language.) However, Prog mode itself is rarely used. Instead, it is the parent mode from which many specific programming major modes are derived. For example, **`emacs-lisp-mode`**, which is used to write Emacs Lisp programs, is derived from **`prog-mode`**.

TEXT MODE is used for working with human languages (as opposed to programming languages). For example, you would use Text mode for editing text in English or other languages. As such, Text mode is often used for basic editing. It is also the parent mode of other major modes designed to work with marked-up text. For example, **`tex-mode`**, which is used to edit TeX files, is derived from **`text-mode`**. And **`html-mode`**, which is used to edit HTML (Web page) files, is derived from **`sgml-mode`**, which is derived from **`text-mode`**.

SPECIAL MODE is the parent of major modes that are used for buffers displaying text that Emacs itself generates. For example, when Emacs creates a buffer to display a Buffer List, or Help information, or a Completion List, it uses a major mode derived from **`special-mode`**. Normally, you would never use Special mode yourself: it is used only by Emacs programs that need to create a buffer in which to display information.

Section 11.3: Lists of Major Modes

For reference, this section contains tables showing the most useful and popular Emacs major modes, organized into families based on their parent mode. If you want, you can think of these tables as the family trees for Emacs major modes.

You do not have to understand, or even know about, all the major modes. Read through the lists and see if any major modes relate to your work or your interests. When you gain more experience, you can come back to this section and look for an appropriate major mode for whatever you are working on at the time.

I'll show you how to set the major mode you want to use in Section 11.5, and how to learn more about a specific mode in Section 11.7. For now, just take a moment to introduce yourself to the various major modes and see how they are used in Figures 11-3 to 11-7.

Fundamental Family	Type of Data to Edit
fundamental-mode	Parent mode: General data, not specialized
array-mode	2-dimensional arrays
css-mode	CSS files (Cascading Style Sheets)
edit-abbrevs-mode	Abbreviation definitions
gud-mode	Using debuggers: **gdb**, **sbd**, **dbx**, **xdb**...

FIGURE 11-3. Major Modes: Fundamental Mode Family.

The major modes in this list are derived from Fundamental mode.

Text Family	Type of Data to Edit
text-mode	Parent mode: human-readable text
bibtex-mode	BibTeX files
change-log-mode	Change logs
html-mode	HTML files
indented-text-mode	Text with indented paragraphs
latex-mode	LaTeX-formatted files
mail-mode	Outgoing email messages
nroffmode	**nroff**- and **troff**-formatted text files
nxml-mode	XML files
org-mode	Outlines for "keeping track of everything"
outline-mode	Outlines with selective display
paragraph-indent-text-mode	Text: leading spaces start paragraphs
plain-tex-mode	TeX-formatted files
rmail-mode	Reading email
sgml-mode	SGML files
slitex-mode	SliTeX-formatted files
tex-mode	TeX-, LaTeX- or SliTeX-formatted files
texinfo-mode	TeXinfo files

FIGURE 11-4. Major Modes: Text Mode Family.

The major modes in this list are derived, directly or indirectly, from Text mode. These modes are used to edit human-readable text (English and other languages), as well as marked-up text (HTML, TeX, and so on).

Prog Family	Type of Data to Edit
prog-mode	Parent mode: Programming source code
ada-mode	Ada programs
antlr-mode	ANTLR grammar files (parsers, etc)
asm-mode	Assembly language programs
awk-mode	**awk** scripts
bat-mode	DOS/Windows batch files
c++-mode	C++ programs
c-mode	C programs
cperl-mode	Perl scripts (alternative to **perl-mode**)
emacs-lisp-mode	**emacs** Lisp programs
f90-mode	Fortran 90/95 programs
fortran-mode	Fortran programs
java-mode	Java programs
javascript-mode	JavaScript programs
lisp-interaction-mode	Evaluating Emacs Lisp forms
lisp-mode	Non-**emacs** lisp programs
m4-mode	M4 macros
makefile-mode	Makefiles
modula-2-mode	Modula-2 programs
opascal-mode	Object Pascal programs
pascal-mode	Pascal programs
perl-mode	Perl scripts
prolog-mode	Prolog programs
ps-mode	Postscript files
scheme-mode	Scheme programs (dialect of Lisp)
sieve-mode	Sieve (email filtering) scripts
simula-mode	Simula programs
sh-mode	Shell scripts
sql-mode	SQL programs
tcl-mode	tcl scripts

FIGURE 11-5. Major Modes: Prog Mode Family.

The major modes in this list are derived, directly or indirectly, from Prog mode. These modes are used to edit computer programs.

Special Family	Used for...
special-mode	Parent mode: built-in Emacs services
help-mode	Emacs Help system
messages-buffer-mode	Display messages in ***Messages*** buffer
tar-mode	Looking inside a **tar** file
todo-mode	Managing todo lists

FIGURE 11-6. Major Modes: Special Mode Family.

The major modes in this list are derived from Special mode. As a general rule, these modes are used by Emacs when it needs to display information, for example, **help-mode**. *However, some of these modes are also used by tools, such as* **tar-mode** *and* **todo-mode**.

Independent Modes	Type of Data to Edit
conf-mode	Configuration files
completion-list-mode	Display a list of possible completions
dired-mode	Dired directory/file tool
doc-view-mode	Documents: MS Office, OpenDocument, PDF, PS
forms-mode	Field-structured data using a form
hexl-mode	Hexadecimal data, ASCII data
Info-mode	Emacs Info system
normal-mode	Reset major mode to the default for this file
picture-mode	Text-based drawings
ses-mode	Simple Emacs Spreadsheet files

FIGURE 11-7. Independent Major Modes.

The major modes in this list are not derived from another major mode. Instead, they are independent tools, written to stand alone.

Section 11.4: Minor Modes

Emacs has a large number of optional features you can turn on or off that affect your work with the current buffer. These features are called MINOR MODES. Although you can set only one major mode at a time, you can turn on as many minor modes as you want.

I'll show you how to turn on the minor modes you want to use in Section 11.5, and how to learn more about a specific mode in Section 11.7. For now, just take a moment to look at the various minor modes and see what they can do for you in Figure 11-8.

Minor Mode	Description
abbrev-mode	Working with abbreviations
auto-fill-mode	Automatic filling
auto-save-mode	Automatic saving
binary-overwrite-mode	Binary overwriting
compilation-minor-mode	Compiling programs
cua-mode	Use Ctrl-X/C/V for cut/copy/paste
delete-selection-mode	Typed text replaces selection
display-time-mode	Mode line shows: time, load level, mail flag
double-mode	Some keys differ if pressed twice
eldoc-mode	Display info about Lisp function/variable
flyspell-mode	Highlight spelling mistakes in regular text
flyspell-prop-mode	Highlight spelling mistakes in programs
font-lock-mode	Text is fontified as you type
hide-ifdef-mode	Hides certain C code within **#ifdef**
indent-according-to-mode	Indent appropriately for major mode
iso-accents-mode	Display ISO accents
ledit-mode	Editing text to be sent to Lisp
line-number-mode	Mode line shows: line numbers
outline-minor-mode	Work with outlines
overwrite-mode	Overwrite/insert text
paragraph-indent-minor-mode	Text: leading spaces start paragraphs
pending-delete-mode	Same as **delete-selection-mode**
read-only-mode	Buffer contents cannot be changed
resize-minibuffer-mode	Dynamically resize minibuffer
ruler-mode	Header line shows: ruler
show-paren-mode	Highlight matching parentheses, brackets
size-indication-mode	Mode line shows: size of buffer
timeclock-mode-line-display	Track time intervals you spend working
toggle-read-only	Buffer contents cannot be changed
toggle-viper-mode	Turn **viper-mode** off and on
tool-bar-mode	Toggle: Display help on tool bar or mode line
tooltip-mode	Display Emacs toolbar
transient-mark-mode	Highlight region when defined
view-mode	Page through a file (similar to **less** pager)
viper-mode	Turn on (not off) emulation of **vi** text editor
whitespace-mode	Show all whitespace: **SPC**, **TAB**, **RET** chars
whitespace-newline-mode	Show **RET** characters

FIGURE 11-8. Minor modes.

As you can see from Figure 11-8, there are a great many minor modes. In fact, there are a lot more. I have shown you the ones that I think are the most useful or interesting. Choosing minor modes can be confusing at first, so to get you started, here are several minor modes that you will find particularly useful:

- **`auto-fill-mode`**: Sets automatic filling. As you type, automatically breaks lines, so you don't have to press the Enter key at the end of each line.
- **`line-number-mode`**: As you move the cursor, displays the number of the current line on the mode line.
- **`overwrite-mode`**: As you type, characters replace the existing text. Normally, characters are inserted.
- **`read-only-mode`**: The contents of the current buffer cannot be changed.
- **`show-paren-mode`**: When point is on one of a pair of characters — parentheses, brackets, and so on — the matching character is highlighted (very cool!).

Section 11.5: Setting Major and Minor Modes

To set a major or minor mode, type the **`M-x`** command (see Section 6.1) followed by the name of the mode. For example, to set the major mode to Text mode, use the command **`M-x text-mode`**. To set the minor mode to Overwrite mode, use **`M-x overwrite-mode`**. Since mode names can be long, you will want to be skillful at using completion (see Section 6.7), so Emacs will do most of the typing for you. For example, to set Overwrite mode, you need only type **`M-x ov RET`** (**`M-x ov`**<Enter>). Take a moment and try it.

■ **Note** When you are not sure which major mode to use, use Fundamental mode.

Most of the minor modes act as on/off switches (sometimes called TOGGLES). So if Overwrite mode is off, you can turn it on by using the command **`M-x overwrite-mode`**. When you want to turn it off, simply use **`M-x overwrite-mode`** again.

Every buffer must have only one major mode, but you can use as many minor modes as you want. If you want to work with more than one type of text at the same time, all you need to do is create more than one buffer, each with its own major mode and assortment of minor modes.

To see which major mode you are using in a buffer, simply look at the mode line, the line near the bottom of the window (see Section 6.4). Emacs displays the name of the major mode, as well as some (but not all) of the minor modes. If a mode name is long, you will see an abbreviation.

Look at the example in Figure 11-9. The buffer contains **`.emacs`**, the Emacs initialization file (see Section 11.8). The major mode is Emacs-Lisp mode (**`emacs-lisp-mode`**), because the buffer contains Lisp code. There is at least one minor mode, **`overwrite-mode`**, which is abbreviated as **`Ovwrt`**. This minor mode has just been turned on, which is why you see the transient message **`Overwrite mode enabled`** in the echo area below the mode line. (We discuss the echo area in Section 6.5.)

FIGURE 11-9. Mode line showing the major mode as well as one minor mode.

In this example, the buffer is named **`.emacs`**. *The major mode is* **`emacs-lisp-mode`**. *In addition, there is a minor mode* **`overwrite-mode`**, *which is abbreviated as* **`Ovwrt`**.

Section 11.6: Read-Only Mode

For practice, here is an example of how to turn on a particularly useful minor mode.

On occasion, you may wish to protect a buffer so that you cannot change its contents accidentally. For example, you may want to use Emacs to read an important file that should not be modified. Or you may have typed some information that, for the time being, you want to remain completely untouched.

To protect such data, you can set Read-only mode. The command is:

`M-x read-only-mode`

Once read-only mode is enabled, Emacs will let you look at the contents of a buffer, but not make any changes. If you try to make a change (say, by typing something), Emacs will display a warning message:

`Buffer is read-only`

To turn off read-only mode, just enter the same command again. Remember, most minor modes act as toggles: off/on switches.

In Section 5.3, I showed you how to start Emacs in Read-only mode by using the command:

`emacs` *file* **`-f read-only-mode`**

If you start Emacs in this way, you can turn off Read-only mode by using the same command:

`M-x read-only-mode`

As a shortcut, you can also use the key sequence **`C-x C-q`**, which is bound to **`read-only-mode`** (see Section 6.4).

■ **Note** Setting Read-only mode within Emacs has nothing to do with Unix read-only file permissions. The Emacs Read-only mode only affects Emacs buffers.

Section 11.7: Learning About Modes

As a reference, Figure 11-10 contains a summary of commands you can use to set modes and to display descriptive information about modes. Notice that all the description commands begin with **C-h**. This is because they are part of the Emacs Help facility, which we will discuss in Section 12.3.

Command	Description
M-x *mode-name* **RET**	Set the specified major or minor mode
C-h v major-mode RET	Display information about current major mode
C-h m	Describe current major and minor modes
C-h f *mode-name* **RET**	Describe the specified mode
C-h f *-mode RET	Display names of all modes
C-h a mode	Display summary of all modes

FIGURE 11-10. Commands to set and describe modes.

At all times, the name of the current major mode for a buffer is stored in a variable named **major-mode**. To see information about the current value of this variable, use the Help command **C-h v**, which displays information about a specific variable:

C-h v major-mode RET

The Help facility will display information about the **major-mode** variable. Near the top will be its value, which will show you the current major mode for the buffer in which you are working.

You can also use the Help facility to describe the current major and minor modes for the buffer in which you are working. The command is:

C-h m

When you use this command you will be surprised. Of course, you will have one major mode enabled, and Emacs will show you what it is. However, you will also find that you have a number of minor modes turned on, most of which will be

unknown to you. For example, while I was editing my **.emacs** file (see the example in Figure 11-9, in Section 11.5), I used the **C-h m** command. Of course, I saw that the major mode was Emacs-Lisp mode, because I was editing an Emacs Lisp file. However, I also found out that there were 14 minor modes enabled! They were:

```
Auto-Composition
Auto-Compression
Auto-Encryption
Blink-Cursor
Electric-Indent
File-Name-Shadow
Font-Lock
Global-Font-Lock
Line-Number
Menu-Bar
Mouse-Wheel
Tool-Bar
Tooltip
Transient-Mark
```

Try **C-h m** for yourself, and see what you get.

As I mentioned in Section 11.1, all modes are implemented as Emacs Lisp functions. The Help command **C-h f** displays information about a function, so you can use this command to display information about a specific mode. Just type **C-h f** followed by the name of the mode, which is also the name of the Lisp function. For example, to display information about Text mode (a major mode):

C-h f text-mode RET

To display information about Auto Fill mode (a minor mode):

C-h f auto-fill-mode RET

Take another look at the example above where I showed you all the minor modes that were enabled when I was editing my **.emacs** file. Suppose you were curious to find out about one of these modes, say, Transient-Mark mode. You can guess that the name of the mode is **transient-mark-mode**, so all you have to do is use the command:

C-h f transient-mark-mode RET

■ **Note**

Help information is easier to read if you maximize the Help window. To do so, move the cursor to the Help window (**C-x o**), and then delete all the other windows (**C-x 1**):

C-x o C-x 1

The first key sequence uses a lowercase letter "o"; the second uses the number "1".

I will remind you that **C-x 1** deletes all the other windows, but it does *not* delete any buffers, so don't worry about losing data. (All of this, including the commands I just used, is explained in Section 7.6.) Once you are finished reading the Help information, you can recall your previous buffer by using **C-x b** (see Section 7.7).

Here is an example you can use for practice:

C-h f transient-mark-mode RET
C-x o C-x 1

There are two ways to display a list of all the major and minor modes. To display a summary of all the modes, use:

C-h a mode
C-x o C-x 1

The **C-h a** command displays information about all the Emacs functions that contain a specific word. (The letter "a" stands for "apropos".) By typing **C-h a mode**, you are asking the Help facility to describe all the functions that contain the word **mode**. The key sequence **C-x o C-x 1** expands the window so it is easier to read.

When you use **C-h a mode**, Emacs will display the name of each mode followed by a one-line description. If you want more information about a specific mode, you can use **C-h f** as I described above.

The second way to look at a list of all the modes is to display only the names. To do so, use **C-h f** with a regular expression that matches all function names ending with **-mode**. (For a discussion of how Emacs uses regular expressions, see Section 10.8.) The command to use is:

C-h f *-mode RET

Emacs will display a long Completion list (see Section 6.7) containing all the names.

Once the list appears, the focus will still be in the minibuffer, because Emacs is waiting for you to type something. However, if you press **M-v**, the command to move up one screenful within a buffer (see Section 8.3), it will have the side effect of moving the focus to the buffer containing the Completion list. You can then expand the window by using **C-x 1**. This makes it easy to move through the list. The entire sequence looks like this:

C-h f *-mode RET
M-v
C-x 1

I know it looks complicated, but it makes sense and it works. Try it for yourself.

Section 11.8: Customizing With the .emacs File; Learning Lisp

Each time Emacs starts, it looks for a file named **`.emacs`** in your home directory. If the file exists, Emacs will read it and execute all the commands it contains as part of the initialization process. By placing commands in your **`.emacs`** file, you can customize just about any facet of your working environment, even to the point of eccentricity. (As we discussed in Section 2.18, files like **`.emacs`** — whose names begin with a period — are called dotfiles. The name **`.emacs`** is pronounced "dot-Emacs".)

In Section 11.9, I will introduce you to using your **`.emacs`** file by showing you an example of how you might customize your working environment. To prepare the way, I need to get a little technical, so don't worry if you don't understand everything right away.

As we discussed in Section 1.4, most of Emacs is written in a computer language called LISP, or more precisely, EMACS LISP. (The name stands for "List Processor".) This means that all the commands in your **`.emacs`** file must be written in Lisp, which can be daunting for a beginner, because Lisp programs look like nothing on Earth. It is not my intention to explain how to program in Lisp: that would be a book in itself. Rather, I will discuss some of the more important customizations you might make and show you a selection of sample commands to place in your **`.emacs`** file.

Note

It is worth your while to learn Lisp for two reasons. First, it will make it possible for you to read Emacs documentation and technical discussions, which often require you to be able to read basic Lisp.

Second, Lisp is fun. It is the second oldest programming language still in widespread use[1] and, once you become comfortable with the Lisp way of thinking, it is enjoyable to learn and to use. I first started to work with Lisp back in the early 1970s, and I can tell you, using Lisp is a lot of *fun*. In fact, the only programming language I can think of that is as much fun as Lisp — in an entirely different way — is APL.

The basic Lisp concept is an EXPRESSION, and the definition of a Lisp expression is deceptively simple: zero or more elements, separated by whitespace and surrounded by parentheses. Both Lisp code and Lisp data are in the form of expressions. For example, Lisp FUNCTIONS — program modules that can be executed — are built from expressions, and Lisp data also consists of expressions. Since both functions and data are built from the same type of stuff, it is possible for Lisp programs to read and modify other Lisp programs. As such, Lisp was the very first homoiconic programming language.

[1] See the end of Personal Note #5, "GNU's Not Unix?" (Appendix A).

When Lisp processes an expression, we say that Lisp EVALUATES it. This involves reading the expression, parsing it (making sense of it), and then performing the appropriate action. So, if you put Lisp expressions in your **.emacs** file, Emacs will evaluate those expressions as part of the initialization process. This is what allows you to customize your working environment. For example, you can use Lisp expressions to set modes and specify options that suit your needs.

Speaking of modes, when you are editing a Lisp program, there are two modes I suggest you set to help you. First, your major mode should be **emacs-lisp-mode**, the mode designed especially for writing Emacs Lisp programs:

M-x emacs-lisp-mode RET

Indeed, this will happen automatically, if Emacs can figure out that the file you are editing contains Lisp code.

Second, it is helpful to turn on the minor mode **eldoc-mode**. (The name stands for "Emacs Lisp documentation lookup".)

M-x eldoc-mode RET

Once this mode is enabled, whenever point is located at a Lisp function or Lisp variable, Emacs will display useful information in the echo area (the line near the bottom of the window; see Section 6.5). To see what I mean, turn on this minor mode and open a Lisp file. Move the cursor around and see what happens as you point to a function or macro, or a variable. As an example, I turned on **eldoc-mode** and opened a file. I then moved the cursor to the word **defun**. In the echo area, I saw the message:

defun: (NAME ARGLIST &optional DOCSTRING DECL &rest BODY)

This is a technical summary of the syntax for the macro **defun**, which is used to define a function.

Finally, when you create a **.emacs** file, you can put in descriptive COMMENTS that are ignored by Lisp. Whenever you write Lisp code — or any code for that matter — it is a good habit to use comments to make it easy for you and other people to understand the file. (If you think that your code will never be read by anyone else, think again. Tomorrow, you will be a different person.)

Lisp considers a comment to be any line that begins with a **;** (semi-colon) character. You will see such comments in all our examples. When you encounter people who don't think that comments are necessary, especially in Lisp programs, feel secure in the knowledge that you and I are right, and they are wrong.

Section 11.9: Using Your .emacs File to Set Default Modes

As I explained in Section 9.7, Emacs has a minor mode called Auto Fill mode. When Auto Fill mode is turned on, Emacs will automatically break lines for you as you type. By default, this mode is turned off: if you want to turn it on, you must use the command **M-x auto-fill-mode**.

If you spend most of your time editing regular text, you will probably want to turn on Auto Fill mode (a minor mode) each time you use Text mode (a major mode). To do so, you can place the following lines in your **.emacs** file. The line that starts with the **;** (semi-colon) character is a comment.

```
; set Auto Fill mode as the default for Text mode
(setq text-mode-hook 'turn-on-auto-fill)
```

Do not leave out the parentheses or the single quote (apostrophe) character. Also, be sure to notice that there is only one single quote.

This particular Lisp expression uses what is called a VARIABLE: a quantity, with a name, that stores a particular value. Lisp and Emacs make widespread use of variables, and you can modify countless facets of Emacs' behavior simply by changing the value of some variable or another.

In this case, the variable is named **text-mode-hook**. The effect of the expression is to give this variable a value of **turn-on-auto-fill**. The single quote in front of **turn-on-auto-fill** tells Lisp that what follows is an actual value and not an expression that needs to be evaluated.

The purpose of this expression is to modify what Emacs does each time it turns on Text mode. We are making use of the fact that whenever Emacs starts Text mode, it looks at the **text-mode-hook** variable and executes its value as a command. In this case, the command will be **turn-on-auto-fill**. Thus, once you put this line of Lisp in your **.emacs** file, Auto Fill mode will be turned on automatically each time you turn on Text mode.

In Emacs, a HOOK is a variable whose value is a function that is evaluated automatically whenever a certain condition arises. For example, Emacs looks at the variable **text-mode-hook** each time Text mode is turned on.

Now let's take another look at the same expression:

```
(setq text-mode-hook 'turn-on-auto-fill)
```

Notice the word **setq**. This is the name of a FUNCTION: something that Lisp can execute. (All Lisp programs consist of one or more functions.)

For our purposes, we don't need to get too technical. All you need to understand is that when Lisp evaluates a function, something happens. In this case, what happens is that the **setq** function sets the value of a variable (**text-mode-hook**) to a particular value (**turn-on-auto-fill**). (The **q** in **setq** stands for "quote", but I can't explain why without going into esoteric details as to the basic nature of Lisp.)

Let's look at another example. Whenever you create a new buffer, Emacs will, by default, turn on Fundamental mode. However, it may be that you use Emacs only for creating regular text documents and that, for you, it would be more convenient to have all your new buffers use Text mode. To do so, use the following lines in your **.emacs** file:

```
; set Text mode as the default major mode
(setq-default major-mode 'text-mode)
```

Here we are using a function named **setq-default**. The purpose of **setq-default** is to set the default value of a particular variable. In this case, we are giving the variable named **major-mode** a default value of **text-mode**.

Of course, you can change this command and use another major mode as the default, simply by substituting a different mode name. (See Section 11.3 for the lists of major modes.) If you do change the command in this way, be sure not to omit the parentheses or the single quote.

A moment ago, we looked at an expression that tells Emacs to turn on Auto Fill mode each time Text mode is turned on:

```
(setq text-mode-hook 'turn-on-auto-fill)
```

You may decide that you would like Auto Fill mode to be turned on automatically for all major modes, not just for Text mode. To do so, use the following command instead of the previous one:

```
; set Auto Fill mode as the default for all major modes
(setq-default auto-fill-hook 'do-auto-fill)
```

In this example, we are telling Lisp to set the default value of the variable named **auto-fill-hook** to **do-auto-fill**.

To summarize, you can customize your Emacs environment by putting the appropriate expressions in your **.emacs** file. Each time Emacs starts, it will evaluate the expressions in this file and perform the appropriate actions. The expressions we discussed in this section are as follows:

- Make Text mode the default for all new buffers:
  (setq-default major-mode 'text-mode)
- Turn on Auto Fill mode automatically for Text mode only:
  (setq text-mode-hook 'turn-on-auto-fill)
- Turn on Auto Fill mode automatically for all major modes:
  (setq-default auto-fill-hook 'do-auto-fill)

CHAPTER 12

Shell Commands; Help and Info; Programs and Games

Section 12.1: Entering Shell Commands

There are several ways you can use shell commands without having to leave the Emacs environment. These tools are summarized in Figure 12-1.

Command	Description
M-!	Run a shell command
M-|	Run a shell command using region as input
M-x shell	Start a separate shell in its own buffer

FIGURE 12-1. Running shell commands.

To enter a single shell command, type **M-!** followed by the command. For example, to display a list of all the userids currently logged into your system, you could use either **M-! users** or **M-! who**. (**users** and **who** are Unix commands.)

When you run a shell command using **M-!**, Emacs always saves the output in a buffer named ***Shell Command Output***. If this buffer does not already exist, Emacs will create it. If it does exist, the previous contents will be replaced. Thus, at any time, this buffer will hold the output from only one command.

If the shell command you enter displays a large amount of output,[1] Emacs will display the ***Shell Command Output*** buffer. However, if the command generates a small amount of output, Emacs will display the output in the echo area, at the bottom of the window (see Section 6.5).

[1] By default, greater than 25 % the height of your Emacs window.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_12

Regardless, the shell output remains in the buffer until it is replaced by the next shell command. So even when the echo area is cleared, you can still switch to the **`*Shell Command Output*`** buffer manually, to see the output of the last shell command:

`C-x b *Shell Command Output*`

You can simplify this command by using completion (Section 6.7):

`C-x b *S TAB`

Another way to run a shell command is to use some of the data in the buffer as input for the command. For example, you may have a large number of lines you would like to sort. All you need to do is use these lines as data for the Unix **`sort`** program.

To perform such an operation, you use the **`M-|`** (<Meta> -vertical-bar) command. This will run whatever shell command you specify, using the contents of the region as input. You may remember that in Section 8.9, "Operating on the Region", we discussed an example that used the **`M-|`** command to create a sorted list of key descriptions. Here is another example.

You want to sort all the lines in the buffer. To do so, you set the region to the entire buffer, and then use **`M-|`** to run the **`sort`** command on the region. Use the following commands:

1. **`C-x h`**: Set the region to be the entire buffer.
2. **`M-| sort`**: Sort all the lines in the buffer.

When you use **`M-|`**, Emacs puts the output into a buffer named **`*Shell Command Output*`** just like when you use the **`M-!`** command. If the buffer does not exist, Emacs will create it. If the buffer does exist, its contents will be replaced.

Sometimes, though, you may want to replace the lines in your original buffer with the output of the shell command. All you need to do is use a prefix argument (explained in Section 8.4). This tells Emacs not to save the output in a special buffer. Any numeric value will do, so you might as well use **`1`**.

Here is an example. Say you are working with a buffer that contains people's names, one name per line. You have just typed in all the names, and now you want to sort them. However, you don't want the output of the **`sort`** command to be saved in a separate buffer; you want the sorted names to replace the original contents of your buffer. Use the commands:

1. **`C-x h`**: Set the region to be the entire buffer.
2. **`ESC 1 M-| sort`**: Sort all the lines in the buffer, replacing the input with the output of the **`sort`** command.

Note When you use the **M-|** command with a prefix argument, the data in your buffer will be replaced by the output of the shell command. If you decide that it was all a mistake, you can undo the effects of the shell command by using the Emacs undo command: either **C-x u** or **C-_** or **C-/** (see Section 7.3).

Section 12.2: Shell Buffers

As we discussed in Section 11.3, you can run a shell command without leaving the buffer in which you are working by using **M-!** and **M-|**. However, sometimes you will want to use more than one shell command. In such cases, it is a lot easier to create a new buffer, just for shell commands.

To create a designated buffer just for running shell commands use **M-x shell**. This command will start a separate shell in its own buffer named ***shell***. You can then enter as many commands as you want, one after the other. Everything you type, along with all of the output, will be saved in the buffer. Whenever you want, you can switch from this buffer to another one. This means you can have a designated buffer just for running shell commands and capturing their output. This makes it easy to edit and then copy the output of a shell command to another buffer.

When you enter the **M-x shell** command, Emacs will check if a buffer named ***shell*** already exists. If not, Emacs will create it. If such a buffer already exists, Emacs will simply switch you to that buffer. When you are finished using that particular shell, you can kill the buffer and stop the shell by using **C-x k** (see Section 7.7).

As I mentioned, if you tell Emacs to create a new shell and a buffer named ***shell*** does not already exist, Emacs will create one. Thus, you can create more than one shell buffer by changing their names. Here is how it works.

The command **rename-uniquely** changes the name of a buffer to be unique. From within the ***shell*** buffer, you can use this command to change the buffer name to something else. You can then use **M-x shell** to start a brand new shell. Since there is no buffer named ***shell***, Emacs will create one and you will have two shells, each in its own buffer. Here is an example.

1. Create your first "shell in a buffer" by typing **M-x shell**. You now have a buffer named ***shell*** that contains a live shell session.

2. Type **M-x rename-uniquely**, and Emacs will change the name of the buffer to ***shell*<2>**.

3. Type **M-x shell** and Emacs will create a second shell in a new buffer named ***shell***.

If you want yet another shell buffer, you can rename the last one and use **M-x shell** again. In this way, you can create as many shell buffers as you want.

Here are two final pieces of advice. First, you do not have to type the full command **M-x rename-uniquely**. Using completion (see Section 6.7), all you need to type is **M-x ren SPC u RET**.

Second, if you are not sure what to get your mother for her birthday, a shell-in-a-buffer is something just about anyone can use.

Section 12.3: The Help Facility

Every Emacs user has access to three comprehensive help systems: the Emacs tutorial, the Emacs reference manuals, and a variety of Help tools. We'll start with the Help tools, and we'll cover the Emacs tutorial and reference manuals in Section 12.4.

To start the Help facility, you type **C-h**. Emacs will then prompt you to type a HELP OPTION. I have summarized the most important Help options in Figure 12-2. If you want to see them all, use either **C-h C-h** or **C-h ?**. if you type **C-h** and then change your mind and decide to quit, simply press **q** (quit).

Command	Description
C-h C-h	Display a summary of all the Help options
C-h ?	Same as **C-h C-h**.
C-h q	Exit from a Help command loop (quit)

Command	Description
C-h a	Show all the functions containing a specified word
C-h b	Display a full list of all the key bindings
C-h c	You specify a key, Emacs tells you what it does
C-h f	You specify a function, Emacs describes it
C-h h	Display the "Hello" file
C-h k	You specify a key, Emacs describes its function
C-h m	Describe the current major and minor modes
C-h v	You specify a variable, Emacs describes it
C-h w	You specify a function, Emacs shows you its key

Command	Description
C-h t	Emacs tutorial
C-h i	Emacs reference manuals (Info documentation browser)

FIGURE 12-2. Help facility options.

In a moment, I'll go over each of these commands. Before we do, here is an important technique I want you to remember.

Many of the Help options display information in a buffer named **`*Help*`**. This information will be a lot easier to read if you maximize the window containing the **`*Help*`** buffer. To do so, use **`C-x o`** to move the cursor to the Help window, and then use **`C-x 1`** to delete all the other windows. (The first key sequence uses a lowercase letter "o"; the second uses the number "1".)

```
C-x o C-x 1
```

I will remind you that when **`C-x 1`** deletes all the other windows, it does *not* delete any buffers, so don't worry about losing data. (All of this, including the commands I just used, is explained in Section 7.6.) Once you are finished reading the Help information, you can recall your previous buffer by using **`C-x b`** (see Section 7.7).

- **`C-h b`**: To see a full list of all the key bindings, use the **`C-h b`** command. This list is more interesting than you might think and is worth checking out from time to time. The more you know, the more you will find interesting key bindings to explore.
- **`C-h c`**: Use this option to find out the name of the function to which a key is mapped. Most of the time, this is enough to tell you what a key does, because Emacs function names are chosen to be descriptive. For example, say that you are wondering what the **`C-x C-w`** key sequence does. Press **`C-h c C-x C-w`**. You will see:

```
C-x C-w runs the command write-file
```

- **`C-h k`**: To display more detailed information about the function, use **`C-h k`**. (This is one of my favorites.) For example, to find out more about the function to which the **`C-x C-w`** key sequence is mapped, use the commands:

```
C-h k C-x C-w
C-x o C-x 1
```

Try it, and see what you get. Remember, the second key sequence maximizes the Help window to make it easier to read.

Note When you have a spare moment, use **`C-h b`** to display the master list of key bindings. Scan this list and find yourself an unfamiliar key that looks interesting. Then use the **`C-h k`** command to find out about that key.

- **`C-h w`**: Conversely, if you know the name of a function, and you want to know what key sequence it uses, type **`C-h w`** (the "where is?" command). For example, if you type **`C-h w write-file`**, you will see:

```
write-file is on C-x C-w
```

When you start working with functions and variables, there are three Help options you will find handy:

- **C-h a**: The "apropos" option shows you all the functions whose names contain a specific regular expression. Usually, you specify a word because you are interested in seeing all the functions whose names contain that word. For example, say that you want to see all the functions that have something to do with killing; that is, deleting text while copying it to the kill ring (see Section 9.1):

```
C-h a kill
C-x o C-x 1
```

- **C-h f**: If you know the name of a function, you can display a description of it by using the **C-h f** command, for example:

```
C-h f write-file
C-x o C-x 1
```

- **C-h v**: Similarly, you can use **C-h v** to describe a variable:

```
C-h v visible-bell
```

- **C-h m**: You can use **C-h m** to display the major and minor modes for the current buffer along with a short description of each one. (See Section 11.7 for a discussion of this command.)

- **C-h h**: Finally, just for fun, **C-h h** displays a list of how to say "Hello" in many different languages. The purpose of this file is to demonstrate a variety of the different characters sets that Emacs supports. However, this file is interesting in its own right.

Section 12.4: The Emacs Tutorial; Info and the Emacs Reference Manuals

Aside from the Help options we discussed in Section 12.3 (Figure 12-2), Emacs also comes with two full documentation systems: the Emacs tutorial and the Emacs reference manuals.

The purpose of the tutorial is to let you teach yourself basic Emacs at your own speed. However, the tutorial can be confusing for beginners, which is why I told you (in Section 4.1) that you will get a much better introduction to Emacs by reading this book. Since you have read this far, however, you already know the basics, so taking the tutorial now will actually serve as a nice review.

To start the Emacs tutorial, type **C-h t**. All you have to do is read from the beginning, and follow the directions as you go.

The Emacs reference manuals are more elaborate, but not for beginners. You access them by using the INFO FACILITY, or INFO, an elaborate tree-structured documentation browser. The Info facility is designed to display documentation consisting of documents that are connected to one another.

To start Info, type **C-h i**. Emacs will create a new buffer named ***info*** to display the top of the tree. This document begins with some basic instructions (worth reading carefully), followed by a table of contents for the Emacs reference manuals: a long list of topics in the form of links. (The version I looked at had 245 such topics.) To navigate, all you have to do is move the cursor to the topic you want to read, and press **RET** (the Enter key). Info will then display the document for that topic.

The two most important topics are **Info**, the manual for the Info facility itself, and **Emacs**, a comprehensive reference manual for Emacs concepts and commands.

As you will see, the Info facility has its own special commands. However, you can still use many of the regular Emacs commands. In particular, you can start Info, read for a while, and then use **C-x b** to change to another buffer. Later, you can return to Info by changing back to the ***info*** buffer.

The best way to learn how to use the Info facility is to start with the built-in documentation. Type the **C-h i** command, and then press **h** to start the Info tutorial. Follow the instructions and work your way through the various topics. Once you know how to use Info, you will be able to read parts of the Emacs manual whenever you want.

It is important — very important — that you learn how to use the Info facility, because it is your doorway to the Emacs reference manuals. If you don't learn how to use Info, you will have no way to find out detailed information about Emacs. The Emacs Help facility contains only summaries. The real information is in the reference manuals — and to get at them, you need Info.

I won't go into all the details of using the Info facility: they are best learned by following the Info tutorial and by practicing. However, I will cover the basic concepts and give you summaries of the important Info commands. The general commands are shown in Figure 12-3.

Command	Description
?	Display a summary of Info commands
h	Start the Info tutorial
q	Quit Info: remember current location
C-x k	Quit Info: do not remember current location

FIGURE 12-3. General Info commands.

Within the Info facility, certain keys work differently than they do with regular Emacs. For example, as you are reading a topic, you display the next screenful by pressing **SPC** (<Space>). You move back to the previous screenful, by pressing **BS** (<Backspace>). With Emacs, you would press **C-v** to move forward and **M-v** to move backward.

The Info facility organizes information into short topics called NODES. ("Node" is a technical term borrowed from a branch of mathematics called graph theory.) Each node has a name and contains information about one specific topic. As you read, you can move from one node to another.

The nodes themselves are organized into an upside-down tree. The top node is what would be the trunk in a real tree. When you start Info for the first time, you will see the top node: *The Info Directory*.

Whenever you select a node, Info will display it for you to read. As you read, there are commands to move within the node, or to jump to another node. These commands are summarized in Figures 12-4 and 12-5.

Command	Description
n	Jump to next node in the sequence
p	Jump to previous node in the sequence
u	Jump to the "up" node (the menu you came from)
l	Jump to last node you looked at
m *selection*	Pick a node from a menu
f	Follow a cross-reference
i	Look up topic in the index, then jump there
,	(comma) Jump to next match from previous **i** command

FIGURE 12-4. Info commands to select a node.

Command	Description
SPC	Go forward (down) one screenful
BS	Go backward (up) one screenful
b	Go to beginning of the node
.	Same as **b**
C-l	(<Ctrl-L>) Redisplay the current screen

FIGURE 12-5. Info commands to read a node.

When you are finished reading, you have three choices:

- Use an Emacs command (such as **C-x o** or **C-x b**) to switch to another buffer. This leaves your Info session alive in its own buffer.
- Use the **q** (quit) command to stop Info completely.
- Use the Emacs **C-x k** command to kill the Info window.

When you quit with the **q** command, Emacs remembers your location within the Info tree. If you later restart the Info facility (with another **C-h i** command), Emacs will put you back where you were when you quit. When you quit by using **C-x k** to kill the buffer, Emacs does not remember your location. The next time you start Info, you will be back at the top node of the tree (the main menu).

Note

To learn how to use the Info facility, type **C-h i** and then press **h** to start the tutorial.

To quit, press **q**. That way, the next time you start Info, you can continue from where you left off.

Section 12.5: Built-In Programs

Over the years, Emacs has provided a variety of built-in programs: tools and diversions (including games). With the growth of the Internet, some of these programs have become obsolete. Others, however, are still useful and interesting and, once in a while, someone adds a new program.

Because these programs are important (or fun), I will give you a quick guided tour so you can decide which ones appeal to you. In Section 12.6, we'll talk about the built-in Emacs tools. In Section 12.7, we'll talk about the games and diversions. Before we start, however, I want to take a moment to explain the general principles that apply to all of these programs.

- Starting a Program

To run a program, type **M-x** followed by the name of the program. For example, to run the **calendar** program, use:

```
M-x calendar
```

Emacs will create a new buffer in which to run the program.

- Stopping a Program

There are several ways to stop a program:

1. The program itself may have its own quit command, such as **q**.
2. Use the standard Emacs quit command, **C-g** (**keyboard-quit**), to stop the program and preserve the buffer.
3. Use **C-x k** (see Section 7.3) to kill the buffer in which the program is running.

- Putting a program on hold

While you are using a program, you can change to a different buffer (by using, for example, the **C-x b** or **C-x o** commands). Then you can return to the program later, and work with it some more.

- Learning how to use a program

There are various ways to get information about a program. First, try the Info facility (Section 12.4). Look for the program within the Emacs reference manual and see if you can find some documentation. The larger programs have menus and submenus of their own. The smaller programs may have only a brief mention.

Second, use the Help facility to tell you what it can about the actual function. For example, to find out about **blackbox**, type **C-h f blackbox**.

Third, start the program and see if it has a built-in help command. For instance, with **dunnet**, the **help** command will display a help document. With **dired**, pressing the letter **h** will display help information.

Section 12.6: Built-In Tools, Including Dired

In this section, I will introduce you to several important tools, which are summarized in Figure 12-6.

Program	Description
calendar	Calendar and diary
customize	Tool to help you change user options
dired	Directory and file manager
eww	Web browser
ses	Create and edit spreadsheet files

FIGURE 12-6. Built-in tools.

I'm going to start with Dired because it is the most important. The **dired** ("directory editor") program provides a complete interface for working with files and directories, letting you edit a directory as easily as you can edit a text file. Specifically, you can perform all the common file and directory operations, such as copy, move, and rename. You can point to a file and tell Emacs to show it to you. You can print a file, or compress and uncompress it.

What I like best about Dired is that it makes it easy to maintain your directory tree by creating and removing directories, and by moving files from one place to another. It does so, by acting as a front end to the Unix directory and file commands, such as **ls**. (With Microsoft Windows, Dired emulates **ls**.)

Note

Dired is a powerful file manager, which makes it an important tool for everyone. However, in order to use a file manager, you need to understand files, directories, and the tree-structured file system.

These are extremely important topics, and if you need some help, please take the time to review Sections 2.15 through 2.19.

DIRED FILE MANAGER (dired)

You can start Dired in two different ways. First, you can use **M-x**, as you would for any program:

```
M-x dired
```

Alternatively, because Dired is so important, it has its own command, **C-x d**. If you want to start Dired this way, here are two possible key sequences you can use:

```
C-x d
C-x 4 d
```

C-x d starts Dired in the same way as **M-x dired**. The **C-x 4 d** key sequence starts Dired in another window (see Section 7.7).

After you type the command, press **RET** (<Enter>). Dired will ask you which directory you want to work with, and will make a suggestion. For example, you might see:

```
Dired (directory): ~/
```

In this case, Dired is suggesting that you might want to start with your home directory (~). If you do, simply press <Enter> to start the program. If not, type the name of directory you do want, and press <Enter>. (If you don't understand the abbreviation **~** for your home directory, see Section 2.17.)

Once Dired starts, there are a large number of commands you can use. I have summarized the important ones in Figure 12-7. Note that the command names are case sensitive.

Program	Description
q	Quit
h	Display Help summary
m	Mark current file/directory, move cursor down
BS	Unmark current file/directory, move cursor up
u	Unmark file/directory, move cursor down
U	Unmark all files/directories
C	Copy marked files or current file
R	Rename current file
R	Move marked files or current file to another directory
d	Flag file for deletion
x	Delete files flagged by **d**
g	Refresh by reading the directories again

FIGURE 12-7. Dired commands.

Here are the most important commands while using Dired, the Emacs file manager. Notice that the commands are case sensitive.

One of the most important Dired concepts is that you can MARK files and then operate on them. For example, you might mark, say, 5 files, and then move all 5 of them to another directory. Within Dired, most commands work on marked files. If no files are marked, Dired will operate on the current file (the location of the cursor).

Dired is a powerful, but complex program. There are three ways you can get the information you need to learn how to use it:

1. Use Info to look at the Dired documentation within the Emacs manual:

```
C-h i
m emacs RET
m dired RET
```

2. Within Dired, use the **h** command and read the Help summary.
3. Search online for "dired reference card". You should be able to find a Dired reference card that you can print.

■ **Note** Dired is so useful, I suggest you keep an active copy open in a buffer at all times. That way, you can switch to it instantly as the need arises.

CALENDAR AND DIARY (`calendar`)

The next useful program I want to talk about is **`calendar`**. Each time you start the program, it shows you a three-month calendar, which is handy by itself. However, **`calendar`** is much, much more. One of the basic **`calendar`** tools is a diary, which you can use to keep daily notes and reminders. Within **`calendar`**, you will find just about anything you can imagine that has to do with days and calendars. For example, you can display various types of calendars (such as Hebrew, Islamic, and astronomical); you can find the dates of the various phases of the moon; and you can display local times for sunrise and sunset.

To start the Emacs Calendar and Diary:

`M-x calendar`

Once the program has started, you can press **`?`** (question-mark) to display documentation about the **`calendar`** program. Specifically, **`calendar`** will jump to its Info page in the Emacs reference manual. Here you will find all the information you need to teach yourself how to use the program. To quit the program, press **`q`** (quit).

CUSTOMIZE (`customize`)

The next built-in program is **`customize`**. This is a very useful program that provides an interface for finding and modifying any of the huge number of Emacs settings. To start the program:

`M-x customize`

To quit the program, press **`q`** (quit).

The best way to learn how **`customize`** works is to use Info to read the documentation in the Emacs reference manual. To do so, start Info, then search for "easy customization interface". The key sequences to use are:

`C-h i`
`m emacs RET`
`m customization RET`
`m easy customization RET`

EMACS WEB BROWSER (`eww`)

The next program is a built-in Web browser named **`eww`**. (This program became available with Emacs version 24.) The name **`eww`** stands for "Emacs Web Wowser". To start the program, use:

`M-x eww`

Emacs will ask you to enter either a URL (Web address) or search keywords. Once you enter your response, **eww** will find the information you requested and display it in a buffer. To quit the program, press **q** (quit).

To learn about the program, use Info to look at the **eww** manual:

```
C-h i
m eww RET
```

The default search engine for **eww** is DuckDuckGo. This is a small, fast search engine that does not accumulate or share personal information. If you want to try it directly, the URL is:

```
https://www.duckduckgo.com/
```

■ **Note**

You will notice that the Web pages **eww** renders do not look nearly as elaborate or flashy as the pages you see with the popular standalone browsers. In fact, they are mostly text. That's because **eww** was not designed to compete with other browsers. Its purpose is to let you search online quickly, without leaving Emacs.

Because everything you see with **eww** is contained in an Emacs buffer (named ***eww***), it is fast and easy to copy information you find online to another Emacs buffer. That is what makes **eww** such a valuable tool.

SIMPLE EMACS SPREADSHEET (ses)

The final program I want to mention is **ses**, the Simple Emacs Spreadsheet program. Although **ses** doesn't have the power of a large, standalone spreadsheet program, it is handy because it is always available and, as the name says, it's simple.

What I like about **ses** is that, when you want to do a quick calculation that requires a spreadsheet, you don't have to leave Emacs. Moreover, because the spreadsheet you create is in an Emacs buffer, it is easy to copy data to another buffer.

Technically, **ses** is a major mode used to edit spreadsheet files. So the easiest way to start the program is to visit (open) a new file with an extension of **.ses**, for example:

```
C-x C-f calculate.ses
```

To learn how to use **ses**, start with the Help facility:

```
C-h f ses-mode
```

To use Info to read the **ses** documentation from the Emacs reference manual:

```
C-h i
m ses
```

Section 12.7: Games and Diversions

Figure 12-8 shows a summary of the most interesting games and diversions that come with Emacs. I'll go through the list, one at a time, and tell you a bit about each one. In Section 12.8, I'll show you a particularly interesting example of the **doctor** program.

Program	Description
animate	Display animated birthday message
bubbles	Game: Remove groups of bubbles until all gone
blackbox	Puzzle: Find objects inside a black box
doctor	Eliza program: acts like a psychotherapist
dunnet	Adventure-style exploration game
gomoku	Game: plays Gomoku with you
hanoi	Visual solution to Towers of Hanoi problem
landmark	Neuro-network robot that learns about landmarks
life	Game of Life: auto-reproducing patterns (not a game)
mpuz	Multiplication puzzle: guess the digits
pong	Video game: classic ping-pong game
snake	Video game: you control a growing snake
solitaire	Jump pegs across other pegs (not the card game)
spook	Generates words to get government's attention
tetris	Video game: you manipulate falling blocks

FIGURE 12-8. Games and diversions.

Before we begin, I will remind you how to start and stop a program, and how to display help information.

To start a program, type **M-x**, followed by the name of the program, followed by **RET** (<Enter>). For example, to start **doctor**:

M-x doctor RET

There are three ways to stop a program (see Section 12.5). Some programs stop when you press **q** (quit). If that doesn't work, try **C-g** (the **keyboard-quit** command). If all else fails, use **C-x k** to kill the buffer in which the program is running.

To display help information for a game, use **C-h f**, followed by the name of the function that runs the game (the name in Figure 12-8). For example, to learn how to play the Bubbles game, use:

C-h f bubbles RET

For long names, you can use completion (see Section 6.7).

- ANIMATE (to start: **M-x animate**)

(If this doesn't work, use **M-x animate-birthday-present**.)

This program calls upon another program that moves characters around within a buffer to create a very specific message: a birthday greeting to someone you miss who has moved away. When you start the program, **animate** will display:

Birthday present for:

It is asking you to enter the name of a friend. Type any name you want and watch the animated greeting. If you want to hack the program to make it work differently, you can find the source code in a file named **animate.el**. The easy way to edit this file is to use:

C-h f animate-birthday-greeting

Then move the cursor to the reference to **animate.el**, and press <Enter> to visit that file. (Before you start, copy the code to another file, so you don't change the original.)

- BLACKBOX (to start: **M-x blackbox**)

A puzzle game in which you use tomography to find objects hidden inside a box. "Tomography" refers to looking at X-ray images in various planes. (When a doctor talks about a CAT scan, he or she is referring to "computerized axial tomography".)

The program hides 4 balls within a box (the Blackbox) represented by an 8 by 8 grid. You "shoot" rays into the box and observe where they emerge. Your goal is to deduce the positions of the hidden balls using as few rays as possible.

This game is fun to play, but it is a bit complicated. The best way to learn how to play it is by reading the help documentation. The command to use is:

C-h f blackbox

- BUBBLES (to start: **M-x bubbles**)

A game using colored bubbles. Move the cursor to highlight contiguous bubbles of the same color. Press <Enter> to remove highlighted bubbles, and the remaining bubbles will come together. The goal is to remove as many bubbles as you can in as few moves as possible.

- DOCTOR (to start: **M-x doctor**)

The Doctor program is an old one. Originally it was called Eliza, after the Eliza Doolittle character in *Pygmalion* and *My Fair Lady*. Doctor acts like the type of psychotherapist who behaves nonjudgmentally, listening with infinite patience, and prompting you to talk about yourself. (Specifically, Doctor emulates a Rogerian psychologist using person-centered therapy.) This is the perfect program to keep someone busy at the computer, while you are in the next room working on something important.

Using Doctor is easy. You start it in the regular manner:

```
M-x doctor
```

All you have to do is carry on a conversation with Doctor by typing. To start, type whatever you want, then press **RET** (<Enter>) twice. This tells Doctor you are finished, and it will respond. Then you talk, then Doctor talks, and so on. Just remember to press **RET** twice whenever you want a response.

When you are ready to stop talking, type **goodbye**. Then close the buffer by using **C-x k**. Here is a short sample of a conversation you might have with Doctor. Your replies are indented and printed in italics:

```
I am the psychotherapist. Please, describe your problems.
Each time you are finished talking, type RET twice.

  Hello.

How do you do? What brings you to see me?

  I have a problem

Why do you say you have a problem?

  Because my mother hates me.

Is it because your mother hates you that you came to me?

  Goodbye.

My secretary will send you a bill.
```

At this point, you end the conversation by closing the buffer:

```
C-x k
```

Because this is an example, I have kept the conversation short and simple. With a little imagination, you can make the conversation longer and more interesting. In Section 12.8, I will show you a longer, much more unusual conversation with Doctor.

- DUNNET (to start: **M-x dunnet**)

Dunnet is a wonderful text-based exploration game in the style of the original Adventure game, which is still included on many Unix systems. I can remember, in the summer of 1981, when I was in medical school, spending a lot of time playing Adventure at home, using a terminal that printed on thermal paper, connected to a remote computer using a slow modem. Playing Adventure was a lot of fun, and I advise you to spend as much time as you can playing Dunnet.

When you start the program you will see:

```
Dead end
You are at a dead end of a dirt road. The road goes to the east.
In the distance you can see that it will eventually fork off. The
trees here are very tall royal palms, and they are spaced
equidistant from each other.
There is a shovel here.
>
```

Whenever you see a **>** (greater-than) character, it means it is your turn to type something. When you are finished playing, type **quit**.

One of the challenges of Dunnet is to figure out where you are, and what you are supposed to be doing (just like real life), so I won't give you any advice. Just start typing and see what happens.

(Okay, one hint: If you are stuck, ask for help.)

- GOMOKU (to start: **M-x gomoku**)

Gomoku is played on a board with 15x15 squares. You play against the program. To win, you must mark off squares in such a way that you get five in a row. The program will try to block you and to mark five consecutive squares of its own.

You and the program take turns marking free squares. When it is your turn, move the cursor to a free square and mark it by pressing **RET** (<Enter>).

To quit, press **q**.

- TOWERS OF HANOI (to start: **M-x hanoi**)

The **hanoi** program displays a visual solution to a mathematical puzzle called the Towers of Hanoi. There are three poles, lined up in a row, as well as a number of different-sized discs with holes. To start, the discs are stacked from largest to smallest on the leftmost pole. The problem is to figure out how to move all the discs from the leftmost pole to another pole without ever placing a larger disc on top of a smaller disc.

If you were to design a computer program to solve this problem, you would find that it lends itself to what programmers call a "recursive" solution. The Lisp programming language (in which Emacs is written) is designed to make recursive programming easy, so it makes sense to find such a demonstration program in a Lisp-based system like Emacs.

To start the program, use **ESC** and a number to specify how many discs you want, followed by **M-x hanoi**. To stop the program, type **q** (quit). For example, to run the program with 5 discs, use:

ESC 5 M-x hanoi

If you don't specify a number, **hanoi** will use 3 discs. Thus, the following commands are equivalent:

M-x hanoi
ESC 3 M-x hanoi

This is a good way to test the program for the first time, as the three-disc solution is over quickly.

- GAME OF LIFE (to start: **M-x life**)

The Game of Life is based on a simple two-dimensional grid-like universe. The Game of Life was invented by John Horton Conway and introduced publicly by Martin Gardner in the October 1970 issue of *Scientific American*.

A number of life forms, called cells, exist. Each cell fills exactly one position in the grid, which is surrounded by eight other positions. The cells reproduce according to a few well-defined rules:

1. If a cell is surrounded by four or more cells, it dies of overcrowding.
2. If a cell is all alone or has only one neighbor, the cell dies of loneliness.
3. An empty location will be filled with a brand new cell, if that location has exactly three neighbors.

As the game progresses, you can watch the patterns change from one generation to the next.

You start the game in the regular manner:

M-x life

The starting pattern of cells is generated randomly. Every generation, the cells change according to the rules I explained above. To quit, use **C-x k** to kill the buffer.

By default, the length of each generation is 1 second. If you want to slow down the program, start it with a prefix number specifying the duration of a generation in seconds. For example, to run the program with 3-second generations, use:

ESC 3 M-x life

- LANDMARK (to start: **M-x landmark**)

Landmark is a diversion in which you watch a neuro-network robot, represented by a small black rectangle, move around a flat grid labeled with the directions of the compass: north, south, east, and west. The robot's goal is to look for a tree at the center of the grid.

The robot figures out which way to move by chemotaxis: moving in response to simulated smells it detects from each of the four directions. However, as the smell of the tree increases, the robot trains itself by adjusting its "weights". This helps it learn how to move in the direction of the tree in the future.

At first, it's mostly trial and error, but it won't take long for the robot to find the tree, and every time it does, it adjusts its weights, which makes it better and better at tree-finding.

The robot stops when it finds the tree, but you can tell it to start again by pressing <Space>. Each time, you can decide whether or not it is allowed to retain its weights (learned skills). If you tell the robot to play the game over and over, you can watch it get better and better at finding the tree. When you get tired of watching a simulated robot look for an imaginary tree using clues that you can't detect, you can quit the program by pressing **q**.

When you start the program, you have some control over what happens. For details, use:

```
C-h f landmark
```

In addition, within the Help information, you will see a link to **landmark.el**. Follow this link to the source code for the program, where you will find a lot of informative comments that explain about the robot. (Use **M-<** to jump to the top of the buffer and then scroll down. Be sure to look through the whole program for comments.)

- MULTIPLICATION PUZZLE (to start: **M-x mpuz**)

This program creates a multiplication calculation of medium complexity. However, each different digit is represented by a letter and not a number. Here is an example:

```
    A E C
    x B H
  -------
  E I I D
H B H I
I C C C D
```

Your goal is to guess which numbers are where. To do so, you press a letter, say **i** (you don't need to hold down <Shift>). The program will ask you to guess which number the letter **I** represents:

```
I =
```

Type a single digit, and see if you are correct. If so, the program will change all the occurrences of that letter to the correct number. For example, if you guessed that I = 6, you would see:

```
    A E C
   x  B H
  -------
  E 6 6 D
H B H 6
6 C C C D
```

Keep going until you have figured out all the numbers.

If you have ever played Hangman, **mpuz** is similar, except you are guessing numbers and not letters. The technical term for this type of puzzle is an ALPHAMERIC. When I was a child growing up in Canada, one of the newspapers published a daily alphameric, and I used to have fun figuring it out. Try one and see how you like it.

Solving an alphameric involves creating and using various strategies. For example, **H** and **A** can't be very low numbers, say 1 and 2, because when you multiply them together, the product has to be large enough to produce the extra digit **E**. Also, notice that when you multiply **H** times **D**, the product ends with **D**. Thus, neither of these letters can represent **1**.

To help you get started, here is the solution to the problem above:

```
    7 3 2
   x  8 5
  -------
  3 6 6 0
5 8 5 6
6 2 2 2 0
```

- PONG VIDEO GAME (to start: **M-x pong**)

This is the Emacs version of Pong, the very first sports arcade video game (released in November 1972). In its time, Pong was incredibly popular.

The goal of Pong is to beat your opponent in a simulated ping-pong game. As the ball moves left and right, you move your paddle up and down in order to reflect the ball back to your opponent. When someone misses, his opponent gets a point.

To control the left paddle: use <Left> (paddle moves up) and <Right> (paddle moves down). To control the right paddle, use <Up> and <Down>. You can play against another person, or you can control both paddles and play against yourself.

To pause the game, press **p**. To continue, press **p** again. To quit press **q**.

- SNAKE (to start: **M-x snake**)

This is the Emacs version of the Snake video game. It takes place in a buffer named ***Snake***.

A yellow snake moves around a grid, leaving red blocks behind it as it moves. You control the head of the snake by using <Left>, <Right>, <Down>, and <Up>. Your goal is to earn points by maneuvering the snake's head into as many red blocks as you can. Each time the snake "eats" a red block, you get one point. However, at the same time, the snake's tail gets longer and more bothersome. At any time, you can pause the game by pressing **p**, and continue by pressing **p** again.

The game is over when the snake runs into its own tail or runs into one of the walls. At that point, the program will change to a buffer named ***snake-scores*** to show you how many points you have accumulated. To play another game, use **C-x b** to change back to the ***Snakes*** buffer, and press **n** to start a new game.

At any time, you can quit by pressing **q**.

- PEG SOLITAIRE (to start: **M-x solitaire**)

This is a very old game for one person, played on a board with holes in a particular pattern. Each hole has a peg in it, except the central hole. To play the game, you jump one peg over another in a straight line, inserting the first peg into an empty hole, and removing the peg you jumped over. The goal is to empty the board, ending up with exactly one peg in the center hole.

There are two traditional patterns for peg solitaire boards: English and European. The Emacs program uses the English pattern with 33 holes in the shape of a cross:

```
    o o o
    o o o
o o o o o o o
o o o . o o o
o o o o o o o
    o o o
    o o o
```

In the diagram above, all the holes are filled with pegs except the center hole, which is empty. This is the starting layout of the game.

To play, use <Left>, <Right>, <Down>, and <Up> to move the cursor to the peg you want to use to make a jump. Then press <Enter> followed by the direction in which you want to jump. The peg you jump over will disappear. Remember, your goal is to get rid of all the pegs but one, which must end up in the center hole of the grid.

At any time, if you get stuck, press <Space> and the program will tell you how many possible moves there are. To reverse a move, press one of the Emacs undo keys: **C-x u**, **C-/** or **C-_** (see Section 7.3). To quit, press **q**.

- SPOOK (to start: **M-x spook**)

Spook was designed to insert special words — with no meaningful context — into your buffer. These words are of the type that the American FBI (Federal Bureau

of Investigation), CIA (Central Intelligence Agency), or NSA (National Security Agency) would be looking for within electronic messages in order to identify subversive people.

The intention is that, after composing a mail message, you execute the **spook** program to create a collection of these words to copy and paste to your outgoing message. If the FBI or CIA or NSA has a program that checks for subversive messages traveling around the Internet, your message will be snagged and archived. Later, some government flunky will have to waste time reading your mail, just to make sure you are not a bad guy.

Here is some sample output from **spook**:

```
Tuberculosis computer terrorism orthodox password
Black out NSWC Al-Qaeda AOL TOS Law enforcement
doctrine Nigeria KLM Plame Critical infrastructure
Suspicious substance
```

Of course, you don't know for sure if the FBI or CIA or NSA is really scanning your email. However, the idea is that if everyone uses Spook regularly, it would overwhelm such agencies, if they did indeed try to spy on us.

- TETRIS (to start: **M-x tetris**)

The Emacs version of the popular video game. It takes place in a buffer named ***Tetris***.

The program generates a series of geometric shapes called tetriminos. (The name is taken from the word "domino".) Each tetrimino consists of four small, square blocks stuck together. One tetrimino at a time falls slowly to the bottom of a vertical shaft. As the tetrimino falls, you can move it and rotate it. Your goal is to maneuver the tetrimino so that, when it lands at the bottom, it fits into the pattern created by the previous tetriminos (which have already fallen), without creating any empty space.

As a tetrimino falls, use <Left> and <Right> to move it sideways. Use <Up> to rotate it clockwise, and <Down> to rotate it counterclockwise. Once you get a tetrimino lined up, you can also press <Space> to force it to fall to the bottom immediately. (This speeds up the game.)

At any time, you can pause the game by pressing **p**, and continue by pressing **p** again. To quit, press **q**.

When the game ends, the program will change to a buffer named ***tetris-scores*** to show you how many points you have accumulated. To play another game, use **C-x b** to change back to the ***Tetris*** buffer, and press **n** to start a new game.

Section 12.8: Zippy the Pinhead Talks to the Emacs Psychotherapist

In this section, I'm going to show you something that used to be included with Emacs, but vanished in 2006. However, it is so interesting I thought you would like to see it — and it's a good way to end the book.

The 1970s in the United States was a time of great turmoil as the social progressive values that emerged in the late 1960s matured to take their rightful place within the popular culture: hippies, free love, drugs, anti-war protests, disco music, and Zippy the Pinhead.

Zippy the Pinhead was a creation of the American cartoonist Bill Griffith. (At the time, "pinhead" was a pejorative term for a cretin: a person with serious deformity and mental retardation, caused by a thyroid deficiency.) Zippy dressed in a baggy yellow clown suit with large red polka dots, and wore white puffy clown shoes. His point-like head was bald, except for a small sprig of hair tied with a red bow.

Zippy the Pinhead was a freakish, counterculture comic character, whose speech was characterized by repeated series of non-sequiturs. For a long time, Emacs had a program called **`yow`** whose purpose was to display a random quotation from Zippy the Pinhead.[2] To show you what I mean, here are three examples of what you might have seen if you had run the Emacs command **`M-x yow`** before 2006:

```
Equip me with MISSILES!!
*Help*I'm RELIGIOUS!!
I love a man with a HAIRPIECE!!
```

```
I don't know why but, suddenly, I want to
discuss declining I.Q. LEVELS with a blue
ribbon SENATE SUBCOMMITTEE!
```

```
My pants just went on a wild rampage through a
Long Island Bowling Alley!!
```

In Section 12.7, I described another very old Emacs program named Doctor (**`doctor`**). Doctor, which is still part of Emacs, acts like a nonjudgmental, infinitely patient, Rogerian psychologist using person-centered therapy. ("Tell me something about your family.")

In 2006, unfortunately, the Zippy quotations were removed for copyright reasons. However, up until that time, one of the most amusing features of Emacs was a command you could use to feed random Zippy quotations to the Doctor program. This would let you watch what happens when a nonjudgmental, infinitely patient, psychotherapist encounters a verbally iconoclastic pinhead. Because the Zippy quotations were random, the conversation was different every time. The command to use was:

```
M-x psychoanalyze-pinhead
```

[2] I have been able to trace it as far back as November 5, 1986.

Unfortunately, this command lives only in our memory so, to end the book, I now present you with Figure 12-9: excerpts from an authentic conversation that once took place between Zippy the Pinhead and the Emacs **doctor** program.

In order to make the conversation easy to follow, I have indented Zippy's comments and put them in italics.

```
I am the psychotherapist.
Please, describe your problems.
Each time you are finished talking, type RET twice.
```

If I am elected no one will ever have to do their laundry again!

```
Why do you say that?
```

Yow! I want my nose in lights!

```
What makes you believe you want your nose in lights?
```

I selected E5... but I didn't hear "Sam the Sham and the Pharaohs"!

```
Is it because you didn't hear sam the sham and the
pharaohs that you came to me?
```

Spreading peanut butter reminds me of opera!!
I wonder why?

```
Can you elaborate on that?
```

Civilization is fun! Anyway, it keeps me busy!!

```
Earlier you said you didn't hear sam the sham and
the pharaohs"?
```

I'm EMOTIONAL now because I have
MERCHANDISING CLOUT!!

```
When did you first know that you have
merchandising clout?
```

I have seen these EGG EXTENDERS in my Supermarket...
...I have read the INSTRUCTIONS...

```
You mentioned that it keeps you busy?
```

All right, you degenerates!
I want this place evacuated in 20 seconds!

FIGURE 12-9. Zippy the Pinhead talks to the Therapist.

Section 12.9: A Personal Note From Harley Hahn

To end the book, I want to ask you a question: What are the best things in life?

At the very beginning of the book, I described four characteristics about the type of people who like to use Emacs. As we end this book together, let me take a moment to recast these observations to describe *you*: the person who is reading this right now:

1. You are smart.
2. You like to use computers.
3. You enjoy teaching yourself how to use complicated, powerful tools.
4. You are able to learn on your own by reading and practicing.

Assuming this does describe you, my reader, let me repeat the question: What are the best things in life?

To be sure, we all have our own answers to this question. However, there is one important need that you and I have in common: a need that — when it is met — goes a long way to making us content. We want to work and think at the natural speed of our mind. This is why we become uncomfortable and frustrated when our thought processes are slowed down unnecessarily. When we have a problem to solve or an idea to explore, we want to be able to do it in a way that make sense to us and is enjoyable.

As such, we want computer tools that are powerful enough for the type of work and the type of thinking we enjoy. Tools that were created *by* smart people *for* smart people. Tools that will shape our minds in a *good* way as we use them. Tools we will never outgrow, that we can modify and extend and — when we feel so inclined — share with others. Finally, we want tools that are supported by a larger system that itself has these characteristics, for example, Linux and other types of Unix.

Can you see that I have just described Emacs?

No one is saying that Emacs is the only computer-based set of tools that serves people like us in that way. However, it is one of the few that has met the test of time, and it is one of the best.

True, Emacs is difficult to learn. In fact, it is very difficult to learn, which is why I spent a lot of time in the first part of this book making sure you had a strong foundation, before I even introduced you to your first Emacs commands.

Yes, it is true that Emacs is hard to learn, but it is just as true that, once you learn it, it is easy to use. And more important, Emacs is one of the few computer-based systems that will let you think and create at the natural speed of your own mind.

This is why I wrote this book: to help people like you learn how to use Emacs. As such, I thank you for spending so much time reading and thinking about what I have had to say. I am grateful for the opportunity to teach you, and to help you change your life.

Harley Hahn

APPENDIX A

Personal Notes

Personal Note #1: Teaching Yourself Emacs

From Section 1.1:

"Out of the 7.28 billion other people in the world, there is nobody who is ever going to teach you Emacs in person. You will have to teach it to yourself by reading and practicing."

Comment:

It wasn't always this way. In the 1970s, systems like Unix (the grandfather of Linux) and Emacs existed within a rich oral tradition among computer science students, professors, and researchers. At the time, computers were so expensive that they had to be shared, and programmers would use a terminal (a medium-sized box with a keyboard and screen) connected by a cable or a phone line to a remote computer. So when Emacs was first developed, programmers and researchers often worked in rooms with shared terminals and — believe it or not — computer programming was often a social activity. Indeed, it was common to see two programmers collaborating, sometimes even sitting at the same desk in front of the same terminal working together. In those days, it was part of the prevailing culture that if you had a problem and you had tried your best to solve it, you could ask anyone in the terminal room (or down the hall) for assistance, and they would be glad to teach you, help you, and even spend time talking about your ideas.

In the 1980s, as inexpensive, personal computers became available, programming increasingly became an individual activity. This tendency has increased to the point that, today, using a computer is very much a solitary activity, even though we are all "connected" via the Internet and mobile phones. On the other hand, because it is now easy for people all over the world to collaborate on large projects, we have a lot of high-quality, free, open source software, and many wonderful free tools.

Still, if you ask your grandparents how it was to use computers back in the 1960s and 1970s, they will tell you that it was a lot more fun than it is now because, much of the time, smart people spent their time working in person with other smart people, and they would all teach and help one another.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0

All of which gets back to my original point. Those days are gone and, today, no one is going to spend time, in person, teaching you how to use Emacs, so you had better be the type of person who likes to teach yourself. Of course, you do have this book, so you aren't completely alone.

Personal Note #2: Computer With a Keyboard

From Section 1.1:

"It will help a lot if you already have some experience using a computer with a keyboard."

Comment:

If you have only ever used a phone or a tablet, I do want to encourage you to learn Emacs. Just push ahead with this book, and I will help you. However, as I mentioned, you will need a computer with a keyboard, and you will have to learn how to use it. This means you will have to learn how to type and how to use all the special keys.

If you are not used to it, using a keyboard may seem a bit awkward at first. However, by the time you get half-way through this book, working with Emacs on a real computer will have permanently changed your brain cells for the better.

Personal Note #3: Usenet, Emacs, and the Growth of the Internet

From Section 1.1:

"For over 25 years, Usenet was used by a very large number of people around the world to post messages discussing every topic you can imagine, as well as to develop and share software. Many users used Emacs to create their messages, and a related program (**`gnus`**) to participate in Usenet discussion groups. Eventually, however, both discussions and software distribution migrated to the Web, which had no need of an external text editor."

Comment:

USENET is a vast, global system of discussion and file sharing groups. It was created in 1979 by two graduate students at Duke University, Jim Ellis and Tom Truscott, as a tool to send news and announcements between two universities in North Carolina (University of North Carolina and Duke University). Within a short time, the system had spread to other schools, and it soon developed into a very large, very popular system of discussion groups. For historical reasons, it became common to refer to Usenet as THE NEWS and to the messages in the discussion groups as ARTICLES.

Usenet, like most Internet-based services, is a client/server system. Data is stored on a large number of NEWS SERVERS; to access a server, you use a Usenet client program, called a NEWSREADER. In the 1980s, one of the more popular Usenet newsreaders was **`gnus`**, which was integrated with Emacs. (Both **`gnus`** and Emacs were created by the Free Software Foundation.)

Usenet had no central authority, so there was no one to manage the system or to make any rules (and even if there were rules, there would be no way to enforce them). For a long time, Usenet functioned well because it was put together in a clever way, and because there was a lot of cooperation among the people who managed the news servers.

Indeed, when the first primitive ancestor of the Web came along in 1991, Usenet was a robust, worldwide communication system, with many tens of thousands of discussion groups. In fact, without Usenet, it would have been impossible to develop the software and technical protocols that created the modern Internet. (Read that last sentence again.)

Eventually, however, Web-based technology became much easier to use and more powerful than Usenet. As a result, both discussion groups and software distribution migrated to the Web. Since the early 2000s, the need for the old Usenet discussions groups diminished significantly, and Usenet evolved into a large, worldwide file-sharing system.

For more information about Usenet and how to use it, visit my special Web site, *Harley Hahn's Usenet Center*:

`http://www.harley.com/usenet/`

Personal Note #4: Free/Open Source Software

From Section 1.5:

"Free software refers to programs that are distributed with a license specifying that anyone in the world is allowed to read the source code, modify the code, and share the results of their work freely. Another name commonly used for free software is open source software."

Comment:

The free software movement and the open source movement come from somewhat different motivations. As such, there are some people who will argue that, technically, "open source" is not the same as "free". My answer is: not in this book.

For practical purposes, "free software" and "open source software" refer to the same thing. Specifically, both Emacs and Linux are free/open source software (see Section 2.2).

Personal Note #5: GNU's Not Unix?

From Section 1.5:

"GNU is the name Richard Stallman chose to describe the Free Software Foundation's project to develop Unix-like tools and programs. The name GNU is an acronym for GNU's Not Unix".

Comment:

Why would Richard Stallman want to name such a large project in a negative way ("GNU's not Unix")? And why, if his goal was to create a complete, free version of Unix, would he want to tell everyone that it was not Unix? Although the name seems strange, it actually made sense in 1985.

Unix was developed in the early 1970s at Bell Labs, a New Jersey research facility that, at the time, was part of AT&T (see Section 2.2). At first, Unix was shared freely with other computer scientists and programmers. However, in 1983, AT&T decided it was time to make money from this project and changed their policy. They packaged Unix as a commercial product called UNIX System V (pronounced "System Five") and announced that the days of free Unix were over. From now on, AT&T Unix would cost money.

Moreover, they also dictated exactly how the name "Unix" should be used. Specifically, the name must be spelled in capital letters (UNIX) and could be used only as an adjective, not a noun. In other words, according to AT&T's lawyers, if you wanted to talk about Unix you must refer to it as the "UNIX Operating System", not Unix (or even UNIX). Much more important: if you wanted to use it, you now had to pay for it.

You can imagine how programmers and computer scientists felt about this ukase. On the one hand, the naming convention was ludicrous and mostly ignored. (Many people who worked with non-System V Unix started to talk about "Unix-like operating systems"; other people referred to such systems as "U*ix": two terms that you will still see today.)

On the other hand, the fees to use Unix as a commercial product were expensive, real, and highly resented; but there was no way around them: AT&T did own the operating system. Even worse, the freedom to examine and change the AT&T UNIX source code was now restricted.

The solution, of course, was to develop new versions of Unix that worked exactly like AT&T's UNIX, without trespassing on their copyrights and trademarks. To be sure, people were already hard at work, developing free versions of Unix: most notably, the BSD project at U.C. Berkeley. ("BSD" stands for Berkeley Software Distribution.) However, none of these free versions of Unix could run on personal computers, which meant they were out of reach for most of the programmers in the world.

This is why, in 1985, Richard Stallman founded the Free Software Foundation with the ambitious goal of creating a completely free version of Unix, called GNU. (You can read more about this in Section 2.2.)

But why the name GNU?

At the time, Stallman — and many, many other programmers around the world — were very uncomfortable with what AT&T (and their lawyers) had done. As a result, Stallman wanted to make it clear that what the Free Software Foundation was going to create was not going to be based, in any way, on AT&T's UNIX. Thus the name GNU: an acronym for "GNU's Not Unix".

As you can see below, within the expression "GNU's Not Unix", the word GNU can be expanded indefinitely:

```
     GNU
    (GNU's Not Unix)
   ((GNU's Not Unix) Not Unix)
  (((GNU's Not Unix) Not Unix) Not Unix)
((((GNU's Not Unix) Not Unix) Not Unix) Not Unix)
```

and so on. Thus, GNU is actually a recursive acronym, a wry joke that demonstrates the sort of whimsy and irony that appeals to computer scientists.

When you expand the word GNU in this way — with each set of parentheses enclosing a list of items — you create the type of structure that is used when you program with Lisp, a language that was popular among artificial intelligence people at the time. Stallman used Lisp when he worked in the MIT AI Lab, and, in fact, Emacs itself was written in Lisp and comes with an entire Lisp programming environment called Emacs Lisp.

Lisp was created, in 1958, by John McCarthy, a computer scientist at MIT, as a mathematical notation for computer programs. As such, it is the second oldest programming language still in widespread use around the world. The name Lisp stands for "List Processor".

In case you are wondering, the oldest programming language still in use today is Fortran, which stands for "formula translation". Fortran was designed at IBM in 1954 and was implemented in 1957. (Fortran was the first programming language I ever used. This was in 1970, during my last year of high school.)

Personal Note #6: Our Tools Shape Our Minds

From Section 2.1:

"Whether you realize it or not, one of the most important choices you make in your life is which operating system you will be using."

Comment:

Imagine how a human brain develops when, from a young age, the only computer it is ever allowed to use is a touch-screen-based phone or tablet running iOS or an Android operating system. To me, this qualifies as a type of child abuse.

Personal Note #7: AT&T

From Section 2.2:

"The first primitive version of Unix was created in 1969 by Ken Thompson, a programmer at the Bell Labs research facility in New Jersey, owned by AT&T."

Comment:

When you read about the history of Unix, the C programming language, and so on, you should know that the AT&T where all this was created has absolutely nothing to do with the companies that, today, use the name AT&T.

The AT&T that, in the 1970s, owned Bell Labs is long gone. In 1984, as part of an enormous antitrust settlement, the company was broken up into eight large pieces, one of which kept the name AT&T. These pieces were sold, recombined, enlarged, diminished, and re-sold, again and again.

Although the original AT&T hasn't existed for decades, the brand is so valuable the name "AT&T" has been preserved, very carefully, for commercial purposes. In fact, I know someone who has an "AT&T" credit card, even though this particular credit card has actually been owned by Citibank since 1998.

If this sort of thing intrigues you, search online for "AT&T Brand Licensing Opportunities".

Personal Note #8: Early Unix on the West Coast

From Section 2.2:

"In 1974, the use of Unix began to spread to the West Coast [of the United States]. The result was a new Unix, called BSD (an acronym for Berkeley System Distribution), which became popular among computer scientists and programmers around the world."

Comment:

I myself first used Unix in 1976, as a graduate student at U.C. San Diego.

Personal Note #9: BSD Unix in the 1980s

From Section 2.2:

"Most of the BSD-based versions of Unix, on the other hand, were non-commercial and were distributed for free."

Comment:

The one main exception was a BSD-based version of Unix named SunOS that was created and sold by Sun Microsystems, a company co-founded in 1982 by Bill Joy, from U.C. Berkeley, and three graduates of Stanford University. Eventually, Sun merged SunOS with System V to create a new version of Unix they named Solaris. In 2010, Sun was bought by Oracle, the company that now owns Solaris, which is still a commercial product.

Personal Note #10: Hackers and Geeks

From Section 2.2:

"There was still no operating system that ran on a PC that was attractive to the type of programmer whose idea of fun was to take things apart and modify them."

Comment:

Such programmers were called "hackers". Over the years, this term has been used so often to refer to troublemakers that, although it is still used in programming circles, you will often see the term "geeks" used instead.

There was a time when such people were called "nerds", but that was before *The Big Bang Theory* TV program made the term nauseatingly mainstream.

Personal Note #11: Bash

From Section 2.2:

"The most popular shell is called Bash."

Comment:

The very first Unix shell, created in 1977 at Bell Labs, was named **`sh`**. In later years, this shell became known as the Bourne Shell, named after its creator Stephen Bourne. Bash was created by the programmer Brian Fox to be a replacement for the aging Bourne Shell, and was released in 1989 as part of the GNU Project.

The name Bash stands for "Bourne-again shell".

Personal Note #12: Linux Is Free

From Section 2.2:

"Linux is free, open source software."

Comment:

As I was researching this part of the book, I found some old articles from *InfoWorld* magazine which was, at the time Linux was being developed, a well-known weekly magazine for personal computing professionals. In 1990, Apple had their own version of Unix, called A/UX, which they sold as a commercial product to people who wanted to run Unix on an Apple computer. On page 40 of the May 28, 1990, issue of the *InfoWorld*, I came across the following short article:

"Apple's A/UX 2.0 will begin shipping in June. The product will cost $795 in a CD ROM version, or $995 for a tape or floppy disk version."

Let's use the second number to do a quick calculation. (I'm going to ignore the first number because, in 1990, most people didn't have CD drives.) At the time, the sales tax in California (where Apple was located) was about 6.75%. Thus, in June of 1990, the total sales price for this particular version of Unix was $1,062. To put this in perspective, $1,062 in 1990 was worth about $2,000 in 2016 dollars. Imagine living in a world where it cost $2,000 for a license for one personal computer to run a version of Unix that, by today's standards, would be horribly sub-standard.

Aren't you glad we have the Free Software Foundation and the Linux Project?

Personal Note #13: Mac OS X Is Unix

From Section 2.3:

"Almost everyone who uses Emacs does so under some type of Unix. Specifically, GNU Emacs is available in specific versions for Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X, and Solaris."

Comment:

Apple's OS X operating system is Unix, because it meets the standards for being Unix and is certified as such. It is not, however, Linux, because it doesn't use the Linux kernel. Nor does it use the Free Software Foundation's GNU utilities. (For a brief discussion of kernels and utilities, see Section 2.2.)

Mac OS X uses a kernel named XNU, whose original ancestors were the Mach kernel from Carnegie-Mellon University and the BSD4.3 kernel from U.C. Berkeley. In addition, the OS X utilities are based on BSD, not the GNU utilities. What XNU does have in common with Linux and the various BSD kernels is that it has been released as free, open source software (as part of a project called Darwin).

The name XNU is an acronym for "XNU is not Unix" which is a misnomer, because OS X really is Unix.

Historical note: In the 1980s, a similar acronym, "Xinu is not Unix", was used for the name Xinu, an operating system developed as a teaching aid by Doug Comer at Purdue University. However, unlike OS X, Xinu really wasn't Unix. It was, however, a very cool operating system that, in its time, was quite popular and is still in use today.

Personal Note #14: Terminals That Print

From Section 2.3:

"Unix was developed in the 1970s, before programmers were able to have their own personal computers. To work with Unix, a person would use a terminal connected to a host. The terminal was an electronic box with a keyboard and a screen, or a keyboard and a printer."

Comment:

The oldest computer terminals produced output by printing on a continuous roll of paper. I used a number of such terminals as a student. The ones that I remember the best are the IBM 2740 and 2741 terminals based on the venerable IBM Selectric Typewriter, and the smaller Texas Instruments Silent 700 series portable terminal that printed on special thermal paper.

Before terminals with screens became cheap and ubiquitous, many people used Unix with printing terminals. In fact, the PDP-11 computer on which Unix was first developed at Bell Labs had printing terminals attached to it. (If you look at the photo in Figure 2-2, you can see these terminals.)

To this day, traditional Unix terminology uses the term "print" to refer to displaying output. For example, the Unix command to display the name of your working directory (the folder you are using at the moment) is **`pwd`**, which stands for "print working directory".

Personal Note #15: Why U.C. San Diego in 1976?

From Section 2.3:

"Let's pretend for a moment that we are going to take a time-travel trip back to the late 1970s. Close your eyes and pretend that it is 1976, and you and I are visiting the computer science department at U.C. San Diego."

Comment:

At the time, the two foci of Unix development were Bell Labs in New Jersey, and the computer science department at U.C. Berkeley, California.

However, in 1976, I was a grad student in San Diego and, believe me, the climate was a lot better there than in either New Jersey or Berkeley. So as long as we are going back in time, we might as well have sunshine and warmth (which is why I went to grad school in San Diego in the first place).

By the way, at U.C. San Diego in 1976, computer science was actually taught within the Department of Applied Physics and Information Science. The current Department of Computer Science and Engineering was not established until 1987.

Personal Note #16: 80- and 132-character Lines

From Section 2.4:

"The DEC VT52 terminal (introduced in July 1974) had 24 rows, each of which could display 80 characters. A few years later, the VT52 was replaced by a more powerful terminal, the DEC VT100 (August 1978), which displayed either 24 rows of 80 characters or 14 rows of 132 characters."

Comment:

With respect to character (text) output, you will find that the numbers 80 and 132 come up a lot. Here is why.

80 characters/line: In 1928, IBM introduced PUNCH CARDS that could hold 80 characters per card, which became the standard. When computers were first used commercially in the early 1960s, input data was stored on punch cards that used the same 80 characters/card form factor. These cards were created using a machine called a KEYPUNCH. The most widely used keypunches were the IBM 026 and IBM 029. Punch cards were also called COMPUTER CARDS and IBM CARDS.

Because everyone was used to 80 characters/card, the first popular video display terminals were designed to have 80-character lines, and that size became a standard for terminals, one that you will still see today with terminal emulators.

132 character/line: In 1959, IBM introduced the IBM 1403 printer, probably the best printer (for its time) in history. The most popular IBM 1403, which was extremely fast, printed 132 characters/line which became an industry standard for output.

The IBM 1403 printed on special continuous, folded and perforated paper, called COMPUTER PAPER that came in boxes. The top end of the paper would be pulled

out of the box and fed into the printer. The printed paper would come out of the back of the printer as one long accordion-like PRINTOUT.

When I was a university student, the IBM 026 keypunches were just being phased out. I did use them for a bit, but I had a lot more experience with the IBM 029 keypunch, punch cards, and the 1403 printer.

Personal Note #17: Unix Workstations

From Section 2.5:

"By the end of the 1980s, many people were using Unix on inexpensive personal computers. Other people, who needed more power and larger monitors, used expensive, high-end Unix machines called workstations."

Comment:

The most popular Unix workstations were made by Sun Microsystems, Silicon Graphics, Apollo, DEC, HP, and IBM, and were not IBM PC compatible. During the 1990s, these expensive, special-purpose Unix computers began to be replaced by new generic IBM-compatible PCs that, every year, were becoming more powerful and less expensive.

As an example, in 1991, Linus Torvalds created the first Linux kernel using a generic, 386-based PC. (Trivia: Linus bought that computer on January 5, 1991.)

Personal Note #18: Time Travel

From Section 2.4:

"There are many different terminal emulators and, believe it or not, they are all based on the old DEC VT100 family of terminals we discussed in Section 2.4. This means that, even today, when you type a Unix command or run a text-based Unix program, you are using a technology that was first introduced in 1978."

Comment:

This suggests an interesting hypothetical question. Let's say you were to install the latest version of Linux on your computer. You then invent a time machine, use it to go back to 1978 and bring back an actual DEC VT100 terminal. Here is the question: If you were able to find a way to connect the VT100 to your Linux computer, would it work?

The answer is yes; it would work. Unix would have no problem communicating with a real VT100.

Of course you would have to find a way to connect a VT100 from 1978 to a modern computer, but if you are the type of person who can figure out how to make a time machine, connecting an old terminal to your computer shouldn't be much of a problem.

Personal Note #19: Midnight Commander

From Section 2.5:

"Emacs is a TUI-based program. In fact, it is one of the most powerful TUI-based programs ever written."

Comment:

If you want to see a wonderful example of another TUI-based program, I recommend that you install a file manager named "Midnight Commander". The name of the Linux version of this program is **mc**.

Midnight Commander is a powerful, easy-to-use program (once you get used to it) that enables you to manage files and directories using only your keyboard. If you want to see what a well-designed TUI-based program can do, Midnight Commander is a lot easier to learn than Emacs, and very useful in its own right.

The **mc** program is a clone of a very old TUI-based program called "Norton Commander" written by John Socha and named after Peter Norton. It was originally released in 1986 to run under DOS on PCs, but Socha's TUI-based design was so good that it is still being used, virtually unchanged, decades later on other systems, including Linux.

John Socha was a good writer and a great programmer. In 1984, he wrote an assembly language book that, in 1987, was republished as the Peter Norton assembly language book.

Believe me, assembly language books are not easy to write. I wrote my first one in 1978 (for the IBM System/360). In 1986, I wrote *The Complete Guide to IBM PC AT Assembly Language*, which was republished in 1992 as *Assembler: Inside and Out*.

At the time, we all (Peter, John, and I) had the same literary agent. Like John, when I was building my reputation, I wrote a fair bit under the Peter Norton brand name. In 1991 I wrote *Peter Norton's Guide to Unix* (my second Unix book) and, later, contributed to five other Peter Norton books. By 1995, I became accomplished enough to publish books with my own name in the title, the first one being *Harley Hahn's Internet Yellow Pages, 3rd Edition*.

I only met Peter Norton once, the day we had our photo taken together for the cover of the Unix book. I don't remember ever meeting John, but I would have liked to.

Personal Note #20: KDE and Gnome

From Section 2.5:

"With Linux, the most popular GUIs are Unity, Gnome and KDE."

Comment:

The oldest of these GUIs is KDE, first released in 1998. KDE was well-received in the Linux community, but it was based on the Qt Toolkit which, at the time, was not totally free software. This was a concern for some of the Free Software Foundation (FSF) programmers, who started a project to create their own, totally free GUI, called Gnome. The first version of Gnome was released in 1999. By 2000, the licensing

terms for Qt had been changed so that even the FSF was satisfied, making KDE completely free.

The newest of the three GUIs is Unity, which is actually based on Gnome. Unity was first released in 2011. (For a discussion of free software and the FSF, see Section 1.5.)

Personal Note #21: Aren't All Terminals Virtual?

From Section 2.6:

"[With Linux] there are two different types of terminal emulators: virtual terminals and terminal windows. When you use a virtual terminal, the terminal emulator uses your entire screen to provide you with a totally text-based experience. When you use a terminal window, the terminal emulator runs in a window within a GUI (graphical user interface)."

Comment:

A terminal window gets its name, obviously, because it is a terminal emulator running inside a window. But what about the "virtual terminal"? What does that name mean?

In science, the term VIRTUAL refers to something that doesn't really exist. For example, when you look at an object in the mirror, what you see is a virtual image. Similarly, when a computer uses disk space to simulate memory that isn't really there, we call it virtual memory. Now you can see that the name "virtual terminal" indicates that the terminal isn't real: it's a simulation created by a program. (More formally, we say that the program emulates a terminal.)

This raises a question. As we discussed in Section 2.5, no one actually uses real terminals anymore: we use only terminal emulators. And even though *all* terminals are virtual, we apply the name "virtual" to only one type of terminal emulator.

If you say it doesn't make sense, you are right. In fact, it is often the case that computer terms don't make sense, which is why we tend to teach them to people without an explanation. (Think about that for a moment.)

To help you understand how this works, I'm going to let you in on an important secret. All professionals need a way to talk about things they don't understand in a way that doesn't detract from their processional expertise.

For this reason, when computer experts need to talk about something that doesn't make sense, something we can't really explain, we often justify it without letting on that we don't understand it. This enables us to sound like we know what we are talking about even when we don't.

Example: "For historical reasons, virtual terminals are often referred to as virtual consoles."

This is a very important part of the computer world, which I first learned in high school. (In the words of my teacher: "Computers are a snow job.")

Compare this, say, to the world of medicine. In medicine, when we don't understand something, we still need to sound like we know what we are talking about. However, human bodies are a lot more complex and problematic than

computer programs, so we can't simply appeal to tradition, such as "historical reasons". Instead, we make up something that seems plausible, and we say it with authority. (Medical professionals do this all the time.)

Of course, I learned how to do this as a medical student and, over the years, I became quite good at it. However, I had an advantage. When I entered medical school, I already had a graduate degree in computer science.

Personal Note #22: Ubuntu Terminal Emulators

From Section 2.6:

"The easiest way to start a terminal window is to click on the Terminal icon in the [Ubuntu] Launcher. If you don't see the Terminal icon in the Launcher, click the top icon on the Launcher to open the Dash and type **`terminal`**. You will see the Terminal icon, which you can click to start the program."

Comment:

The Ubuntu terminal emulator, called simply "Terminal", is actually Gnome Terminal, a terminal emulator for the Gnome desktop environment. As we discussed in Section 2.5, the default Ubuntu GUI, Unity, is based on Gnome. Gnome and Gnome terminal were developed by the Free Software Foundation, the same organization that developed Emacs.

You may also see two other terminals: Xterm and UXTerm. Xterm is a terminal emulator that comes with X Window. (We discussed X Window in Section 2.4.) UXTerm is a version of XTerm that supports UNICODE, a system for encoding characters for most of the languages of the world beyond the standard Roman alphabet. You probably won't need these two programs.

Personal Note #23: How to Access the Command Line With Mac OS X and Windows

From Section 2.7:

"In this section, I am going to show you how to use the Unix command line. What you are about to read will work for any type of Unix, including Linux and Mac OS X. If you use Windows, the basic ideas will be the same, but the details and the commands will be different."

Comment:

Mac OS X:

To access the command line with Mac OS X, you need to open a terminal window. Use Finder to open the Applications folder. Then open the Utilities subfolder and start the Terminal program.

To make it convenient to open a terminal window whenever you want, simply drag the Terminal program to the Dock.

Windows:

Windows doesn't use terminals. To access the Windows command line, you must open a Command Prompt window. With Windows 7, click the Start button. Select "All Programs", "Accessories", then click "Command Prompt". With Windows 8 or Windows 10, right-click the Start button, then click "Command Prompt".

While the Command Prompt window is open, there will be an icon in the Taskbar. To make it easy to open a Command Prompt window whenever you want, right-click this icon and select "Pin this program to taskbar" (Windows 7) or "Pin to taskbar" (Windows 8, Windows 10).

Personal Note #24: Freddy and the Men From Mars

From Section 2.15:

"You might create a directory named **`circus`** to hold a collection of photos from a trip to Centerboro, New York, where you saw Boomschmidt's Stupendous and Unexcelled Circus. If you have a lot of photos, you might organize them by using three subdirectories: **`friends`**, **`animals`**, and **`martians`**."

Comment:

Boomschmidt's Stupendous and Unexcelled Circus appears in several of the Freddy the Pig books. The specific references above are from the book *Freddy and the Men from Mars*. As an Emacs user, it will probably do you a lot of good to read all the Freddy books, so here is the basic information.

Between 1927 and 1958, the American writer Walter R. Brooks (1886-1958) wrote 26 children's books about Freddy the Pig. Freddy lives on a farm owned by Mr. and Mrs. Bean, near the town of Centerboro in upstate New York, along with many other animals. All the animals can talk, but Freddy is, by far, the most talented. In the course of the books he becomes a detective, a poet, a newspaper publisher, a banker, a pilot, a football player, a politician, a magician, a baseball coach, and much more. *Freddy and the Men from Mars* was published in 1954.

I have a large Freddy collection and have read each book many times. In fact, much of what I know about human nature, I learned from Freddy and his friends. For more information, see the Freddy the Pig page on my Web site:

`http://www.harley.com/freddy-the-pig/`

Personal Note #25: Special Files and Proc Files

From Section 2.15:

"Pseudo files provide services to programs using the same methods that are normally used to read and write data from ordinary files. The three most important types of pseudo files are special files, named pipes, and proc files. Special files (also called device files) represent physical or emulated devices. Named pipes connect the output of one program to the input of another. Proc files provide technical information about the system itself."

Comment:

You can use special files to read or write using a physical device, for example, a display or printer (for output); a keyboard or mouse (for input); or a disk partition (input and output). You can also read and write using special files that emulate a physical device, such as a terminal.

Finally, there are special files that create virtual devices that provide esoteric services. For example, a program can obtain random numbers by reading from the special file **`/dev/random`**. Another example: there is a special file called **`/dev/null`** that discards anything you write to it. In fact, this is how Unix programmers get rid of a stream of data they don't want: they redirect it to **`/dev/null`**.

Proc files provide technical information directly from the kernel, the central part of the operating system (see Section 2.2). For instance, you can obtain information about your computer's memory (RAM) by reading from the file **`/proc/meminfo`**. To show you how it works, the following command uses the **`less`** program (see Section 2.14) to read from the file **`/proc/meminfo`** and to display the data one screenful at a time. Try it, and see what you get.

```
less /proc/meminfo
```

Here is a similar command that uses the **`cat`** program (used to display very short files) to read from the file **`/proc/sys/kernel/hostname`**. Reading this file accesses the kernel to obtain the name of the computer you are using:

```
cat /proc/sys/kernel/hostname
```

A program can read from this file to find out the name of the computer on which the program is running. Finally, here is a command that displays information about the processors (CPUs) in your computer:

```
less /proc/cpuinfo
```

Personal Note #26: How Many Files Are on Your Unix System?

From Section 2.16:

"A typical Unix system contains well over 300,000 files."

Comment:

To count the number of files and directories on a Unix system, use the following command:

```
sudo ls -R / | wc -l
```

We can take this command apart as follows:

- **`sudo`** runs the command as superuser, so you will have permission to count all the files on the system, not just your own. (You will need the superuser/administrator password to run this command.)
- **`ls -R /`** lists the name of every file and directory in the system, one name per line.

- **|** (the pipe symbol) sends the output of **ls** to the **wc** program.
- **wc -l** counts the number of lines.

I ran this command on a newly installed Linux system (Ubuntu Desktop Linux 14.04.3 LTS) without any extra software installed. It took 24 seconds to find that there were 340,615 files. When you have a few moments, try this command on your own computer.

If you have a lot of time and you want to see the names of all these files, you can pipe (send) the output of the **ls** command to the **less** pager (Section 2.14), which will display the output one screenful at a time:

```
sudo ls -R / | less
```

If you view one screenful every half-second, it will take you 47 hours and 19 minutes to look at all the names. If you get bored, you can stop **less** by pressing **q** to quit.

Alternatively, if you want to save the output to a file and print it out, you could have the pages bound into books with a nice cover. At 60 lines/page, you would have 5,637 pages. If you were to bind 200 pages/book, you would end up with 29 volumes, which would make a wonderful Mother's Day gift.

The point is, Unix systems come with a *lot* of files. (And remember, we are only taking about the *names* of the files, not the actual contents.)

Personal Note #27: Comparing Unix Packages to Commercial Apps

From Section 3.1:

"A package manager automates the various processes required to install, upgrade, configure, and uninstall software. All you have to do is tell the package manager what you want, and it does all the work."

Comment:

If the description of how a package management system functions sounds familiar, it is because package managers are also used to download consumer apps from app stores. This is the case with mobile devices, such as phones and tablets, and with personal computers running Mac OS X and some versions of Windows.

The Unix package managers differ from app package managers in several ways. The biggest difference is that the consumer app market is very tightly controlled by the companies that sell the operating system. They run the apps stores as they see fit, in their own interests, and most of the software (which is not open source) is designed for the money-driven consumer market. This is why commercial apps are secret, inflexible programs that either cost money, force you to look at advertisements, try to sell you something, capture and use your personal data, or all of the above.

The package management systems used with Linux and BSD systems are open source and non-commercial. As such, they are designed for the benefit of users. Generally speaking, the software you download is free, does not show you ads, does not try to sell you something, and is designed to respect your privacy. (All of which, by the way, is true for Emacs.)

APPENDIX B

Command Summaries

Throughout the book, there are 60 figures that have summaries of various Emacs or Unix/Linux commands. For reference, I have collected them all in this appendix.

To make it easy for you to find what you want, here is a list of the summaries in the same order that you will find them in the book. All you have to do is scan the list, find what you want, and look below for the actual command summary. In case you need more information, I have included the section number in which each figure appears, so you can read about the summary in its original context.

- Accessing a virtual terminal from the GUI. (Figure 2-4)
- Changing from one virtual terminal to another. (Figure 2-5)
- Keys to make corrections when typing a command. (Figure 2-7)
- Key combinations to use when typing a command. (Figure 2-8)
- Commands to use with less. (Figure 2-9)
- Important directories in filesystem hierarchy standard. (Figure 2-10)
- The most important file commands. (Figure 2-11)
- The most important directory commands. (Figure 2-12)
- Bash configuration files. (Figure 2-13)
- Linux package management systems. (Figure 3-1)
- BSD package management systems. (Figure 3-2)
- Emacs names for special keys. (Figure 4-1)
- Choosing to save files after stopping Emacs with **C-x C-c**. (Figure 5-1)
- Status characters within the mode line. (Figure 6-5)
- Completion commands. (Figure 6-6)
- Choosing whether or not to run a disabled command. (Figure 6-7)
- Keys to use while typing. (Figure 7-1)
- Commands for controlling windows. (Figure 7-2)
- Commands for controlling buffers. (Figure 7-3)
- Commands for working with files. (Figure 7-4)
- Commands for moving the cursor. (Figure 8-1)
- Commands for moving cursor through a paragraph/sentence. (Figure 8-2)
- Major modes to use when editing a human language. (Figure 8-3)
- Prefix argument combinations. (Figure 8-4)

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0

- Commands to move throughout the buffer. (Figure 8-5)
- Commands to use line numbers. (Figure 8-6)
- Commands to set mark and define a region. (Figure 8-8)
- Commands that act upon the region. (Figure 8-9)
- Commands to delete text. (Figure 9-1)
- Commands to kill text. (Figure 9-2)
- Commands to move and kill by word or sentence. (Figure 9-3)
- Commands to yank text. (Figure 9-4)
- Commands for correcting common typing mistakes. (Figure 9-5)
- Commands for correcting spelling mistakes. (Figure 9-6)
- Commands to fill text. (Figure 9-7)
- Search commands. (Figure 10-1)
- Keys to use during a search. (Figure 10-2)
- Non-incremental search commands. (Figure 10-3)
- Search commands. (Figure 10-4)
- Search commands for regular expressions. (Figure 10-5)
- Characters to use with regular expressions. (Figure 10-6)
- Search and replace commands. (Figure 10-7)
- Responses during a search and replace command. (Figure 10-8)
- Minimum keystrokes to invoke search/replace commands. (Figure 10-9)
- The four basic major modes. (Figure 11-2)
- Major modes: Fundamental mode family. (Figure 11-3)
- Major modes: Text mode family. (Figure 11-4)
- Major modes: Prog mode family. (Figure 11-5)
- Major modes: Special mode family. (Figure 11-6)
- Independent major modes. (Figure 11-7)
- Minor modes. (Figure 11-8)
- Commands to set and describe modes. (Figure 11-10)
- Running shell commands. (Figure 12-1)
- Help facility options. (Figure 12-2)
- General info commands. (Figure 12-3)
- Info commands to select a node. (Figure 12-4)
- Info commands to read a node. (Figure 12-5)
- Built-in tools. (Figure 12-6)
- Dired commands. (Figure 12-7)
- Games and diversions. (Figure 12-8)

Figure 2-4 (from Section 2.6). Accessing a Virtual Terminal From the GUI.

Terminal	Key Combination
1	<Ctrl-Alt-F1>
2	<Ctrl-Alt-F2>
3	<Ctrl-Alt-F3>
4	<Ctrl-Alt-F4>
5	<Ctrl-Alt-F5>
6	<Ctrl-Alt-F6>

To switch from the GUI to a specific virtual terminal, use <Ctrl-Alt-F1> through <Ctrl-Alt-F6>. To return to the GUI, use <Alt-F7> or <Ctrl-Alt-F7>.

Figure 2-5 (from Section 2.6). Changing From One Virtual Terminal to Another.

Terminal	Key Combination
Next	<Alt-Right>
Previous	<Alt-Left>
1	<Alt-F1> or <Ctrl-Alt-F1>
2	<Alt-F2> or <Ctrl-Alt-F2>
3	<Alt-F3> or <Ctrl-Alt-F3>
4	<Alt-F4> or <Ctrl-Alt-F4>
5	<Alt-F5> or <Ctrl-Alt-F5>
6	<Alt-F6> or <Ctrl-Alt-F6>
Return to GUI	<Alt-F7> or <Ctrl-Alt-F7>

To switch from one virtual terminal to another, you have several choices. <Alt-Right> changes to the terminal with the next highest number. <Alt-Left> changes to the terminal with the previous number. <Alt-F1> through <Alt-F6> changes to a specific terminal. Finally, <Alt-F7> will return you to the GUI (the Desktop Environment). Please note that, for convenience, the function key combinations can use either <Alt> or <Ctrl-Alt>.

Figure 2-7 (from Section 2.10). Keys to Make Corrections When Typing a Command.

Key	Function
<Left>	Move cursor left one position
<Right>	Move cursor right one position
<Backspace>	Delete character to the left of the cursor
<Delete>	Delete character at the cursor
<Ctrl-W>	Delete the previous word
<Ctrl-U>	Delete the entire line (on some systems <Ctrl-X>)

As you enter a command, here are the most important keys you can use to make corrections. These keys work with all Unix systems, including Mac OS X. The first four keys also work when you type commands with Microsoft Windows. However, <Ctrl-W> and <Ctrl-U> do not work with Windows, as they are Unix keys.

Figure 2-8 (from Section 2.12). Key Combinations to Use When Typing a Command.

Unix Keys	Function
<Up>	Display previous line in history list
<Down>	Display next line in history list
<Left>	Move cursor left one position
<Right>	Move cursor right one position
<Backspace>	Delete character to the left of the cursor
<Delete>	Delete character at the cursor
<Ctrl-W>	Delete the previous word
<Ctrl-U>	Delete the entire line (on some systems <Ctrl-X>)

Emacs Keys	Function
<Ctrl-P>	Display previous line in history list
<Ctrl-N>	Display next line in history list
<Ctrl-A>	Move cursor to beginning of the line
<Ctrl-E>	Move cursor to end of the line
<Ctrl-B>	Move cursor left (backward) one position
<Ctrl-F>	Move cursor right (forward) one position
<Alt-F>	Move cursor right (forward) one word
<Alt-B>	Move cursor left (backward) one word
<Ctrl-K>	Delete from cursor to end of the line
<Ctrl-A> <Ctrl-K>	Delete the entire line

The top list shows the keys we discussed in Section 2.10 that you can use to make corrections when you are typing a command. The bottom list shows some of the key combinations copied from Emacs as part of the GNU Readline facility. You can use the keys in both these lists to move within the current line and the history list, and to help you make changes.

Figure 2-9 (from Section 2.14). Commands to use with `less`.

Letters	Function
h	Display help information
q	Quit the program
g	Go to first line of text
G	Go to last line of text
<Space>	Move forward (down) one screenful
b	Move backward (up) one screenful
/pattern	Search forward for specified pattern
?*pattern*	Search backward for specified pattern
n	Repeat search in the same direction (next)
N	Repeat search in the opposite direction
!*command*	Run the specified shell command

Special Keys	Function
<Home>	Go to first line of text
<End>	Go to last line of text
<PageDown>	Move forward (down) one screenful
<PageUp>	Move backward (up) one screenful

*The default Unix pager program is named **less**. The purpose of **less** is to display text, one screenful at a time. While you are reading, there are many commands you can use. Here are the most important ones.*

Figure 2-10 (from Section 2.16). The Most Important Directories Within the Filesystem Hierarchy Standard.

Directory	Description
/	Root directory
/bin	Essential user commands (binaries)
/boot	Boot loader files
/dev	Device files (special files)
/etc	System configuration files
/home	User home directories
/lib	Essential shared libraries and kernel modules
/media	Mount point: removable media
/mnt	Mount point: temporarily mounted filesystems
/opt	Application programs ("optional" software)
/proc	Proc files
/root	Home directory for the root userid (superuser)
/run	Temporary data used by programs that are running
/sbin	System programs (binaries)
/srv	Data for system services
/tmp	Temporary files, not preserved between reboots
/usr	Sharable, read-only data
/var	Variable (changeable) system data

All Unix systems organize files and directories into a tree-structured filesystem using a single root directory. The Filesystem Hierarchy Standard is the basic plan followed by Linux systems. To see the details for your particular system, look at the **hier** *man page.*

Figure 2-11 (from Section 2.17). The most important file commands.

Command	Description
cat	Display a very short file
cat	Combine (catenate) multiple files
chmod	Modify (change) file permissions
cmp	Compare two files to see if they are the same
cp	Copy files
du	Display disk usage for files
file	Analyze file type
find	Search for files in directory tree, then process results
head	Display the beginning of a file
less	Display contents of file, one screenful at a time
ls	Display (list) information about files
ls -l	Display full information (long listing) about files
mv	Move files
mv	Rename files
od	Display contents of a binary file (octal dump)
pwd	Display name of current directory
rm	Delete (remove) files
tail	Display the end of a file
touch	When file does not exist: create brand new empty file
touch	When file exists: update access and modification times
whereis	Find files associated with a command

When you use Emacs there will be many times when you need to manipulate your files. Sometimes you can do it from within Emacs, but a lot of the time, it will be easier and faster to use the standard Unix file commands. These are the most important ones, and I recommend you learn how to use them all. (Notice that **cat**, **mv**, *and* **touch** *each perform two different functions.)*

Figure 2-12 (from Section 2.17). The most important directory commands.

Command	Description
cd	Change your current (working) directory
chmod	Modify (change) directory permissions
du	Display disk usage for directories
ls	Display (list) information about directories
ls -l	Display full information (long listing) about directories
mkdir	Create (make) a directory
mv	Move directories
mv	Rename directories
pwd	Display name of current directory
rmdir	Delete (remove) an empty directory
tree -d	Display a diagram of a directory tree

Unix uses a very large, tree-structured file system. Within that file system, starting from your home directory, you can build a tree structure of your own in a way that suits you. This will become more and more important, as you develop a facility with Emacs. Here are the commands you can use to build, maintain, and use your own personal directory tree. (Notice that **mv** *performs two different functions.)*

Figure 2-13 (from Section 2.18). Bash Configuration Files.

File	Description
.bash_profile	Login file
.bash_login	Login file
.profile	Login file (POSIX mode)
.bashrc	Environment file
.bash_logout	Logout file

Many programs use dotfiles to hold configuration information. These are the files used by Bash, the default shell on many Unix systems (including most types of Linux as well as Mac OS X). Once you learn how to use Emacs, I suggest that you take some time to learn about these files and customize them.

Figure 3-1 (from Section 3.2). Linux Package Management Systems.

Package Manager	Linux Family	Linux Distributions
APT	Debian-based	Debian, Mint, Ubuntu
RPM	Fedora-based	Fedora, Mageia, Manjaro, RHEL, CentOS
RPM	SUSE-based	OpenSUSE
Pacman	Arch-based	Arch
Portage	Gentoo-based	Gentoo
`pgktool`	Slackware-based	Slackware

To make software easy to share and download, most Unix systems use a package management system. The most widely used Linux package managers are shown in this table.

Figure 3-2 (from Section 3.2). BSD Package Management Systems.

Package Manager	BSD System
`pkg`	FreeBSD
`pkg_add`	NetBSD
`pkg_add`	OpenBSD

This table shows the package managers you are most likely to encounter when you use a BSD-based version of Unix.

Figure 4-1 (from Section 4.4). Emacs names for special keys.

Name	Key
`C-`	Ctrl
`M-`	Meta
`M-C-`	Meta-Ctrl
`BS`	Backspace
`DEL`	Delete
`ESC`	Esc
`RET`	Return
`SPC`	Space
`TAB`	Tab

When you read about Emacs, you will sometimes see abbreviated names used for special keys. These names are derived from the technology of the 1970s, and they are important enough that it is a good idea to memorize them.

Figure 5-1 (from Section 5.5). Choosing whether or not to save files.

y	Save the specified file
n	Do not save the specified file
!	Save all the remaining files
q	Quit immediately without saving
.	Save the specified file and then quit
C-r	View the specified file
C-h	Display help information

When you tell Emacs to quit, and you have files that have not been saved, Emacs will ask you what to do with each file in turn. Here are the responses you can use to make a choice for each such file.

Figure 6-5 (from Section 6.4). Status characters within the mode line.

Characters	Meaning
--	Buffer has not been modified
******	Buffer has been modified (not yet saved)
%%	Read-only mode: buffer has not been modified
%*	Read-only mode: buffer has been modified

On the mode line, to the right of the colon, Emacs displays two characters to show you the status of the buffer.

Figure 6-6 (from Section 6.7). Completion Commands.

Command	Action
TAB	Complete text in minibuffer as much as possible
C-i	Same as **TAB**
SPC	Complete text in minibuffer up to end of word
RET	Same as **TAB**, then enter the command
?	Create new window, display list of possible completions

When you type in the minibuffer, Emacs helps you, whenever possible, by guessing what you want and letting you use completion commands to make choices. **TAB** *and* **?** *work for commands, files and buffer.* **SPC** *works for commands and buffers.* **RET** *works only for commands.*

Figure 6-7 (from Section 6.8). Choosing whether or not to run a disabled command.

y	Run command; enable it for rest of the work session
n	Do not run command; leave it disabled
SPC	Run command once; leave it disabled
!	Run command; enable *all* commands for the rest of the work session

When you type a disabled command, Emacs tells you the command is disabled and asks you what you want to do. Most of the time, you will answer either **y** *or* **n**. *Note: You don't need to press <Enter>.*

Figure 7-1 (from Section 7.2). Keys to use while typing.

Key	Action
BS	Delete one character to the left of cursor
DEL	Delete one character at the position of cursor
C-d	Same as **DEL**
C-o	Open a new line
C-x u	Undo the last change to the buffer
C-_	Same as **C-x u**
C--	Same as **C-x u**
C-/	Same as **C-x u**
C-q	Insert the next character literally

Note: **C-_** is "<Ctrl>-underscore".

Figure 7-2 (from Section 7.6). Commands for controlling windows.

Command	Description
C-x 0	Delete the selected window
C-x 1	Delete all windows except selected window
C-x 2	Split selected window vertically
C-x 3	Split selected window horizontally
C-x o	Move cursor to the next (other) window
C-x }	Make selected window wider
C-x {	Make selected window narrower
C-x ^	Make selected window larger
M-x shrink-window	Make selected window smaller

Figure 7-3 (from Section 7.7). Commands for controlling buffers.

Command	Description
C-x b	Display a different buffer in selected window
C-x b	Create a new buffer in selected window
C-x C-b	Display a list of all your buffers
C-x k	Kill (delete) a buffer
C-x 4 b	Display a different buffer in next window
C-x 4 C-o	Same as **C-x 4 b**, but don't change selected window

Figure 7-4 (from Section 7.8). Commands for working with files.

Command	Description
C-x C-f	Switch to buffer containing specified file
C-x C-v	Replace buffer contents with specified file
C-x C-s	Save a buffer to file
C-x C-w	Save a buffer to specified file
C-x i	Insert contents of a file into buffer
C-x 4 C-f	Read contents of file into next window
C-x 4 f	Same as **C-x 4 C-f**
C-x 4 r	Same as **C-x 4 C-f**, read-only

Figure 8-1 (from Section 8.2). Commands for moving the cursor.

Backward	Forward	
C-b	**C-f**	a single character
<Left>	<Right>	a single character
M-b	**M-f**	a word
C-p	**C-n**	a line
<Up>	<Down>	a line
M-a	**M-e**	a sentence
M-{	**M-}**	a paragraph

Beginning	End	
C-a	**C-e**	the current line
M-<	**M->**	the entire buffer

Figure 8-2 (from Section 8.3). Commands for moving the cursor through a paragraph or a sentence.

Command	Description
M-}	Move forward one paragraph
M-{	Move backward one paragraph
M-e	Move forward one sentence
M-a	Move backward one sentence

Figure 8-3 (from Section 8.3). Major modes to use when editing a human language.

Mode	Command
Fundamental mode	**`fundamental-mode`**
Text mode	**`text-mode`**
Indented Text mode	**`indented-text-mode`**
Paragraph-Indent Text mode	**`paragraph-indent-text-mode`**

Figure 8-4 (from Section 8.4). Prefix argument combinations.

Prefix	Effect
`M-`*number*	Repeat command specified number of times
`ESC` *number*	Repeat command specified number of times
`C-u` *number*	Repeat command specified number of times
`C-u`	Repeat command 4 times
`C-u C-u`	Repeat command 16 times
`C-u C-u C-u`	Repeat command 64 times
`C-u C-u C-u C-u`	Repeat command 256 times

Figure 8-5 (from Section 8.5). Commands to move throughout the buffer.

Command	Description
`C-v`	Scroll down one screenful
<PageDown>	Same as **`C-v`**
`M-v`	Scroll up one screenful
<PageUp>	Same as **`M-v`**
`M-C-v`	Scroll down in the next window
`M-<`	Jump to the beginning of buffer
`M->`	Jump to the end of buffer
`C-l`	Redisplay the screen, current line in middle

Figure 8-6 (from Section 8.6). Commands to use line numbers.

Command	Description
`M-g g`	Jump to line with specified number
`M-x line-number-mode`	ON/OFF: display line number on mode line

Figure 8-8 (from Section 8.8). Commands to set mark and define a region.

Command	Description
C-@	Set mark to current location of point
C-SPC	Same as **C-@**
C-x C-x	Interchange mark and point
M-@	Set mark after next word (do not move point)
M-h	Put region around paragraph
C-x h	Put region around entire buffer

Figure 8-9 (from Section 8.9). Commands that act upon the region.

Command	Description
C-w	Kill (erase) all the characters
C-x C-l	Convert the characters to lowercase
C-x C-u	Convert the characters to uppercase
M-=	Count the lines and characters
M-|	Run a shell command, use the characters as data

Figure 9-1 (from Section 9.2). Commands to delete text.

Command	Description
BS	Delete one character to the left of cursor
DEL	Delete one character at the position of cursor
C-d	Same as **DEL**
M-	Delete spaces & tabs around point
M-SPC	Delete spaces & tabs around point; leave one space
C-x C-o	Delete blank lines around current line
M-^	Join two lines (delete **RET** + surrounding spaces)

Figure 9-2 (from Section 9.3). Commands to kill text.

Command	Description
C-k	Kill from cursor to end of line
M-d	Kill a word
M-BS	Kill a word backward
M-k	Kill from cursor to end of sentence
C-x BS	Kill backward to beginning of sentence
C-w	Kill the region
M-z *char*	Kill through next occurrence of specified character

Figure 9-3 (from Section 9.3). Commands to move and kill by word or sentence.

	WORDS			SENTENCES	
	Backward	Forward		Backward	Forward
Move:	**M-b**	**M-f**	Move:	**M-a**	**M-e**
Kill:	**M-BS**	**M-d**	Kill:	**C-x BS**	**M-k**

Figure 9-4 (from Section 9.4). Commands to yank text.

Command	Description
C-y	Yank most recently killed text
C-u C-y	Same as **C-y**, cursor at beginning of new text
M-y	Replace yanked text with previously killed text
M-w	Copy region to kill ring, without erasing
M-C-w	Append next kill to newest kill ring entry
C-h v kill-ring	Display the actual values in the kill ring

Figure 9-5 (from Section 9.5). Commands for correcting common typing mistakes.

Command	Description
BS	Delete one character to the left of cursor
DEL	Delete one character at the position of cursor
C-d	Same as **DEL**
M-BS	Kill the previous word
C-x BS	Kill backward to beginning of sentence
M-- M-l	Change previous word to lowercase
M-- M-u	Change previous word to uppercase
M-- M-c	Change previous word to lowercase, initial cap
M-l	Change following word to lowercase
M-u	Change following word to uppercase
M-c	Change following word to lowercase, initial cap
C-t	Transpose two adjacent characters
M-t	Transpose two adjacent words
C-x C-t	Transpose two consecutive lines

Note: **BS** is <Backspace>; **M--** is <Meta-hyphen>.

Figure 9-6 (from Section 9.6). Commands for correcting spelling mistakes.

Command	Description
M-$	Spell check: word at point, or region
M-x ispell-buffer	Spell check: buffer (only)
M-x ispell-region	Spell check: region (only)
M-x ispell	Spell check: buffer, or region
M-TAB	Complete word before point, based on dictionary
ESC TAB	Same as **M-TAB**
M-x flyspell-mode	Highlight spelling mistakes in regular text
M-x flyspell-prog-mode	Highlight spelling mistakes in programs

Figure 9-7 (from Section 9.7). Commands to fill text.

Command	Description
M-x auto-fill-mode	Turn on/off Auto Fill mode
M-q	Fill a paragraph
ESC 1 M-q	Fill+justify a paragraph
M-x fill-region	Fill each paragraph in the region
ESC 1 M-x fill-region	Fill+justify each paragraph in region
M-x fill-region-as-paragraph	Fill region as one long paragraph
ESC 1 M-x fill-region-as-paragraph	Fill+justify region as one paragraph
C-x f	Set the fill column value
C-h v fill-column	Display current fill column value

Figure 10-1 (from Section 10.1). Search commands.

Command	Description
C-s	Forward: Incremental search
C-s RET	Forward: Non-incremental search
M-s w	Forward: Incremental *word* search
M-C-s	Forward: Incremental *regexp* search
M-C-s RET	Forward: Non-incremental *regexp* search

Command	Description
C-r	Backward: Incremental search
C-r RET	Backward: Non-incremental search
M-s w C-r	Backward: Incremental *word* search
M-C-r	Backward: Incremental *regexp* search
M-C-r RET	Backward: Non-incremental *regexp* search

Figure 10-2 (from Section 10.3). Keys to use during a search.

Key	Description
BS	Erase last character you typed
RET	Terminate the search
C-s	Search forward for same pattern
C-r	Search backward for same pattern
C-g	While search is in progress: Stop current search
C-g	While waiting for input: Abort entire command
C-w	Copy the word after point to search string
C-y	Copy current kill ring entry to search string
M-y	Copy previous kill ring entry to search string

Figure 10-3 (from Section 10.5). Non-incremental search commands.

Command	Description
C-s RET	Forward: non-incremental search
C-r RET	Backward: non-incremental search

Figure 10-4 (from Section 10.6). Search commands.

Command	Description
M-s w	Forward: Incremental word search
M-s w C-r	Backward: Incremental word search

Figure 10-5 (from Section 10.7). Search commands for regular expressions.

Command	Description
M-C-s	Forward: Incremental search for regexp
M-C-s RET	Forward: Non-incremental search for regexp

Command	Description
M-C-r	Backward: Incremental search for regexp
M-C-r RET	Backward: Non-incremental search for regexp

Note: If you have a problem using the key sequence **M-C-s**, you can use **ESC C-s** instead.

Figure 10-6 (from Section 10.8). Characters to use with regular expressions.

Character	Description
Char	Any regular character matches itself
`.`	Match any single character except **RET**
`*`	Match zero or more of the preceding characters
`+`	Match one or more of the preceding characters
`?`	Match exactly zero or one of the preceding characters
`^`	Match the beginning of a line
`$`	Match the end of a line
`\<`	Match the beginning of a word
`\>`	Match the end of a word
`\b`	Match the beginning or end of a word
`\B`	Match anywhere not at the beginning or end of a word
`\``	Match the beginning of the buffer
`\'`	Match the end of the buffer
`\`*char*	Quotes a special character
`[ ]`	Match one of the enclosed characters
`[^ ]`	Match any character that is not enclosed

Figure 10-7 (from Section 10.10). Search and replace commands.

Command	Description
`M-%`	Query: Search and replace
`M-C-%`	Query: Search and replace (regexp)
`M-x replace-string`	No query: Search and replace
`M-x replace-regexp`	No query: Search and replace (regexp)

Figure 10-8 (from Section 10.10). Responses during a search and replace command.

Command	Description
?	Display help summary
y	(yes) Replace
n	(no) Do not replace
q	Quit immediately
SPC	Same as **y**
BS	Same as **n**
RET	Same as **q**
!	Replace all remaining matches, no questions
.	(period) Replace current match and then quit
,	(comma) Replace but stay at current position
^	(circumflex) Move back to previous match
C-l	Clear screen, redisplay, and ask again
C-r	Start recursive editing (use **M-C-c** to return)
C-w	Delete matching pattern, start recursive edit

Figure 10-9 (from Section 10.10). Minimum keystrokes to invoke search and replace commands.

Key Sequence	Command	Keystrokes to Use
M-%	**M-x query-replace**	**M-x que RET**
M-C-%	**M-x query-replace-regexp**	**M-x que TAB SPC RET**
M-C-%	**M-x query-replace-regexp**	**M-x que SPC SPC SPC RET**
-none-	**M-x replace-string**	**M-x repl SPC s RET**
-none-	**M-x replace-regexp**	**M-x repl SPC reg RET**

Figure 11-2 (from Section 11.2). The Four Basic Major Modes.

	Name	Type of Data to Edit
Fundamental Mode	**fundamental-mode**	Anything Emacs doesn't know about
Prog Mode	**prog-mode**	Programming source code
Text Mode	**text-mode**	Human-readable text
Special Mode	**special-mode**	Special text created by Emacs

Almost all the major modes are derived, directly or indirectly, from one of these four basic modes.

Figure 11-3 (from Section 11.3). Major Modes: Fundamental Mode Family.

Fundamental Family	Type of Data to Edit
fundamental-mode	Parent mode: General data, not specialized
array-mode	2-dimensional arrays
css-mode	CSS files (Cascading Style Sheets)
edit-abbrevs-mode	Abbreviation definitions
gud-mode	Using debuggers: **gdb**, **sbd**, **dbx**, **xdb**...

The major modes in this list are derived from Fundamental mode.

Figure 11-4 (from Section 11.3). Major Modes: Text Mode Family.

Text Family	Type of Data to Edit
text-mode	Parent mode: human-readable text
bibtex-mode	BibTeX files
change-log-mode	Change logs
html-mode	HTML files
indented-text-mode	Text with indented paragraphs
latex-mode	LaTeX-formatted files
mail-mode	Outgoing email messages
nroffmode	**nroff**- and **troff**-formatted text files
nxml-mode	XML files
org-mode	Outlines for "keeping track of everything"
outline-mode	Outlines with selective display
paragraph-indent-text-mode	Text: leading spaces start paragraphs
plain-tex-mode	TeX-formatted files
rmail-mode	Reading email
sgml-mode	SGML files
slitex-mode	SliTeX-formatted files
tex-mode	TeX-, LaTeX- or SliTeX-formatted files
texinfo-mode	TeXinfo files

The major modes in this list are derived, directly or indirectly, from Text mode. These modes are used to edit human-readable text (English and other languages), as well as marked-up text (HTML, TeX, and so on).

Figure 11-5 (from Section 11.3). Major Modes: Prog Mode Family.

Prog Family	Type of Data to Edit
prog-mode	Parent mode: Programming source code
ada-mode	Ada programs
antlr-mode	ANTLR grammar files (parsers, etc)
asm-mode	Assembly language programs
awk-mode	**awk** scripts
bat-mode	DOS/Windows batch files
c++-mode	C++ programs
c-mode	C programs
cperl-mode	Perl scripts (alternative to **perl-mode**)
emacs-lisp-mode	**emacs** Lisp programs
f90-mode	Fortran 90/95 programs
fortran-mode	Fortran programs
java-mode	Java programs
javascript-mode	JavaScript programs
lisp-interaction-mode	Evaluating Emacs Lisp forms
lisp-mode	Non-**emacs** lisp programs
m4-mode	M4 macros
makefile-mode	Makefiles
modula-2-mode	Modula-2 programs
opascal-mode	Object Pascal programs
pascal-mode	Pascal programs
perl-mode	Perl scripts
prolog-mode	Prolog programs
ps-mode	Postscript files
scheme-mode	Scheme programs (dialect of Lisp)
sieve-mode	Sieve (email filtering) scripts
simula-mode	Simula programs
sh-mode	Shell scripts
sql-mode	SQL programs
tcl-mode	tcl scripts

The major modes in this list are derived, directly or indirectly, from Prog mode. These modes are used to edit computer programs.

Figure 11-6 (from Section 11.3). Major Modes: Special Mode Family.

Special Family	Used for...
special-mode	Parent mode: built-in Emacs services
help-mode	Emacs Help system
messages-buffer-mode	Display messages in ***Messages*** buffer
tar-mode	Looking inside a **tar** file
todo-mode	Managing todo lists

The major modes in this list are derived from Special mode. As a general rule, these modes are used by Emacs when it needs to display information, for example, **help-mode**. *However, some of these modes are also used by tools, such as* **tar-mode** *and* **todo-mode**.

Figure 11-7 (from Section 11.3). Independent Major Modes.

Independent Modes	Type of Data to Edit
conf-mode	Configuration files
completion-list-mode	Display a list of possible completions
dired-mode	Dired directory/file tool
doc-view-mode	Documents: MS Office, OpenDocument, PDF, PS
forms-mode	Field-structured data using a form
hexl-mode	Hexadecimal data, ASCII data
Info-mode	Emacs Info system
normal-mode	Reset major mode to the default for this file
picture-mode	Text-based drawings
ses-mode	Simple Emacs Spreadsheet files

The major modes in this list are not derived from another major mode. Instead, they are independent tools, written to stand alone.

Figure 11-8 (from Section 11.4). Minor modes.

Minor Mode	Description
abbrev-mode	Working with abbreviations
auto-fill-mode	Automatic filling
auto-save-mode	Automatic saving
binary-overwrite-mode	Binary overwriting
compilation-minor-mode	Compiling programs
cua-mode	Use Ctrl-X/C/V for cut/copy/paste
delete-selection-mode	Typed text replaces selection
display-time-mode	Mode line shows: time, load level, mail flag
double-mode	Some keys differ if pressed twice
eldoc-mode	Display info about Lisp function/variable
flyspell-mode	Highlight spelling mistakes in regular text
flyspell-prop-mode	Highlight spelling mistakes in programs
font-lock-mode	Text is fontified as you type
hide-ifdef-mode	Hides certain C code within **#ifdef**
indent-according-to-mode	Indent appropriately for major mode
iso-accents-mode	Display ISO accents
ledit-mode	Editing text to be sent to Lisp
line-number-mode	Mode line shows: line numbers
outline-minor-mode	Work with outlines
overwrite-mode	Overwrite/insert text
paragraph-indent-minor-mode	Text: leading spaces start paragraphs
pending-delete-mode	Same as **delete-selection-mode**
read-only-mode	Buffer contents cannot be changed
resize-minibuffer-mode	Dynamically resize minibuffer
ruler-mode	Header line shows: ruler
show-paren-mode	Highlight matching parentheses, brackets
size-indication-mode	Mode line shows: size of buffer
timeclock-mode-line-display	Track time intervals you spend working
toggle-read-only	Buffer contents cannot be changed
toggle-viper-mode	Turn **viper-mode** off and on
tool-bar-mode	Toggle: Display help on tool bar or mode line
tooltip-mode	Display Emacs toolbar
transient-mark-mode	Highlight region when defined
view-mode	Page through a file (similar to **less** pager)
viper-mode	Turn on (not off) emulation of **vi** text editor
whitespace-mode	Show all whitespace: **SPC**, **TAB**, **RET** chars
whitespace-newline-mode	Show **RET** characters

Figure 11-10 (from Section 11.7). Commands to set and describe modes.

Command	Description
M-x *mode-name* **RET**	Set the specified major or minor mode
C-h v major-mode RET	Display information about current major mode
C-h m	Describe current major and minor modes
C-h f *mode-name* **RET**	Describe the specified mode
C-h f *-mode RET	Display names of all modes
C-h a mode	Display summary of all modes

Figure 12-1 (from Section 12.1). Running shell commands.

Command	Description
M-!	Run a shell command
M-|	Run a shell command using region as input
M-x shell	Start a separate shell in its own buffer

Figure 12-2 (from Section 12.3). Help facility options.

Command	Description
C-h C-h	Display a summary of all the Help options
C-h ?	Same as **C-h C-h**.
C-h q	Exit from a Help command loop (quit)

Command	Description
C-h a	Show all the functions containing a specified word
C-h b	Display a full list of all the key bindings
C-h c	You specify a key, Emacs tells you what it does
C-h f	You specify a function, Emacs describes it
C-h h	Display the "Hello" file
C-h k	You specify a key, Emacs describes its function
C-h m	Describe the current major and minor modes
C-h v	You specify a variable, Emacs describes it
C-h w	You specify a function, Emacs shows you its key

Command	Description
C-h t	Emacs tutorial
C-h i	Emacs reference manuals (Info documentation browser)

Figure 12-3 (from Section 12.4). General Info commands.

Command	Description
?	Display a summary of Info commands
h	Start the Info tutorial
q	Quit Info: remember current location
C-x k	Quit Info: do not remember current location

Figure 12-4 (from Section 12.4). Info commands to select a node.

Command	Description
n	Jump to next node in the sequence
p	Jump to previous node in the sequence
u	Jump to the "up" node (the menu you came from)
l	Jump to last node you looked at
m *selection*	Pick a node from a menu
f	Follow a cross-reference
i	Look up topic in the index, then jump there
,	(comma) Jump to next match from previous **i** command

Figure 12-5 (from Section 12.4). Info commands to read a node.

Command	Description
SPC	Go forward (down) one screenful
BS	Go backward (up) one screenful
b	Go to beginning of the node
.	Same as **b**
C-l	(<Ctrl-L>) Redisplay the current screen

Figure 12-6 (from Section 12.6). Built-in tools.

Program	Description
calendar	Calendar and diary
customize	Tool to help you change user options
dired	Directory and file manager
eww	Web browser
ses	Create and edit spreadsheet files

Figure 12-7 (from Section 12.6). Dired commands.

Program	Description
q	Quit
h	Display Help summary
m	Mark current file/directory, move cursor down
BS	Unmark current file/directory, move cursor up
u	Unmark file/directory, move cursor down
U	Unmark all files/directories
C	Copy marked files or current file
R	Rename current file
R	Move marked files or current file to another directory
d	Flag file for deletion
x	Delete files flagged by **d**
g	Refresh by reading the directories again

Here are the most important commands while using Dired, the Emacs file manager. Notice that the commands are case sensitive.

Figure 12-8 (from Section 12.7). Games and diversions.

Program	Description
animate	Display animated birthday message
bubbles	Game: Remove groups of bubbles until all gone
blackbox	Puzzle: Find objects inside a black box
doctor	Eliza program: acts like a psychotherapist
dunnet	Adventure-style exploration game
gomoku	Game: plays Gomoku with you
hanoi	Visual solution to Towers of Hanoi problem
landmark	Neuro-network robot that learns about landmarks
life	Game of Life: auto-reproducing patterns (not a game)
mpuz	Multiplication puzzle: guess the digits
pong	Video game: classic ping-pong game
snake	Video game: you control a growing snake
solitaire	Jump pegs across other pegs (not the card game)
spook	Generates words to get government's attention
tetris	Video game: you manipulate falling blocks

Index of Emacs Key Sequences

<Alt>, 72–73, 75, 78, 106, 127
<Alt-Ctrl-S>, 166
<Backspace>, 74, 75
<Ctrl>, 74
<Delete>, 74, 75
<Down>, 122
<Esc>, 74
<Left>, 122
<Meta>, 74
<Meta>+<Ctrl>, 74
<PageDown>, 127
<PageUp>, 127
<Return>, 74, 75
<Right>, 122
<Space>, 74, 75
<Tab>, 74, 75
<Up>, 122

BS, 107, 138, 147, 157, 159
DEL, 107, 138, 147
ESC, 74, 75
RET, 137, 157, 158
SPC, 137
TAB, 137

C-= <Ctrl>, 74
C-M-= <Ctrl>+<Meta>, 74
M-= <Meta>, 74
M-C-= <Meta>+<Ctrl>, 74

C--, 107
C-/, 107–111, 191, 210
C-@, 131, 132
C-_, 107–110

C-a, 122
C-a C-k C-k, 141
C-b, 122
C-d, 107, 138, 147
C-e, 122
C-f, 111, 122
C-g, 97, 98, 112, 157, 158, 161, 198, 203

C-h, 105
C-h ?, 192
C-h a, 192
C-h a mode, 182, 184
C-h b, 106, 134, 156, 192, 193
C-h c, 192, 193
C-h c C-x C-w, 193
C-h C-h, 192
C-h f, 192, 194, 203
C-h f *-mode, 182, 184
C-h f auto-fill-mode, 183
C-h f blackbox, 198, 204
C-h f bubbles, 203
C-h f landmark, 208
C-h f mode-name, 182
C-h f ses-mode, 202
C-h f text-mode, 183
C-h f transient-mark-mode, 183, 184
C-h f write-file, 194
C-h h, 192, 194

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0

C-h i, 90, 192, 195, 197
C-h k, 192, 193
C-h k C-x C-w, 193
C-h m, 182, 183, 192, 194
C-h q, 192
C-h t, 192, 194
C-h v, 146, 192, 194
C-h v kill-ring, 143, 146
C-h v major-mode, 182
C-h v visible-bell, 194
C-h w, 192, 193

C-k, 141
C-k C-k, 141
C-l, 127, 128
C-n, 122, 125
C-o, 107, 108
C-p, 122, 124
C-q, 107, 108
C-r, 155–158, 161, 170, 171
C-r RET, 155, 161
C-s, 155–161
C-s RET, 155, 161
C-SPC, 129, 131, 132
C-t, 148
C-u, 125, 126
C-u C-u, 126
C-u C-u C-u, 126
C-u C-u C-u C-u, 126
C-u C-y, 143–145
C-v, 106, 127
C-w, 133, 142, 157, 171

C-x, 110
C-x 0, 105, 106, 114, 156
C-x 1, 114, 193
C-x 2, 114, 115
C-x 3, 114, 115
C-x 4 b, 117
C-x 4 C-f, 120
C-x 4 C-o, 116, 117
C-x 4 d, 199
C-x 4 f, 118, 120
C-x 4 r, 118, 120
C-x ^, 114, 115
C-x b, 116, 117, 193, 195, 197, 198, 210, 211
C-x b *Shell Command Output*, 190
C-x BS, 109, 141, 142, 147
C-x C-b, 90, 116
C-x C-c, 84, 85
C-x C-f, 97, 118–120
C-x C-f calculate.ses, 202
C-x C-l, 102–104, 133
C-x C-o, 138, 140
C-x C-q, 95
C-x C-s, 118, 119
C-x C-t, 147, 148
C-x C-u, 102–104, 133
C-x C-v, 118, 119
C-x C-w, 118, 119, 134, 193
C-x C-x, 131, 132
C-x d, 199
C-x f, 152, 154
C-x h, 131, 132, 134, 190
C-x h C-w, 93, 142, 156
C-x i, 118, 119
C-x k, 89, 96, 116, 191, 197, 198, 203, 205, 207
C-x o, 114, 115, 134, 193, 197, 198
C-x o C-x 1, 184, 193, 194
C-x u, 103, 108–110, 191
C-x z, 108–111
C-x z…, 109–111
C-x {, 114
C-x }, 114

C-y, 143, 144, 157, 159, 161

ESC |, 134
ESC 1 M-q, 152, 153
ESC 1 M-x fill-region, 152
ESC 1 M-x fill-region-as-paragraph, 152, 153
ESC 1 M-| sort, 190
ESC 3 M-x hanoi, 207
ESC 5 M-x hanoi, 206

ESC *number command*, 126
ESC C-s, 166
ESC TAB, 151

M-!, 189–191
M-! users, 189
M-! who, 189
M-$, 149–151
M-%, 167–170
M--M-c, 147
M--M-l, 147
M--M-u, 147
M->, 122, 128, 130
M-*number command*, 126
M-=, 130, 133, 134
M->, 122, 127, 128, 130
M-@, 131, 132
M-, 138, 139
M-^, 138, 140
M-{, 122–124
M-|, 134, 189–191
M-| fmt, 135
M-| sort, 190
M-| sort-u, 134
M-}, 123–124

M-a, 122, 124, 142
M-b, 122, 142
M-BS, 139, 141, 142, 147, 160
M-c, 147
M-C-%, 167
M-C-c, 170, 171
M-C-r, 155, 163, 165
M-C-r RET, 155, 163, 165
M-C-s, 155, 163, 165, 166
M-C-s RET, 155, 163, 165
M-C-v, 127, 128
M-C-w, 143, 145
M-d, 141, 142
M-e, 122–124, 142
M-f, 122, 142
M-g g, 128, 129
M-h, 131, 132
M-k, 141, 142
M-l, 147
M-q, 153
M-s w, 155, 162
M-s w C-r, 155, 162
M-SPC, 138, 139
M-t, 147, 148
M-TAB, 149, 150
M-u, 147
M-v, 106, 127
M-w, 143, 145

M-x, 123, 124, 180, 197, 199, 203
M-x *command*, 83, 84, 88, 101
M-x animate, 204
M-x animate-birthday-present, 204
M-x auto-fill-mode, 152, 153, 187
M-x blackbox, 204
M-x bubbles, 204
M-x calendar, 197, 201
M-x customize, 201
M-x dired, 199
M-x doctor, 204, 205
M-x dunnet, 205
M-x eldoc-mode, 186
M-x emacs-lisp-mode, 186
M-x eww, 201
M-x fill-region, 152, 153
M-x fill-region-as-paragraph, 153
M-x flyspell-mode, 149, 151
M-x flyspell-prog-mode, 149, 151
M-x gomoku, 206
M-x hanoi, 206, 207
M-x ispell, 149, 150
M-x ispell-buffer, 149, 150
M-x ispell-region, 149, 150
M-x landmark, 207
M-x life, 207
M-x mode-name, 182
M-x mpuz, 208–209
M-x overwrite-mode, 180
M-x pong, 209

M-x psychoanalyze-pinhead, 212
M-x read-only-mode, 181
M-x rename-uniquely, 181
M-x replace-regexp, 167, 169
M-x replace-string, 167, 169
M-x shell, 189, 191
M-x snake, 209
M-x solitaire, 210
M-x spook, 210–211
M-x tetris, 211
M-x text-mode, 180
M-x yow, 212

M-y, 143, 144, 157, 159, 160
M-z char, 141

Index of Emacs Variables and Functions

Emacs Lisp Variables

fill-column, 153, 154
kill-ring, 146
major-mode, 182
setq, 170
setq-default, 188
text-mode-hook, 187
visible-bell, 194

Emacs Lisp Functions

animate, 203, 204
auto-fill-mode, 152, 179, 182
blackbox, 203, 204
bubbles, 203, 204
calendar, 198, 201
customize, 198, 201
dired, 198, 199
disable-command, 104
doctor, 203, 204
downcase-region, 102–104
dunnet, 203, 205
eldoc-mode, 186
emacs-lisp-mode, 173, 181, 186
enable-command, 104
eww, 198, 201
fill-column, 153, 154
fill-region, 153
fill-region-as-paragraph, 153
flyspell-mode, 151
flyspell-prog-mode, 151
forward-char, 87, 88
forward-word, 87, 88
fundamental-mode, 175
gomoku, 203, 206
hanoi, 203, 206
html-mode, 175
isearch-backward, 156
isearch-forward, 156
ispell, 150
ispell-buffer, 88, 150
ispell-region, 150
keyboard-quit, 203
landmark, 203, 207
life, 203, 207
line-number-mode, 129, 180
mpuz, 203, 208
next-line, 87, 88
overwrite-mode, 179–181
pong, 203, 208
previous-line, 87, 88
prog-mode, 175
psychoanalyze-pinhead, 212
query-replace, 170
query-replace-regexp, 170
read-only-mode, 95, 179–181
rename-uniquely, 191, 192

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0

repeat, 108–111
replace-regexp, 169
replace-string, 169
ses, 198, 201
ses-mode, 202
sgml-mode, 175
shell, 189-191
show-paren-mode, 180
shrink-window, 115
snake, 203, 208
solitaire, 203, 209
special-mode, 175
spook, 203, 209
tetris, 203, 210
tex-mode, 175
text-mode, 173, 174, 180, 183
toggle-read-only, 82, 95
transient-mark-mode, 183–184
undo, 108–111
universal-argument, 125
upcase-region, 102–104
write-file, 193–194
yow, 212

Index of Unix Keys, Files and Commands

Unix Keys

(superuser shell prompt), 32
$ (Bash shell prompt), 32, 33
% (C-Shell prompt), 32
- (command options), 34
. (current directory), 49
.. (parent directory), 49–51
/ (root directory), 45, 46
| (pipe symbol), 230
~ (home directory within shell prompt), 32, 33
<Backspace>, 34, 35
<Ctrl>, 72
<Ctrl-C>, 35–37
<Ctrl-D>, 35–37
<Ctrl-U>, 35
<Ctrl-W>, 35
<Ctrl-X>, 35
<Enter>, 35, 37
<Left>, 34, 35
<Right>, 34, 35

Unix Files

/dev/null, 229
/dev/random, 229
/proc/meminfo, 229
/proc/sys/kernel/hostname, 229

Unix Commands

cat, 51
cat /proc/sys/kernel/hostname, 229
cd, 50, 52
chmod, 51, 52
cmp, 51
cp, 51
du, 51, 52
file, 51
find, 51
fmt, 135
head, 51
less, 41, 42
less /proc/cpuinfo, 229
less /proc/meminfo, 229
logout, 19, 37
ls, 34, 49
ls -R, 229
mkdir, 52
mv, 51, 52
od, 51
pwd, 49, 51, 52
rm, 51

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0

sort, 134, 190
sudo, 229
sudo ls -R / | less, 230
sudo ls -R / | wc -l, 299
tail, 51
touch, 51
tree -d, 52
users, 189
wc -l, 230
whereis, 51
who, 189

General Index

80 characters/line (IBM punch cards), 223
132 characters/line (IBM 1403 printer), 223–224

Acknowledgments, xix, xx
Alphameric, 209
Archive. *See* Installing Unix Software: archive
Argument, 124–126
ASCII code, 76
ASCII text, 3
Asimov, Isaac, 158, 159, 162
AT&T. *See also* Unix: History, 14, 15, 220

Bash. *See* Shell: Bash
Brooks, Walter R. *See* Freddy the Pig: Walter R. Brooks
Bucky Bits, 75–77
Buffers
- **`*Buffer List*`** buffer, 116
- **`*Choices*`** buffer, 150
- **`*Completions*`** buffer, 101, 102
- **`*Help*`** buffer, 105, 106, 113, 128, 134, 156, 193
- **`*info*`** buffer, 195
- **`*scratch*`** buffer, 90, 93, 95, 105, 106, 114, 115, 117, 156
- **`*Shell Command Output*`** buffer, 134, 189, 190
- **`*shell*`** buffer, 191
- change to another buffer, 117
- commands to control buffers, 115–117
- compared to windows, 114
- creating, 115–117
- creating just for shell commands, 191
- definition, 89
- deleting, 116
- display list of buffers, 116
- major mode, each buffer has only one, 180–181
- minor modes, each buffer can have as many as you want, 180–181
- multiple buffers using the same file, 115, 116
- point, each buffer has its own, 121
- read a file into a buffer, 97
- read-only, 94–96
- scrolling through, 127
- using, 89–91

Case
- case insensitive, 54, 160
- case sensitive, 54, 160
- convert text to lowercase, 102, 147
- convert text to uppercase, 102, 147
- lowercase, 54
- searching, upper- and lower case, 160–161, 165
- Unix commands, 33–34
- uppercase, 54

Cicero, Marcus Tullius, 107
CLI. *See* User interfaces: command-line interface

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0

Command line. *See* User interfaces: command-line interface
Command line editing
  history list, 37–40
  key combinations to use, 39
  starting Emacs (*see* Starting Emacs: from the command line)
  two different meanings, 31
  with Microsoft Windows, 6, 227–228
  with OS X, 227–228
  with Unix and Linux, 6, 31
Command-line interface. *See* User interfaces: command-line interface
Command processor. *See* Shell
Command summaries. *See* Appendix B on page 231
Compiling files. *See* Installing Unix Software: compiling files
Completion
  commands, 99–100
  completion list, 184
  definition, 99
  importance of, 98, 99
  modes, used to set, 180
  search and replace commands, minimum sequences, 167–170
Computer paper, 21
Correcting text. *See* Text editing: correcting and modifying
Current directory. *See* Unix filesystem: current directory

Dedication, xix, xx
Deleting text. *See* Text editing: deleting text
Delimiter, 33
Desktop environment. *See* User interfaces
Dired (Emacs file manager)
  commands, 199, 200
  reference card, 200
  starting, 199
Disabled commands. *See* Emacs commands: disabled
Distribution, 15, 17
DMG file. *See* Installing Emacs: using OS X
Dmitry Shkatov. *See* Shkatov, Dmitry
Dotfiles. *See* Unix filesystem: file names

Echo area. *See* Windows: echo area
Editing. *See* Text editing
Editor. *See* Text editor
`.emacs` initialization file
  `auto-fill-mode`, set when using **`text-mode`**, 187, 188
  default mode, how to set, 187–188
  suggested customizations, 185–186
Emacs
  basic design, 112–113
  buffer (*see* Buffer)
  command, definition of, 87
  compared to **`vi`**/Vim, 112
  completion (*see* Completion)
  correcting text (*see* Text editing: correcting and modifying text)
  definition, 1, 3
  deleting text (*see* Text editing: deleting text)
  entering text (*see* Text editing: entering text)
  games (*see* Emacs games)
  GNU Emacs, 2, 8, 18
  help (*see* Help facility)
  importance of teaching yourself, 5
  installing, 57–59
  key sequences (*see* Emacs key sequences)
  keys (*see* Emacs keys)
  learning, strategy for, 71
  meaning of name "Emacs", 8
  modes (*see also* Major modes, Minor modes, 173)
  modifying text (*see* Text editing: correcting and modifying text)

origin, 8
overview, 5
practicing, 105–106
read-only, 82, 94–96, 181
recovering data after system failure, 83–84
starting (*see* Starting Emacs)
stopping, 82
three comprehensive help systems, 192
tools (*see* Emacs tools)
versions, 18
windows (*see* Windows)
working environment, 5–8
Emacs commands
canceling a command, 97, 98, 112
command summaries (*see* Appendix B on page 231)
correcting, 107–108, 146–149
disabled, 102–104
echoing, 96–97
enabled, 102–104
input event, 87
key bindings, 87–89, 105, 106, 193
long names, why?, 89
practicing, 105–107
redoing a command, 108–111
repeating a command, 108–111
typing, 107–108
undoing a command, 108–111
Emacs games
Animate (moving text birthday greeting), 204
Blackbox (puzzle game), 198
Bubbles (colored bubbles game), 203, 204
Doctor (acts like a psychotherapist), 88, 89, 203, 204, 212–214
Dunnet (text-based exploration game), 198, 203, 205–206
Game of Life (two-dimensional grid-like universe), 203, 207
Gomoku (board game using markers), 203, 206
Landmark (watch simple neuro-network robot), 203, 207
Multiplication Puzzle (alphameric math puzzle), 203, 208
Peg Solitaire (board with holes and pegs), 203, 209
Pong Video Game (simulated ping-pong game), 203, 208
Snake (the Snake video game), 203, 208
Spook (generate subversive-sounding words), 203, 209
starting a program, 197–198
Tetris (the Tetris video game), 203, 210
Towers of Hanoi (visual recursive puzzle), 203, 206
Yow (acts like Zippy the Pinhead), 212
Emacs key sequences. *See also* Emacs keys
correcting mistakes, 97
display information about a specific key sequence, 193
fixing <Alt-Ctrl-S> problem with Unity desktop environment, 166
key sequence, definition, 87
M-SPC key problems within a terminal window, 140
M-x, using to execute a specific command, 88, 89
Macintosh keyboard, using, 75
Meta key problems within a terminal window, 78, 127, 163, 166
prefix argument, 124–126
related to early Unix terminals, 74
searching (*see* Searching)
strange key sequences, why so many?, 77, 110, 112
strategies for learning, 88
writing **C-** for <Ctrl>, 72
writing **C-M-** for <Ctrl>+<Meta>, 73

Emacs key sequences (*cont.*)
writing **M-** for <Meta>, 73
writing **M-C-** for <Meta>+<Ctrl>, 73
Emacs keys. *See also* Emacs key sequences
<Option> (Macintosh keyboard), 72, 73
Knight keyboard, 75–77
Meta key, history of, 75–77
Sail keyboard, 76
Symbolics keyboard, 76
Emacs Lisp
comments (;), 186
documentation lookup, 186
evaluates functions, 186
expressions, 185
function names that end in "**.el**" , 173
functions, 74, 82, 87, 186–188
functions that define modes, 173
functions, display information about, 193
functions, display key bindings, 193
macro **defun**, 186
macros, 7, 8
recursive programming, 206
used in **.emacs** initialization file, 185–186
variables, 187, 188
Emacs tools
Calendar (calendar and diary), 197, 198, 201
Customize (change user options), 198, 201
Dired (*see* Dired Emacs file manager)
Emacs Web browser, 198, 201
Info documentation, 200
learning how to use a program, 198
Simple Emacs Spreadsheet (create/edit spreadsheet files), 198, 202
starting a program, 197
stopping a program, 198
Emacs tutorial, 194–197
Enabled commands. *See* Emacs commands: enabled

Files. *See also* Unix files and Unix filesystem
ASCII File, 4
commands to work with files, 117–120
compared to buffers, 119
definition, 117
read file contents into buffer, 117, 118
save buffer to file, 118
switch to buffer containing specific file, 118
text File, 4
visiting a file, 118, 119
Fortran, history, 219
Freddy the Pig
books, 228
Boomschmidt's Stupendous and Unexcelled Circus, 43, 45, 46, 49
Centerboro, New York, 43, 228
Freddy and the Men From Mars, 43, 228
Ignormus, 159, 160
Leo the lion, 46, 48
Martians, 43, 46, 49
Walter R. Brooks (author), 228
Free Software Foundation (FSF)
GNU (*see* GNU)
FSF. *See* Free Software Foundation

Games. *See* Emacs games
GCC, 7
Gnome. *See* User interfaces
GNU
General Public License, 9
history of, 14
Manifesto, 10–12
meaning of name "GNU", 10, 217
Project, The, 11
GNU compiler collection. *See* GCC
GNU Emacs. *See* Emacs

GPL. *See* GNU: General Public License
Graphical user interface.
See User interfaces
Graphics, 4
GUI. *See* User interfaces

Hackers, 77, 221
Hahn, Harley. *See* Harley Hahn
Hardware, 13, 14
Harley Hahn
"Everything we teach you is true…", 139, 140
"Here is a way to make a bit of money for yourself…", 139
"Much of what I know about human nature…", 228
"What are the best things in life?"
"When computer experts need to talk…", 226
Biography, xxi
early books, 225
personal note from, xxi, 214
Why I went to graduate school in San Diego., 223
Why I wrote this book., 214
Harley Hahn's Guide to Unix and Linux, 18, 52, 90, 148, 164
Heffalump, 159
Help facility
buffer, 90, 105
C-h to access the Emacs help facility, 182
display "hello" in many different languages, 192, 194
display information about a Lisp function, 183, 192–194
display information about a specific key sequence, 192, 193
display information about current major and minor modes, 192, 194
display key binding for a specific function, 192, 193
Emacs tutorial (*see* Emacs tutorial)
help options, display summary, 192
help window, maximizing, 184
Info facility (*see* Info facility)
key bindings, display, 105, 106
reference manuals (*see* Info facility)
starting, 192
Home directory. *See* Unix filesystem: home directory
Host, 19

IBM compatible computers, 24, 224
IBM PC, 23, 24
IDE (integrated development environment), 6
Ignormus. *See* Freddy the Pig: Ignormus
Info facility
commands to read a node, 196
commands to select a node, 196
definition, 195
Emacs reference manual, 195
Emacs reference manual compared to Emacs Help facility
Info reference manual (describes Info itself), 195
learning, 195, 197
nodes, 196
starting, 195–197
stopping, 197
Input event. *See* Emacs commands: input event
Inserting text. *See* Text editing: inserting text
Installing Emacs
overview, 57
using a Linux manual installation, 62–65
using a package manager, 60–62
using Linux package manager APT, 60, 61
using Linux package manager RPM, 60, 62
using Microsoft Windows, 68–70
using OS X, 65–68

Installing Unix Software
archive, 58, 59
binary files, 58
compare packages to consumer apps, 230
compiling files, 58
extracting files, 58
important concepts
metadata within an archive, 57
package, 57–59
package manager, 59
package managers for BSD, 60
package managers for Linux, 60–62
repository, 59
source file, 58, 59
tarball, 58, 59
zip file, 59
Investing, psychology of , 129

John McCarthy. *See* McCarthy, John
John Socha. *See* Socha, John
Joy, Bill, 15

Kajsa Anka, 19, 32, 33, 50, 51
Kernel, 15–18
Key binding. *See* Emacs commands: key binding
Killing text. *See* Text editing: killing text
King Solomon, 5
Knights of the Lambda Calculus. *See* Lisp: hackers, Knights of the Lambda Calculus

Line Numbers. *See* Text editing: line numbers
Linus Torvalds. *See* Torvalds, Linus
Linux
Harley Hahn's Guide to Unix and Linux, 18, 52, 90, 148, 164
Debian, 60
distribution, 17
history, 16
installing Emacs (*see* Installing Emacs)
Mint, 60
terminal emulators, 27, 226
terminal windows (*see* Terminal windows)
Ubuntu, 27
Ubuntu Dash, 29, 30
Ubuntu Launcher, 29, 30
Ubuntu Terminal program, 30, 227
Ubuntu Unity desktop environment, 29, 30
Unity desktop environment, 166
virtual terminals (*see* Virtual terminals: using with Linux)
Lisp
at MIT, 75, 76
at Stanford University, 76
comments (;), 186
computers, 76
evaluates functions, 186
expressions, 185, 186
first homoiconic programming language, 185
functions, 185, 186
hackers, Knights of the Lambda Calculus, 77
history, 219
keyboard, 75–77
macro **`defun`**, 186
reasons to learn, 185
recursive programming, 206, 219
variables, 187, 188
Lisp, Emacs. *See* Emacs Lisp
Lisp machine, 75–77

Major modes
buffer has only one major mode, 180
comprehensive list of major modes, 175–178
current major mode name displayed on mode line, 174
default, setting your, 174
Fundamental mode family, 176
Independent major modes, 178
parent mode (used to derive other modes), 174

Prog mode family, 177
purpose: to edit particular type of text, 173
Special mode family, 178
Text mode family, 176
used to edit particular type of text, 175
which major mode to when not sure, 180
Marked-up text, 5
McCarthy, John, 219
Microsoft Windows. *See* Operating systems: Microsoft Windows
Midnight Commander, TUI-based fine manager, 225
Minibuffer, 97–98
Minix, 18
Minor modes
buffer can have multiple minor modes, 180
comprehensive list of important minor modes, 178–180
purpose: to turn features on and off, 173, 178, 180
Mode line. *See* Windows: mode line
Modes
all modes, display names , 184
all modes, display summary, 184
current major and modes, display information, 182, 183, 194
defined by Emacs Lisp functions, 173
definition of major and minor modes, 173
describe specific mode, 183, 184
display information about a specific mode, 182, 183
learning about, 182–184
major mode, definition, 174–175
names end in "**`-mode`**", 173
setting, 180–182
setting, using completion, 180
Modifying text. *See* Text editing: correcting and modifying text
Money, secret way to make, 139
Multiuser system, 19

Norton, Peter, 225

Operating systems
Linux (*see* Linux)
Microsoft Windows, 14
Microsoft Windows commands, 31
Microsoft Windows filenames, 56
OS X, 15, 222
OS X filenames, 55, 56
Unix (*see* Unix)
Xinu, 222
XNU kernel, 222
OS X (*see* Operating systems: OS X)
installing Emacs (*see* Installing Emacs: using OS X)

Package. *See* Installing Unix Software: package; repository
Package management system. *See* Installing Unix Software: package manager
Pathname. *See* Unix filesystem: pathname
Personal notes, 2, 3, 215–230
Peter Norton. *See* Norton, Peter
Point. *See* Region: point

Queen of Sheba, 5, 8

Read-only. *See* Emacs: read-only
Recovering data after system failure. *See* Emacs: recovering data after system failure
Region
case, change, 133
count lines and characters, 133, 134
cursor position, 121
definition, 129, 131
format, 135
kill (erase), 133
mark, definition, 129
mark, setting, 131, 132
operating on, 133–135

Region (*cont.*)
point, 121, 167, 185
point, definition, 131
region, defining, 131
region, defining as entire buffer, 132
region, exchange mark and point, 132
setting and using, 129
shell command, use contents of region as data, 134, 190
sort, 134
Regular expressions
case sensitive, 165
characters to use, 164, 165
definition, 162
range of characters, 165
searching for, 162–163
Richard Stallman. *See* Stallman, Richard
Ritchie, Dennis, 14, 21
Root directory. *See* Unix filesystem

Scrolling. *See* Text editing: scrolling
Searching
append kill ring entry to search string, 159
append word after point to search string, 158, 159
corrections, 157
incremental search, 156, 157
keys to use during a search, 157–159
keys to use during a search and replace, 167, 169, 170
non-incremental search, 161
practicing, 156
recursive editing, putting search and replace on hold, 170, 171
recursive editing, when to use, 170–171
regular expressions (*see* Regular expressions)
search and replace, 167–170
search for word in buffer without typing the word, 158
search string, 155
upper- and lower case, 165
word searching, 162
Shell
Bash, 16, 32
Bash configuration files, 54
Bash history, 221
commands (*see* Shell commands)
default, 32
definition, 4
Korn Shell, 32
new shell, start from within Emacs, 190
script, 4, 16
shell prompt (*see* Shell prompt)
Shell commands
create sorted list of all the Emacs commands, 134
multiple buffer just for shell commands, 191
prefix argument to indicate replace data with output, 190
single buffer just for shell commands, 191
sort all the lines in a buffer, 190
use contents of region as data, 134
using contents of region as data, 190
using without leaving Emacs, 189
Shell prompt
(Unix superuser shell prompt), 32
$ (shell identifier), 32
$ (Unix Bash shell prompt), 32, 33
% (Unix C-Shell prompt), 32
~ (home directory), 33
~ (Unix home directory within shell prompt), 32, 33
delimiter, 33
Shkatov, Dmitry (technical reviewer), xvii, xix
Socha, John, 225
Software
Free, 9–10, 217, 221, 225, 226, 221
open source, 9, 217, 225
proprietary, 9, 225

Source files. *See* Installing Unix software: binary file
Source files. *See* Installing Unix software: source file
Stallman, Richard, 8–11, 15, 217, 218
Starting Emacs
  & (using at end of command), 81
  -f option, using, 82, 95, 181
  -nw option, using, 81, 82, 93, 105, 107, 114, 156
  -Q option, using, 79, 80, 93, 105, 106, 156
  .emacs initialization file (*see* **.emacs** initialization file)
  alias, using, 82
  from a terminal window, 81–82
  from the command line, 79
  read-only editor, as, 82, 95, 181
  specifying a file to edit, 82, 119, 181
  without specifying a document, 90
Stevenson, Robert Louis, 8
Stopping Emacs. *See* Emacs: stopping
Sun Microsystems, 220

Tannenbaum, Andrew, 18
Tarball. *See* Installing Unix software: tarball
TECO text editor, 8, 9
Terminal emulators
  virtual terminal (*see* Virtual terminals)
  UXterm, 227
  VT100, 24, 223, 224
  Xterm, 227
Terminal windows
  M-SPC problems, 140
  Meta key problems, 78, 127
  starting Emacs (*see* Starting Emacs: from a terminal window)
Terminals
  emulators (*see* Terminal emulators)
  graphics, 22, 23
  in the 1970s, 215
  printing, 21, 22, 222
  text, 22
  using with Linux, 27
  using with Unix, 27–30
  video display, 21, 22
  VT100, 22, 223
  VT52, 223
Terminals, real. *See* Terminals
Terminals, virtual. *See* Virtual terminals
Text, 3–5
Text editing
  appending to kill ring entry, 143–146
  blank line, 123, 124
  buffer, scrolling through, 127
  case, changing, 146
  character, counting, 129, 130
  characters, working with, 126
  correcting and modifying text, 107–108
  cursor, moving the, 122, 125
  deleting compared to killing, 137
  deleting text, 107–109, 137–140, 147
  entering text, 108
  erasing text (*see* Text editing: killing text, Text editing: deleting text)
  filling text, 151–154
  indentation, 124
  inserting text, 107, 108
  kill ring, 143–146
  kill ring entry, 143–146
  killing compared to deleting, 137
  killing text, 137, 145, 147
  line numbers, displaying, 128
  line numbers, jumping to, 128, 129
  lines, counting, 129, 130
  lines, joining two into one, 140
  lines, working with, 124, 125
  major modes, which ones to use, 123
  mode-less editor, 112
  next window, scrolling through, 128
  numeric argument, 124
  paragraph definition, 124
  paragraphs, working with, 123, 124
  practicing, 107

Text editing (*cont.*)
prefix argument, 124–126, 138, 190, 191
redisplaying the screen, 128
reformatting text, 151
repeating a command multiple times, 124–126
searching (*see* Searching)
scrolling, 127
sentence definition, 124
sentences, working with, 123, 124
spelling mistakes, correcting, 149–151
spelling, checking, 150, 151
transposing characters, words, lines, 148
whitespace, 124, 137
words, counting, 129, 130
yanking text, 143–145
Text editor, 3–5
Text-based user interface. *See* User interfaces
Thompson, Ken, 14, 15, 21, 220
Time travel, 19, 223, 224
Time-sharing system, 19
Tools. *See* Emacs tools
Torvalds, Linus, 16–18, 224
TUI. *See* User interfaces
Typing mistakes. *See* Text editing: correcting and modifying text

Unity. *See* User interfaces
Unix
A/UX (from Apple), 221
Account, 19, 20
BSD, 15, 218
command line (*see* User interfaces: command-line interface)
files (*see* Unix files)
filesystem (*see* Unix filesystem)
Harley Hahn's Guide to Unix and Linux, 18, 52, 90, 148, 164
History, 14, 218
in the 1970s, 215
manual (*see* Unix manual)
password, 19, 20
print (meaning "display"), 222
`root` userid, 20, 46, 48
signal, end of file (**`eof`**), 19, 37
signal, interrupt (**`intr`**), 36
superuser, 20
system administrator, 19, 20
terminal (*see* Terminals)
userid, 18–20
`vi`/Vim text editor, 112
workstations, 24, 224
Unix commands
case sensitive, 34
command line editing (*see* Command line editing)
command summaries (*see* Appendix B on page 231)
directory commands, list of most important, 51, 52
echoing, 96–97
file commands, list of most important, 51
format, 33, 34
history list (*see* Command line editing)
installing Emacs (*see* Installing Emacs)
long options, 33, 34
making corrections, 34–35
pager programs, 41–42
short options, 33, 34
syntax, 40
what happens when you enter a command?, 35–37
Unix files
device files, 44, 228
directory, 43, 44
file, definition, 43
folder, 43, 44
named pipes, 44, 228
ordinary file, 43
parent directory, 43
proc files, 44, 228–229

pseudo files, 43, 44, 228
special files, 44, 228–229
subdirectory, 43
subfolder, 43
Unix filesystem
`..` (parent directory), 49
`.` (current directory), 49
`~` (home directory), 48, 50
current directory, 48–52
directories, list of most important, 45
directory commands, list of most important, 51, 52
directory names, 52–55
dotfiles, 54–55, 185
file commands, list of most important, 51
file names, 53, 54
file permissions, read-only is not the same as Emacs **`read-only-mode`**, 181–182
filename, 46
Filesystem Hierarchy Standard (FHS), 47
`hier` command to display hierarchy, 45, 47
home directory, 46, 48
home directory contains **`.emacs`** file, 185–186
number of files on a Ubuntu Linux system, 229
parent directory, 49, 50
path, 45, 46
pathname, 46
pathname, absolute, 48, 49
pathname, relative, 48, 49
root directory, 45–48
tree-structured filesystem, 45–48
working directory, 48–50
Unix keys
command line editing (*see* Command line editing)
Macintosh keyboard, 35
making corrections, 34–35
sending the **`eof`** signal, 19, 36
sending the **`intr`** signal, 36
writing **`^`** for <Ctrl>, 72
Unix manual, 40
Unix shell. *See* Shell
Usenet, 16, 216
User interfaces
command-line interface (CLI), 25, 36
desktop, 25
desktop environment, 25, 26
focus, 25
Gnome desktop environment, 225–226
graphical user interface (GUI), 24, 25
KDE desktop environment, 225–226
text-based user interface (TUI), 25, 36, 225
Unity desktop environment, 29, 166, 227
Users, 13, 14
Utilities, 15–17

`vi`/Vim text editor (*see* Unix: **`vi`**/Vim text editor)
Virtual console. *See* Virtual terminals
Virtual terminals, 226–227
Virtual, meaning of, 226
Visit a file. *See* Files: visiting a file

Walter R. Brooks. *See* Freddy the Pig: Walter R. Brooks
Windows
commands to control windows, 113–115
compared to buffers, 113–115
creating, 113–115
current window, 93
definition, 91
deleting, 114, 115
echo area, 96–97
help window, maximizing, 184
mode line, 94–96

Windows (*cont.*)
mode line shows current major mode, 174
mode line status characters, 94
mode line, [ ] characters, recursive editing, 170
moving, 113
next window, 114
next window, scrolling through, 127
selected window, 93
sizing, 114
splitting, 115
using, 93
Windows, Microsoft. *See* Operating systems: Microsoft Windows
installing Emacs (*see* Installing Emacs: using Microsoft Windows)
Wisdom
"Everything we teach you is true…" (Harley Hahn), 139, 140
"Here is a way to make a bit of money for yourself…" (Harley Hahn), 139
"I have always thought of myself…" (Isaac Asimov), 159
"Much of what I know about human nature…" (Harley Hahn), 228
"My turn will come too, eventually…" (Isaac Asimov), 162
"Perhaps writers are so self-absorbed…" (Isaac Asimov), 158
"When computer experts need to talk…" (Harley Hahn), 226
"Who said Emacs was for beginners?" (Harley Hahn), 214
Emacs is a way of life., 6
Emacs is good for your brain., 14
Importance of teaching yourself Emacs., 215
Our tools shape our minds., 219
What are the best things in life?, 214
What to get your mother for her birthday., 192
Why I wrote this book., 214
Wonderful Mother's Day gift., 230
Working directory. *See* Unix filesystem: current directory

X Window, 23

Zip file. *See* Installing Unix software: zip file
Zippy the Pinhead Talks to the Emacs Psychotherapist, 212–213