A Tutorial Introduction to GNU Emacs

Introduction and History

What GNU Emacs Is

GNU Emacs is a free, portable, extensible text editor. That it is free means specifically that the source code is freely copyable and redistributable. That it is portable means that it runs on many machines under many different operating systems, so that you can probably count on being able to use the same editor no matter what machine you're using. That it is extensible means that you can not only customize all aspects of its usage (from key bindings through fonts, colors, windows, mousage and menus), but you can program Emacs to do entirely new things that its designers never thought of.

Because of all this, Emacs is an extremely successful program, and does more for you than any other editor. It's particularly good for programmers. If you use a common programming language, Emacs probably provides a mode that makes it especially easy to edit code in that language, providing context sensitive indentation and layout. It also probably allows you to compile your programs inside Emacs, with links from error messages to source code; debug your programs inside Emacs, with links to the source; interact directly with the language interpretor (where appropriate); manage change logs; jump directly to a location in the source by symbol (function or variable name); and interact with your revision control system.

Emacs also provides mail readers, news readers, World Wide Web, gopher, and FTP clients, spell checking, and a Rogerian therapist, all of which are also useful for programming. But in this document we'll concentrate on the basics of Emacs usage for programmers.

What GNU Emacs Is Not

First and foremost, GNU Emacs is not a WYSIWYG word processor. This is primarily because it's designed as a programmer's editor, but Emacs is also used to edit documents by people who use non-WYSIWYG typesetters like TeX and Troff.

Actually, this is about the only thing I can think of that GNU Emacs is not.

Varieties of Emacs

Emacs is actually the name of a family of text editors that are either descended from or inspired by one another. The original Emacs was written in the programming language of the text editor TECO, and ran on DEC PDP-10s and -11's. Early on it inspired other Emacsen for Multics, and Lisp Machines.

The first Emacs for Unix machines was Gosling Emacs, which later went commercial as Unipress Emacs. GNU Emacs was written by Richard Stallman, the main author of the original TECO Emacs.

For an editor to be called "emacs" the main requirement is that it be fully extensible with a real programming language, not just a macro language. For GNU Emacs, this language is Lisp. Other Emacsen have used TECO, Scheme, a dialect of Trac called Mint, interpreted C-like languages, etc.

GNU Emacs itself runs on a large number of Unix machines, and under VMS, DOS/Windows, and OS/2, among others. GNU Emacs is currently at version 19.29; at Library Systems we have this version installed. On the AIT machines (quads, ellis, kimbark and woodlawn) the default emacs is version 18.57: a major revision behind. Version 18 is probably 92% compatible with version 19; 19 mostly adds new functionality. But some important commands were changed slightly. (However, an *unsupported* version of Emacs v19 is available as `/usr/unsupported/gnu/bin/emacs19`.)

Theory and Practice of Keyboards and Character Sets

I find that, for beginners, the most confusing thing about Emacs is the keyboard and character set. It's important to understand that, as a portable program, Emacs can't always know about all the keys on your keyboard. All Emacs commands are 8-bit ASCII characters. Some keyboards, however, have keys which don't correspond well to any particular ASCII characters. This is especially true of PC and Macintosh keyboards, when connected to a Unix machine through the intermediary of some telecommunications program, where the only data that flows between your PC and the Emacs running on the Unix box are 8-bit ASCII bytes.

For example, the typical PC keyboard has keys labelled `PAGE UP` and `HOME`, arrow keys, function keys, etc. Not only do these keys have no official ASCII values, but they generate no ASCII characters at all. The PC can distinguish between them by means of *scan codes*, but what happens when you're connected to a Unix machine via a telecommunications program and you type one of these keys?

Any of the following things can happen, depending on the telecommunications program:

1. No ASCII character is sent at all;
2. One ASCII character is sent, but it's an arbitrary one likely to be different under another telecommunications program;
3. Some escape sequence is sent.

The characters generated by different keys can vary even within the same telecommunications program depending on what terminal emulation is chosen. In fact, the telecommunications program can even change the characters sent by keys that *do* have official ASCII characters associated with them (control-@ is a common problem).

Finally, the operating system and even terminal concentrators can muddy things further. Emacs (by default only) expects that the full ASCII character set is available. But the OS or some mux may be usurping some of your characters. The best examples are control-S and control-Q, which are sometimes used for flow control. These are important Emacs commands and you should make sure that neither your telecommunications program nor your terminal concentrator are grabbing them (Emacs can handle the OS). Other examples are nulls, sometimes swallowed up whole, and characters with the high-order bit set, sometimes stripped, or -- worse -- used for parity. Any internationalized system (like modern Unixes) work best with 8-bit no parity data channels anyway.

There's nothing Emacs can do about any of the above: it only sees 8-bit ASCII characters given to it by the operating system. But if you understand your telecommunications program, you won't have any problems with Emacs and keystrokes.

> **Note:** Emacs actually can use special keys like the arrow keys under certain circumstances (like under X or when running natively under DOS, where Emacs

understands keyboard *events*). But I recommend not using these keys even if they work, so that you can use Emacs from any terminal.

Notation

In the rest of this document I use the standard Emacs notation to describe keystrokes:

`C-x`
: For any *x*, the character Control-*x*.

`M-x`
: For any *x*, the character Meta-*x* (see below for more details on Meta characters).

`C-M-x`
: For any *x*, the character Control-Meta-*x*.

`RET`
: The return key (`C-m` actually).

`SPC`
: The space bar.

`ESC`
: The escape key, or, equivalently, `C-[`

Emacs Command Structure

For Emacs, every keystroke is actually a command, even simple keystrokes like the letters `A` and `z`: printing characters are commands to insert themselves. Non-printing characters are editing commands, which might move the cursor, scroll some text, delete or copy text, etc.

Every command has a long name, which you can look up in the documentation, like `kill-line`, `delete-backward-char`, or `self-insert-command`. These commands are *bound* to keystrokes for convenient editing. We call such a pairing of keystroke and command a *key binding*, or *binding* for short.

The set of all bindings make up the Emacs command set. However, Emacs is an extensible, customizable editor. This means that:

- Bindings can be different in different modes by virtue of extensibility
- Bindings can be different for different users by virtue of customizability

In this document I describe the standard Emacs key bindings.

Simple Keys

There are 95 different printable ASCII characters, and they are all bound to `self-insert-command` so that they insert themselves as text when typed. For editing commands, Emacs uses all the control characters: `C-a`, `C-b`, etc. But this is only 32 more characters, and Emacs has more than 32 editing commands.

The 128 characters in the upper half of ASCII are not taken yet, but how do you type them? Emacs uses a `Meta` key, which works exactly like a `Control` key or a `Shift` key in that it generates no character by itself, but rather modifies another character on the keyboard. The `Meta` key actually generates the same character that the key it's used with generates, but with the high-order bit set. This gives us access to characters such as `M-a`, `M-b`, etc. (There's also `M-A`, which is a distinct character, but to minimize confusion the uppercase metacharacters are equated to the corresponding lowercase metacharacter.)

What about the control characters with the high-order bit set? These are valid metacharacters as well; they are notated `C-M-a`, etc. To key them you hold down `Control` and `Meta` simultaneously and strike the desired key. Because both `Control` and `Meta` are shift keys, `C-M-a` is the same key (and the same ASCII character) as `M-C-a`. But for consistency we always write the former.

Prefix or Compound Keys

The `Control` and `Meta` keys plus the printing characters give us 256 possible keystrokes, or 160 editing commands after eliminating the self-inserting characters. But Emacs has many more than 160 commands! To handle this we also use *prefix* commands. A prefix command is a keystroke that, when typed, waits for another character to be typed, making a pair of keystrokes bound to one command. Each prefix command adds another 256 keystrokes that we can bind commands to. Prefix commands often group together commands that are somehow related.

The standard prefix commands are:

`C-c`
: Used for commands that are specific to particular modes, so they are free to be used for different commands depending on context. These are the most variable of Emacs commands.

`C-h`
: Used for Help commands.

`C-x`
: This prefix is used mostly for commands that manipulate files, buffers and windows.

These prefixes give us another 768 keystrokes, for a total 928. But Emacs has far more than 928 commands! To handle this, you can bind one of the subcommands of a prefix command to another prefix command, like `C-x 4` for example, or `C-x v`, each such binding yielding another 256 keystrokes. A number of these two-character prefixes exist, but they're rather specialized, and don't contain a full set of 256 commands (usually there are only three or four, and the prefix is just used for a mnemonic grouping). There are even three character prefixes, but most people won't admit to using them.

Using Extended Commands

But now we're entering a sort of rarefied atmosphere: even an Emacs geek like myself doesn't really use all these key bindings. Some Emacs commands are used very rarely, and, when you need it, it's easier to look up the long name of the command (using Info, Emacs' online searchable help system) and type it directly.

There's one Emacs command that can be used to execute any other command by typing it's long name: `M-x`. When you type `M-x` Emacs prompts you for the name of any command, and then executes it.

The ESC Prefix

There's one other prefix command that's both very important and completely redundant: the `ESC` prefix.

Not all keyboards provide a `Meta` key that sets the high order bit. On a PC running Emacs natively, the `ALT` key is used for `Meta`. But when using a PC to talk to a Unix box via some telecommunications program -- well, you guessed it -- the `ALT` key may not work for this.

But if you have no `Meta` key, all is not lost. You just use the `ESC` prefix. `M-a` becomes `ESC a`; `C-M-f` becomes `ESC C-f` (remember the equivalence of `C-M-f` and `M-C-f` and this will make sense).

There's only one trick: `ESC` is *not* a shift key. It's actually an ASCII character, not a key modifier. This means that you don't try to hold down `ESC` at the same time as the other key: use it as a prefix character and type it separately and distinctly. If you lean on it it's likely to autorepeat (like any other key) and you'll get very confused.

A true `Meta` is a wonderful thing for Emacs (it makes typing much faster), but I used `ESC` for years with no trouble.

Too Many Commands?

How does anyone remember all these commands? Simple: you don't. Every Emacs user knows a different set of commands. I've used Emacs for 15 years (starting with the original TECO Emacs), and I learn useful new Emacs commands all the time. Often I notice another Emacs user doing something and I have no idea how they've done it, so I ask and learn some Emacs command that I just never came across, or never developed as a habit.

Some Emacs users just learn the basics and are completely happy. Most users learn the basics and then some advanced commands that suit their needs. Some users are constantly learning new commands to speed their editing. A few users progress to writing their own totally new Emacs commands.

Files, Buffers and Windows

Emacs has three data structures (actually four) that are intimately related, and very important to understand:

File
: A file is the actual Unix file on disk. You are never editing this file. Rather, you can read a copy into Emacs to initialize a buffer, and write a copy of a buffer out to a file to save it.

Buffer
: A buffer is the internal data structure that holds the text you actually edit. Emacs can have any number of buffers active at any moment. Most, but by no means all, buffers are associated with a file. Buffers have names; a buffer that has been initialized from a file is almost always named for that file, and we say that the buffer is *visiting* the file. This means, in particular, that when you save the buffer, it's saved to the proper file. At any given time exactly one buffer is *selected*: this is the buffer that your hardware cursor is in, and this is where commands you type take effect (including self-insert commands). Buffers can be deleted at will; deleting a buffer in no way deletes the file on disk (though you may lose any editing changes you made if you don't save first).

Window
: A window is your view of a buffer. Due to limited screen real-estate, you may not have room to view all your buffers at once. You can split the screen, horizontally or vertically, into as many windows as you like (or at least have room for), each viewing a different buffer. It's also possible to have several windows viewing different portions of the same buffer. Windows can be created and deleted at will; deleting a window in no way deletes the buffer associated with the window. Each window has its own Mode Line, but there's still only one minibuffer.

Frame
: A frame is like a window, but is treated as a separate entity under a windowing system like X. I won't be discussing frames.

Commands to Manipulate Files

`C-x C-f`
: `find-file`. This is the main command used to read a file into a buffer for editing. It's actually rather subtle. When you execute this command, it prompts you for the name of the file (with completion). Then it checks to see if you're already editing that file in some buffer; if you are, it simply switches to that buffer and doesn't actually read in the file from disk again. If you're not, a new buffer is created, named for the file, and initialized with a copy of the file. In either case the current window is switched to view this buffer.

`C-x C-s`
: `save-buffer`. This is the main command used to save a file, or, more accurately, to write a copy of the current buffer out to the disk, overwriting the buffer's file, and handling backup versions.

`C-x s`
: `save-some-buffers`. Allows you to save all your buffers that are visiting files, querying you for each one and offering several options for each (save it, don't save it, peek at it first then maybe save it, etc).

Commands to Manipulate Buffers

`C-x b`
: `switch-to-buffer`. Prompts for a buffer name and switches the buffer of the current window to that buffer. Doesn't change your window configuration. This command will also create a *new* empty buffer if you type a new name; this new buffer will not be visiting any file, no matter what you name it.

`C-x C-b`
: `list-buffers`. Pops up a new window which lists all your buffers, giving for each the name, modified or not, size in bytes, major mode and the file the buffer is visiting.

`C-x k`
: `kill-buffer`. Prompts for a buffer name and removes the entire data structure for that buffer from Emacs. If the buffer is modified you'll be given an opportunity to save it. Note that this in no way removes or deletes the associated file, if any.

`C-x C-q`
: `vc-toggle-read-only`. Make a buffer read-only (so that attempts to modify it are treated as errors), or make it read-write if it was read-only. Also, if the files is under version control, it will check the file out for you.

Commands to Manipulate Windows

`C-v`
: `scroll-up`. The basic command to scroll forward (towards the end of the file) one screenful. By default Emacs leaves you two lines of context from the previous screen.

`M-v`
: `scroll-down`. Just like `C-v`, but scrolls backwards.

`C-x o`
: `other-window`. Switch to another window, making it the active window. Repeated invocation of this command moves through all the windows, left to right and top to bottom, and then circles around again. Under a windowing system, you can use the left mouse button to switch windows.

`C-x 1`
: `delete-other-windows`. Deletes all other windows except the current one, making one window on the screen. Note that this in no way deletes the buffers or files associated with the deleted windows.

`C-x 0`
: `delete-window`. Deletes just the current window, resizing the others appropriately.

`C-x 2`
: `split-window-vertically`. Splits the current window in two, vertically. This creates a new window, but *not* a new buffer: the same buffer will now be viewed in the two windows. This allows you to view two different parts of the same buffer simultaneously.

`C-x 3`
: `split-window-horizontally`. Splits the current window in two, horizontally. This creates a new window, but *not* a new buffer: the same buffer will now be viewed in the two windows. This allows you to view two different parts of the same buffer simultaneously.

`C-M-v`
: `scroll-other-window`. Just like `C-v`, but scrolls the *other* window. If you have more than two windows, the other window is the window that `C-o` would switch to.

Fundamental Concepts

It's probably more important to understand these fundamental Emacs concepts than it is to understand any of the actual editing commands. The editing commands are details: you can learn them easily on your own if you get the groundwork right.

Entering and Exiting

To enter Emacs, you just say:

```
emacs
```

when it comes up, you won't be editing any file. You can then use the file commands to read in files for editing. Alternatively, you can fire up Emacs with an initial file (or files) by saying:

```
emacs foo.tcl
```

To exit Emacs, use the command `C-x C-c` (which is bound to `save-buffers-kill-emacs`). It will offer to save all your buffers and then exit.

You can also *suspend* Emacs (in the Unix sense of stopping it and putting it in the background) with `C-x C-z` (which is bound to `suspend-emacs`). How you restart it is up to your shell, but is probably based on the `fg` command.

Self Inserting Commands

Once you've got Emacs running, you can type into it. There's no need for any special insert mode or anything like that: remember, printing characters insert themselves because each one is bound to `self-insert-command`.

The Screen

The Mode Line

The Emacs screen is completely devoted to the text of your file, except for one line near the bottom of the screen: the Mode Line. This line is informational: you can never move into it. It's almost always in reverse video or otherwise highlighted. It displays important information (which may change), including:

- The *state* of the buffer, one of modified (indicated by a pair of asterisks), unmodified (hyphens), or read-only (indicated by a pair of `%` signs).
- The name of the file you're editing (it will be `*scratch*` if you're not editing any file).
- The major mode (in parens).
- The amount of the file that you can see on the screen:
  All
  : You can see all of the file.
  Top
  : You can see the top of the file.
  Bot
  : You can see the bottom of the file.
  Percentage
  : NN% indicates the percentage of the file above the top of the window.

The Minibuffer

The blank line below the mode line is the *minibuffer*. The minibuffer is used by Emacs to display messages, and also for input when Emacs is prompting you to type something (it may want you to type yes or no in answer to a question, the name of a file to be edited, the long name of a command, etc).

The minibuffer is also known as the *echo area*, because Emacs echoes keystrokes here if you're typing really slowly. To see this, type any multi-character keystroke (like, `ESC q`) with a long pause between the keystrokes.

Strange Messages

Emacs will occasionally print messages in the minibuffer of its own accord, seemingly unrelated to what you're doing. The two most common messages are "Mark set" and "Garbage collecting...". The former means that Emacs has set the mark for you as a result of your last command; automatic mark setting is a convenient feature of some commands; see The Mark and The Region. The latter means that Emacs' lisp engine is reclaiming storage. You can just ignore it and keep typing, if you like: Emacs won't lose your characters.

Funny Characters

When your file contains control characters or characters with the high-order bit set, Emacs displays them as a backslash followed by three octal digits (where the digits give the ASCII value of the character). You'll notice that there's no ambiguity here: Emacs can tell the difference between a funny character and the four characters backslash digit digit digit. (You can't move the cursor between the digits of a funny character.) Note that if your terminal or windowing system is known to be able to display an 8-bit character set, like ISO Latin-1, Emacs will display those characters rather than the octal digits.

Long Lines

Emacs doesn't break lines for you automatically, unless you ask it to. By default it lets lines be as long as you type them. More importantly, it doesn't mess with long lines in files that you may read in. (Some editors, like old versions of vi, truncate long lines.)

It may seem annoying to have to hit return at the end of long lines, but this is actually just the default for certain modes. The reason for this is that Emacs is a programmer's editor, and any editor that will insert line breaks without your telling it to isn't safe for editing code or data. In modes oriented towards text, Emacs does insert line breaks for you automatically.

Eight Bit Clean

Emacs is eight bit clean, meaning you can edit binary files without corruption. (Some editors, like old versions of vi, corrupt certain characters, notably nulls and characters with the high bit set.)

Interrupting and Aborting

Sometimes Emacs will do something that you don't understand: it will prompt you for some information, or beep when you try to type, or something equally confusing. This just means that you've typed some command unwittingly (hitting a random function key is a good way to demonstrate this).

When this happens, you just need to type `C-g` (which is bound to `keyboard-quit`). This is the ASCII BEL character, and so `C-g` is sort of mnemonic for ringing the bell, which is what it does. But it also does something very important: it interrupts what Emacs is doing. This will get you out of any questions that Emacs may be asking you, and it will abort a partially typed key sequence (say if you typed C-x by mistake).

Because Emacs is fully recursive, you may occasionally need to type `C-g` more than once, to back out of a recursive sequence of commands. Also, if Emacs is really wedged (say, in a network connection to some machine which is down), typing three `C-g`'s quickly is guaranteed to abort whatever's wedging you.

Help

Emacs has extensive online help, most of which is available via the help key, `C-h`. `C-h` is a prefix key. Type `C-h` twice to see a list of subcommands; type it three times to get a window describing all these commands (a `SPC` will scroll this window). Some of the most useful help commands are:

`C-h a`
: `command-apropos`. Prompts for a keyword and then lists all the commands with that keyword in their long name.

`C-h k`
: `describe-key`. Prompts for a keystroke and describes the command bound to that key, if any.

`C-h i`
: `info`. Enters the Info hypertext documentation reader.

`C-h m`
: `describe-mode`. Describes the current major mode and its particular key bindings.

`C-h p`
: `finder-by-keyword`. Runs an interactive subject-oriented browser of Emacs packages.

`C-h t`
: `help-with-tutorial`. Run the Emacs tutorial. This is very helpful for beginners.

Info

Emacs has a builtin hypertext documentation reader, called *Info*. To run it, type `C-h i` or `M-x info RET`. It has it's own tutorial, which you should run the first time through by typing `h`. The tutorial assumes you understand about as much about Emacs as is covered in this document.

Infinite Undo with Redo

One of the most important Emacs commands is `undo`, invoked with `C-_` (control underbar). `C-_` is a valid ASCII character, but some keyboards don't generate it, so you can also use `C-x u` -- but it's more awkward to type, since it's a two-character command.

The undo command allows you to undo your editing, back in time. It's handy when you accidentally convert all of a huge file to uppercase, say, or delete a huge amount of text. One keystroke changes everything back to normal.

We say Emacs has infinite undo because, unlike some editors, you can undo a long chain of commands, not just one previous one, even undoing through saves. We say Emacs has redo because you can reverse direction while undoing, thereby undoing the undo.

Once you get used to this feature you'll laugh at any editor that doesn't have it (unless you're forced to use it...). It's very important to get comfortable with undo as soon as possible; I recommend reading the undo section of the manual carefully and practising.

Backups and Auto Save Mode

Emacs never modifies your file on disk until you tell it to, but it's very careful about saving your work for you in a number of ways.

Backup files.
: Emacs always saves the previous version of your file when you save. If your file is named `foo`, the backup will be called `foo~` (note the squiggle). Although it is off by default, Emacs will keep any number of previous versions for you, named `foo.~1~`, `foo.~2~`, etc. You can decide how many versions are to be kept. (But Unix provides more powerful tools for managing multiple versions of files.)

Auto-Save files.

Emacs also, be default, *auto-saves* your file while you're editing it. The auto-save file for a file `foo` is called `#foo#`. If Emacs (or the system) were to crash before you could save your edits, you can recover almost all of it from this file. Auto-saving happens (by default) every 300 characters, or when a system error is encountered.

Completion

To save you typing, Emacs offers various forms of *completion*: this means Emacs tries to complete for you partially typed file names, command names, etc. To invoke completion, you usually type `TAB`.

Giving Commands Arguments

Many Emacs commands take *arguments*, exactly the way a procedure or function takes arguments in a programming language. Most commands prompt you for their arguments: e.g., a command to read in a file will prompt you for the filename.

There's one kind of argument that's so commonly accepted that there's a special way to provide it: *numeric arguments*. Many commands will interpret a numeric argument as a request to repeat that many times. For example, the `delete-char` command (bound to `C-d`), which normally deletes one character to the right of the cursor, will delete *N* characters if given a numeric argument of *N*. It works with self-inserting commands too: try giving a numeric argument to a printing character, like a hyphen.

To give a command a numeric argument of, say, 12, type `C-u 12` before typing the command. If you type slowly, you'll see:

```
C-u 1 2-
```

in the echo area. Then type `C-d` and you'll have given `delete-char` an argument of 12. You can type any number of digits after `C-u`. A leading hyphen makes a negative argument; a lone hyphen is the same as an argument of -1. If you begin typing a numeric argument and change your mind, you can of course type `C-g` to abort it.

Since one often isn't interested in *precisely* how many times a command is repeated, there's a shorthand way to get numeric arguments of varying magnitudes. `C-u` by itself, without any subsequent digits, is equal to a numeric argument of 4. Another `C-u` multiplies that by 4 more, giving a numeric argument of 16. Another `C-u` multiplies *that* by 4 more, giving a numeric argument of 64, etc. For this reason `C-u` is called the `universal-argument`.

Note that commands aren't *required* to interpret numeric arguments as specifying repetitions. It depends on what's appropriate: some commands ignore numeric arguments, some interpret them as Boolean (the presence of numeric argument -- *any* numeric argument -- as opposed to its absence), etc. Read the documentation for a command before trying it.

Quoting Characters That Are Bound As Commands

Sometimes one needs to insert control characters into a file. But how can you insert an `ESC`, say, when it's used as a prefix command? The answer is to use the command `quoted-insert`, which is bound to `C-q`. `C-q` acts like a prefix command, in that when you type it it waits for you to type another character. But this next character is then inserted into the buffer, rather than being executed as a command. So `C-q ESC` inserts an `Escape`.

`C-q` can also be used to insert characters by typing `C-q` followed by their ASCII code as three octal digits.

Disabled Commands

Some commands that are especially confusing for novices are disabled by default. When a command is disabled, invoking it subjects you to a brief dialog, popping up a window displaying the documentation for the command, and giving you three choices:

- Space to try the command just this once, but leave it disabled,
- Y to try it and enable it (no questions if you use it again),
- N to do nothing (command remains disabled).

You're very likely to encounter one particular disabled command: `M-ESC` (aka `ESC ESC`), because it's very easy to type two escapes in a row when using the `Escape` prefix.

Motion and Objects

One of the main things one does in an editor is move around, in order to apply editing commands. Emacs provides many motion commands, which are arranged around *textual objects*: for each textual object, there is typically a motion command that moves forward over such an object and backward over it (or you can think of this as moving to the beginning and to the end).

All these motion commands take numeric arguments as repetitions.

The most basic textual object is the character. Emacs understand many other objects, sometimes depending on what mode you're in (a C function textual object probably doesn't make much sense if you're not editing C source code).

The exact definition of what makes up a given textual object is often customizable, but more importantly varies slightly from mode to mode. The characters that make up a word in Text Mode may not be exactly the same as those that make up a word in C Mode for example. (E.g., underbars are considered word constituents in C Mode, because they are legal in identifier names, but they aren't considered word constituents in Text Mode.) This is extremely useful, because it means that you can use the same motion commands and yet have them automatically customized for different types of text.

Characters

`C-f`
: `forward-char`. Moves forward (to the right) over a character.

`C-b`
: `backward-char`. Moves backward (to the left) over a character.

The f for forward and b for backward mnemonic will reoccur.

Words

`M-f`
: `forward-word`. Moves forward over a word.

`M-b`
: `backward-word`. Moves backward over a word.

Note the f/b mnemonic. Also, as another mnemonic, note that `M-f` is like a "bigger" version of `C-f`.

Lines (vertically)

`C-n`
: `next-line`. Moves down to the next line.

`C-p`
: `previous-line`. Moves up to the previous line.

When moving by lines, the cursor tries to stay in the same column, but if the new line is too short, it will be at the end of the line instead. This is very important: Emacs doesn't insert spaces at the ends of lines (end of line is unambiguous).

Lines (horizontally)

`C-a`
: `beginning-of-line`. Moves to the beginning of the current line.

`C-e`
: `end-of-line`. Moves to the end of the current line.

E for end, A for the beginning of the alphabet.

Sentences

`M-a`
: `backward-sentence`. Moves to the beginning of the current sentence.

`M-e`
: `forward-sentence`. Moves to the end of the current sentence.

Note the mnemonic relation between `C-a` / `M-a` and `C-e` / `M-e`.

Paragraphs

`M-{`
: `backward-paragraph`. Move to the beginning of the current paragraph.

`M-}`
: `forward-paragraph`. Move to the end of the current paragraph.

Pages

`C-x [`
: `backward-page`. Moves to the beginning of the current page.

`C-x ]`
: `forward-page`. Moves to the end of the current page.

Pages are separated by formfeed characters (`C-l`) in most modes.

Buffers

`M-<`
: `beginning-of-buffer`. Moves to the beginning of the buffer.

`M->`
: `end-of-buffer`. Moves to the end of the buffer.

S-Expressions (balanced parentheses)

An *S-expression* (*sexp* for short) is the name for balanced parentheses (and the text they enclose) in Lisp. In Emacs, this useful notion is available in most modes; it's especially useful for editing programming languages. The characters that Emacs recognizes as parens are usually regular parentheses (aka round brackets), square brackets, and braces (aka curly brackets), but it depends on the mode (for some languages, angle brackets may act as parens).

But sexps are more than just balanced parens: they're defined recursively. A word that doesn't contain any parens also counts as a sexp. In most programming language modes, quoted strings are sexps (using either single or double quotes, depending on the syntax of the language). The sexp commands move in terms of all these units.

These commands may seem confusing at first, but for editing most programming languages they're fantastic. Not only do they move you around quickly and accurately, but they help spot syntax errors while you're editing, because they'll generate an error if your parens or quotes are unbalanced.

`C-M-b`
: `backward-sexp`. Moves backward over the next sexp. If your cursor is just to the right of a left paren, `C-M-b` will beep, because there's no sexp to the left to move over: you have to move *up*.

`C-M-f`
: `forward-sexp`. Moves forward over the next sexp. Same deal if your cursor is just to the left of a right paren.

`C-M-u`
: `backward-up-list`. Move backward up one level of parens. In other words, move to the left paren of the parens containing the cursor, skipping balanced sexps.

`C-M-d`
: `down-list`. Move down one level of parens. In other words, move to the right of the next left paren, skipping balanced sexps. E.g., if your cursor is sitting on the return type of a C function declaration, `C-M-d` moves to the inside of the formal parameter list.

Functions

Since functions are such an important unit of text in programming languages, whether they're called functions, subroutines, procedures, procs, defuns or whatever, Emacs has commands to move over them. Like the sexp commands, these commands work appropriately in most programming language modes. Emacs calls this generic notion of function or procedure *defun*, again after Lisp.

`C-M-a`
: `beginning-of-defun`. Move to the beginning of the current defun.

`C-M-e`
: `end-of-defun`. Move to the end of the current defun.

Note the mnemonic analogy with lines and sentences.

Deleting, Killing and Yanking

Emacs' deletion commands are also based on the textual objects above. But first, a terminological distinction: *Deletion* means to remove text from the buffer without saving it; most deletion commands operate on small amounts of text. *Killing* means to save the removed text, so that it can be yanked back later someplace else.

Killed text is saved on the *kill ring*. The kill ring holds the last *N* kills, where *N* is 30 by default, but you can change it to anything you like by changing the value of the variable *kill-ring-max*. The kill ring acts like a fifo when you're killing things (after the 30th kill, kill number one is gone), but like a ring when you're yanking things back (you can yank around the ring circularly). *kill-ring-max* doesn't apply to the amount of text (in bytes) that can be saved in the kill ring (there's no limit), only to the number of distinct kills.

Characters

`C-d`
: `delete-char`. Deletes the character to the right of (under, if the cursor is a block that covers a character) the cursor.

`DEL`
: `delete-backward-char`. Deletes the character to the left of the cursor.

Remember, Emacs does *not* use `C-h` (aka `Backspace`) for deletion, but for help! Depending on your cultural background, this may well be the single hardest thing to learn about Emacs. Why does Emacs make this choice? Well, since there's no need for two commands to delete one character backward, and since `DEL` (aka `Delete`) can only be mnemonic for deletion while `C-h` is a nice mnemonic for help, the choice was made. Me, I've never used `C-h` to delete, so it suits me fine.

Words

`M-d`
: `kill-word`. Kills to the end of the word to the right of the cursor (forward).

`M-DEL`
: `backward-kill-word`. Kills to the beginning of the word to the left of the cursor (backward).

Lines (vertically)

It doesn't seem too intuitive to kill lines vertically by analogy with `C-n` and `C-p`; I know of no such commands.

Lines (horizontally)

`C-k`

`kill-line`. Kills to the end of the current line, not including the newline. Thus, if you're at the beginning of a line it takes two `C-k`'s to kill the whole line and close up the whitespace.

`C-u 0 C-k`
: `kill-line`. Kills to the beginning of the current line, not including the newline.

You might think that `C-u -1 C-k` would be used to kill to the beginning of the line, and it does, but it includes the newline before the line as well.

Sentences

`M-k`
: `kill-sentence`. Kills to the end of the current sentence, including any newline within the sentence.

`C-u -1 M-k`
: `kill-sentence`. Kills to the beginning of the current sentence, including any newlines within the sentence.

Paragraphs

The commands `forward-kill-paragraph` and `backward-kill-paragraph` exist, but are not bound to any keys by default.

Pages

There are no commands to kill pages (but see The Mark and the Region).

Buffers

The command `kill-buffer` doesn't kill all the text in the buffer, but rather the entire buffer data structure; see The Mark and the Region.

S-Expressions (balanced parentheses)

`C-M-k`
: `kill-sexp`. Kills the sexp after the cursor.

`C-u -1 C-M-k`
: `kill-sexp`. Kills the sexp before the cursor.

The command `backward-kill-sexp` exists, but is not bound to any key by default.

Functions

There are no commands to kill functions (defuns) (but see The Mark and the Region).

Extending Kills

If you kill several times in a row, with any combination of kill commands, but without any *non*-kill commands in between, these kills are appended together in one entry on the kill ring. For example you can kill a block of text as several lines by saying `C-u 6 C-k`, which kills (as one kill) 6 lines.

Yanking

Once you've killed some text, how do you get it back? You can yank back the most recently killed text with `C-y` (`yank`). Since Emacs has only one kill ring (as opposed to one per buffer), you can kill in one buffer, switch to another and yank the text there.

To get back previous kills, you move around the kill ring. Start with `C-y` to get the most recent kill, and then use `M-y` to move to the previous spot in the kill ring by replacing the just-yanked text with the previous kill. Subsequent `M-y`'s move around the ring, each time replacing the yanked text. When you reach the text you you're interested in, just stop. Any other command (a motion command, self-insert, anything) breaks the cycling of the kill ring, and the next `C-y` yanks the most recent kill again.

Copying and Moving Text

Emacs has no need for special commands to copy or move text; you've already learned them! To move text, just kill it and yank it back elsewhere. To copy text, kill it, yank it back immediately (so it's as if you haven't killed it, except it's now in the kill ring), move elsewhere and yank it back again. For commands to copy and move arbitrary regions of text, as opposed to textual objects, see The Mark and The Region.

Searching and Replacing

Emacs has a variety of unusual and extremely powerful search and replace commands. The most important one is called *incremental search*. This is what the command `isearch-forward`, bound to `C-s`, does: it searches incrementally, one character at a time, as you type the search string. This means that Emacs can often find what you're looking for before you have to type the whole thing. To stop searching, you can either hit `RET` or type any other Emacs command (which will both stop the search and execute the command). You can search for the next match at any point by typing another `C-s` at any point; you can reverse the search by typing `C-r`; and you can use `DEL` to delete and change what you're searching for.

`isearch-backward`, bound to `C-r`, works the same way, but searches backward. (Use `C-r` to search for the next match and `C-s` to reverse the search.)

Occasionally you may want to search non-incrementally (though I rarely do). You can do this by typing `C-s RET text RET`, where `text` is the text to search for.

Much more useful is *word search*, which lets you search for a sequence of one or more words, regardless of how they're separated (e.g, by any number and combination of newlines and whitespace). To invoke word search, type `C-s RET C-w word word word RET`.

Emacs can also search incrementally (or not) by regular expressions; this is an extremely powerful feature, but too complex to describe here.

Replacement

Emacs' most important command for replacing text is called `query-replace` (bound to `M-%`). This command prompts you for the text to replace, and the text to replace it with, and then searches and replaces within the current buffer. `query-replace` is interactive: at each match, you are prompted to decide what to do; you have the following options:

`SPC`
: Perform this replacement.

`DEL`
: Don't perform this replacement.

`RET`
: Terminate `query-replace` without performing this replacement.

`ESC`
: Same as `RET`.

`.`
: Perform this replacement but then terminate the `query-replace`.

`!`
: Perform this replacement and all the rest in the buffer without querying (ie unconditionally).

There are many more subcommands, but they require more Emacs expertise to understand them.

There are also more replacement commands you should look into, including `replace-string` (simple unconditional replacement), `replace-regexp` and `query-replace-regexp` (which use regular expressions), and `tags-query-replace`, which replaces all identifiers in a collection of source code files.

`query-replace` and the other replacement commands are, by default, smart about case. For example, if you're replacing `foo` with `bar` and find `Foo`, Emacs replaces it with `Bar`; if you find `FOO`, Emacs replaces it with `BAR`, etc.

The Mark and The Region

Emacs can manipulate arbitrary chunks of text as well as distinct textual objects. The way this is done is to define a *region* of text; many commands will operate on this region.

The region is the text between *point* and *mark*. Point is actually the Emacs term for what we've been calling the cursor up to now. The mark, on the other hand, is set with a special command `C-@` (`set-mark-command`). This sets the mark exactly where point is, but now you can move point elsewhere and you have: the region.

Each buffer has a distinct point and mark, and therefore a distinct region. (It's also possible for there to be no mark in a buffer, and therefore no region.)

The region is the same regardless of whether point comes first in the buffer or mark does; it makes no difference, just do what's convenient.

The region is normally invisible (but see `C-x C-x`). You'll get used to this. However, if you're running Emacs under a windowing system, you can make the region visible by executing `M-x transient-mark-mode`.

Many commands that move point a significant distance (like `M-<` and `C-s`, for example) leave the mark set at the spot they moved from. You'll see "Mark set" in the echo area when this happens.

When using Emacs under a windowing system like X, the mouse can be used to sweep out the region, but many Emacsers find it faster to keep their hands on the keyboard and use the familiar motion commands.

There are some special commands that are specifically designed to set the region around some interesting text.

`M-@`
: `mark-word`. Sets the region around the next word, or from point to the end of the current word, if you're in the middle of one.

`M-h`
: `mark-paragraph`. Sets the region around the current paragraph.

`C-M-@`
: `mark-sexp`. Sets the region around the same sexp that `C-M-f` would move to.

`C-M-h`
: `mark-defun`. Sets the region around the current defun.

`C-x C-p`
: `mark-page`. Sets the region around the current page.

`C-x h`
: `mark-whole-buffer`. Sets the region around the entire buffer.

So now you know how to define the region: what can you do with it?

`C-x C-x`
: `exchange-point-and-mark`. Swaps mark and point. Repeated rapid execution of this command makes it easy to see the extent of the region.

`C-w`
: `kill-region`. Kills the region. It goes on the kill ring, of course.

`M-w`
: `kill-ring-save`. Saves the region to the kill ring without removing it from the buffer. This is exactly equivalent to typing `C-w C-y`.

`C-x C-i`
: `indent-rigidly`. Rigidly indents the region by as many characters (columns) as you provide as a numeric argument (default is 1 column).

`C-x C-l`
: `downcase-region`. Convert the entire region to lowercase. This command is disabled by default.

`C-x C-u`
: `upcase-region`. Convert the entire region to uppercase. This command is disabled by default.

`M-x fill-region`
: `fill-region`. Fills, i.e., justifies with a ragged right margin, all the paragraphs within the region.

There are many, many more.

Indentation

In programming language modes, Emacs uses the `TAB` key to indent a line automatically, in accordance with indentation rules for your programming language. For example, in C Mode Emacs understands if, while, do, for, functions, switch, etc and indents appropriately. Of course, there are no hard and fast rules for indentation in most languages, so Emacs allows you to customize the indentation for your own style.

Modes

The main way Emacs customizes commands for different kinds of text is through major and minor *modes*. Every buffer has a major mode, and may have zero or more minor modes. Sometimes Emacs chooses a major mode for you automatically, typically based on a file extension (e.g., files ending in `.c` will automatically be in C Mode; files ending in `.tcl` will automatically be in Tcl Mode). But you can always set the mode explicitly.

Some Major Modes

`Fundamental Mode`
: The basic mode in reference to which all specialized modes are defined. Perfectly fine for editing any kind of text, just doesn't provide any special features.

`Text Mode`
: For editing text. Has special commands for spell-checking, centering lines, etc.

`Outline Mode`
: For editing a stylized type of outline. Implements folding of outline levels, etc.

`Lisp Mode`
: For editing Common Lisp source code. Has an interactive link to a Common Lisp interpreter in another buffer.

`Tcl Mode`
: For editing Tcl source code. Has an interactive link to a Tcl interpreter in another buffer.

`C Mode`
: For editing C source code. Has special indentation, etc.

There are many other major modes, some very specialized (e.g., modes for editing sending email, reading Usenet news, browsing directories, browsing the World Wide Web, etc).

Further Information

GNU Emacs Frequently Asked Questions List

The GNU Emacs FAQ is very well done; I recommend it highly.

Info

Don't forget that the complete text of the *GNU Emacs Manual* is available via Info, Emacs' hypertext documentation reader.

Usenet Newsgroups

Only a selection of some of the Emacs-related Usenet newsgroups.

gnu.emacs.help
: Help for users' GNU Emacs problems, answered by your peers.

comp.emacs
: General coverage of all Emacs-like editors.

gnu.emacs.announce
: Announcements of new versions of GNU Emacs, etc.

alt.religion.emacs
: Emacs as religion. Official home of the perennial flame wars.

gnu.emacs.sources
: Postings of source code for new GNU Emacs programs.

Bibliography

- Richard M. Stallman. *GNU Emacs Manual.* Cambridge, MA: Free Software Foundation, [n.d.]. The authoritative user's reference; also a fine introduction. The complete text is available on the Web and via Info.
- Debra Cameron and Bill Rosenblatt. *Learning GNU Emacs.* Sebastopol, CA: O'Reilly and Associates, 1991. A good introduction. Covers version 18 only.
- Michael A. Schoonover et al. *GNU Emacs: Unix Text Editing and Programming.* Reading, MA: Addison-Wesley, 1992.
- Bill Lewis, Dan LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual.* Cambridge, MA: Free Software Foundation, 1993. Covers version 19 Elisp. Only if you want to learn to program in Emacs Lisp. The complete text is available on the Web and via Info.
- Richard M. Stallman. "EMACS: The Extensible, Customizable, Self-Documenting Display Editor". In *Interactive Programming Environments*, edited by David R. Barstow, Howard E. Shrobe, and Erik Sandewall. New York: McGraw-Hill, 1984. An interesting article on the design of Emacs. Predates GNU Emacs; covers the original TECO Emacs and Lisp Machines Emacsen.

Keith Waclena `<k-waclena@uchicago.edu>`
The University of Chicago Library

This page last updated: Mon Jul 25 18:23:02 CDT 2005
This page was generated from Extended HTML by xhtml.